25

JavaBeans

Objetivos

- Entender os JavaBeans e como eles facilitam a construção de software orientado a componentes.
- Ter uma visão geral do *JavaBeans Development Kit*.
- Utilizar o contêiner de teste **BeanBox** para modificar propriedades de *beans* e associar *beans* através de eventos.
- Empacotar definições de classe em arquivos JAR para utilizar como aplicativos JavaBeans e aplicativos independentes *(stand-alone)*.
- Definir propriedades e eventos de JavaBeans.

Os espelhos deveriam refletir um pouco antes de realmente refletir as imagens.
Jean Cocteau

A televisão é como a invenção do encanamento interno. Não alterou os hábitos das pessoas. Apenas as manteve dentro de casa.
Alfred Hitchcock

And this is good old Boston,
The home of the bean and the cod,
Where the Lowells talk only to Cabots,
And the Cabots talk only to God.
Toast at Holy Cross alumni dinner, 1910

Um escritor é como um pé de feijão —
Ele começa pequeno e então cresce vigorosamente.
E. B. White

O poder do visível é o invisível.
Marianne Moore

A linguagem são os arquivos da história.
Ralph Waldo Emerson

Sumário do capítulo

25.1 Introdução

Neste capítulo, apresentamos o modelo de componente de software reutilizável de Java — os *JavaBeans*. Os JavaBeans (muitas vezes chamados simplesmente de *beans)* permitem que os desenvolvedores colham os benefícios do desenvolvimento rápido de aplicações em Java montando componentes predefinidos de software para criar aplicativos e *applets* poderosos. Os *ambientes de programação e projeto gráficos* [freqüentemente chamados de *ferramentas de desenvolvimento (builder tools)*] que suportam *beans* fornecem aos programadores tremenda flexibilidade ao permitir que eles reutilizem e integrem diferentes componentes existentes que em muitos casos não foram concebidos para serem utilizados em conjunto. Esses componentes podem ser associados para criar *applets*, aplicativos ou ainda novos *beans* para reutilizar por outros.

Os JavaBeans e outras tecnologias baseadas em componentes levaram a um novo tipo de programador — o *montador de componentes* — que utiliza componentes bem-definidos para criar funcionalidades mais robustas. Os montadores de componentes não precisam conhecer os detalhes de implementação de um componente. Em vez disso, o montador de componentes precisa conhecer os serviços fornecidos por um componente para poder fazê-lo interagir com outros componentes. Freqüentemente, os montadores de componentes estão mais preocupados com o projetos da interface gráfica com o usuário de um aplicativo ou com a funcionalidade que o aplicativo fornece para um usuário.

Como um exemplo do conceito de *bean* (freqüentemente dizemos "*bean*" quando mais precisamente deveríamos dizer "JavaBean"), imagine um montador de componentes que dispõe de uma animação *bean* contendo os métodos **`startAnimation`** e **`stopAnimation`**. O montador de componentes pode querer fornecer dois botões — um que irá iniciar a animação e outro que irá parar a animação (um exemplo que você verá mais adiante neste capítulo). Com *beans*, podemos simplesmente "enganchar" um botão no método **`startAnimation`** da animação e "enganchar" um botão no método **`stopAnimation`** da animação de tal modo que, quando um botão for pressionado, o método apropriado do *bean* de animação é chamado. A ferramenta de desenvolvimento faz todo o trabalho de associação do evento de pressionar o botão com o método apropriado para chamar o *bean* de animação. Tudo que o programador precisa fazer é dizer à ferramenta de desenvolvimento qual dos dois componentes deve ser associado.

O benefício de *beans* nesse exemplo é que o *bean* de animação e os *beans* de botão não precisam saber um do outro antes de serem montados em uma ferramenta de desenvolvimento.

Uma outra pessoa pode ser responsável pela definição de conceito de um botão de uma maneira reutilizável (como é feito com os componentes **javax.swing**). Um botão não é específico para nosso exemplo. Em vez disso, é um componente utilizado em muitos aplicativos e *applets*. Quando o usuário de um programa pressiona um botão, ele espera que ocorra uma ação específica com esse programa (alguns botões, como um botão **OK**, em geral tem o mesmo significado em todos os programas). Mas o conceito básico de um botão — como ele é exibido, como ele funciona e como ele notifica outros componentes de que foi pressionado — é o mesmo em cada aplicativo (embora em geral personalizemos seu rótulo). O trabalho do montador de componentes não é criar o conceito de um botão, mas sim utilizar o componente preexistente de botão para fornecer funcionalidade para o usuário do programa.

Os montadores de componentes podem fazer com que os *beans* se comuniquem por meio de serviços bem-definidos dos *beans* (isto é, métodos), em geral sem escrever nenhum código (com freqüência o código é gerado pela ferramenta de desenvolvimento e às vezes ainda oculto do programador — dependendo da ferramenta). De fato, um montador de componentes freqüentemente pode criar aplicativos complexos literalmente ligando os pontos.

Mostraremos como utilizar *beans* existentes e como criar seus próprios *beans* básicos. Depois de estudar este capítulo, você terá um entendimento sobre a programação de JavaBeans que lhe permitirá desenvolver rapidamente aplicativos e *applets* que utilizam os recursos mais avançados de ambientes de desenvolvimento integrados que suportam *beans*. Você também terá um entendimento consistente para posterior estudo sobre o JavaBeans.

Para mais informações, visite o site da Web da Sun Microsystems para JavaBeans:

```
http://java.sun.com/beans/
```

Esse site fornece um conjunto completo de recursos para aprender e utilizar JavaBeans.

25.2 Visão geral da BeanBox

Esta seção introduz a ***BeanBox*** — um utilitário do *JavaBeans Development Kit (BDK)* que pode ser descarregado gratuitamente do site da Web da Sun Microsystems

```
http://java.sun.com/beans/software/index.html
```

Na época desta publicação, a versão mais recente do BDK era BDK 1.1 de abril, 1999. Há várias versões disponíveis para descarregar, incluindo a versão de Windows, a versão de Solaris e uma versão que pode ser utilizada em qualquer plataforma que suporte J2SDK 1.2. Se você estiver utilizando este livro, você já deve ter J2SDK 1.2 (ou superior). O site também fornece acesso à versão anterior do BDK para utilizar com o JDK 1.1 (a versão anterior da implementação de Java pela Sun). [*Nota:* na época em que este livro foi originalmente publicado, havia uma falha menor no programa de instalação para a versão de Windows do BDK. O BDK não executará corretamente a partir de sua localização de instalação padrão (**C:\Program Files\BDK1.1**). Ao executar o programa de instalação, simplesmente remova "**Program Files**" da localização de instalação padrão (ou escolha qualquer caminho do diretório que não inclua caracteres de espaço) e o BDK executará adequadamente.]

A **BeanBox** é um *contêiner de teste* para seu JavaBeans. Ela é projetada para permitir que os programadores visualizem como será exibido e manipulado um *bean* que eles criaram em uma *ferramenta de desenvolvimento.* Entretanto, ela não se destina a ser utilizada como uma ferramenta de desenvolvimento robusta. Esta seção apresenta uma visão geral dos recursos da **BeanBox** e vários *beans* de demonstração fornecidos com a **BeanBox**.

Dica de teste e de depuração 25.1

*A **BeanBox** pode ser utilizada para testar e depurar JavaBeans.*

Observação de engenharia de software 25.1

*A **BeanBox** não é uma ferramenta de desenvolvimento. Em vez disso, a **BeanBox** permite que os programadores visualizem como um bean será exibido e utilizado por uma ferramenta de desenvolvimento.*

Depois de instalar o BDK, você pode executar a **BeanBox** posicionando o *diretório* ***BDK1.1*** onde você instalou o *JavaBeans Development Kit*. Nesse diretório, há um *subdiretório* ***beanbox***. Outro subdiretório importante é ***doc***, onde se encontram os arquivos de HTML de ajuda on-line para a **BeanBox** (utilize o arquivo ***beanbox.html*** para começar). Os arquivos de ajuda discutem profundamente todos os recursos da **BeanBox**.

O diretório **beanbox** contém os arquivos de inicialização para Windows (**run.bat**) e Solaris (**run.sh**). Para executar a **BeanBox**, execute o arquivo adequado para seu sistema operacional — no Windows, dê um clique duplo em **run.bat** no Windows Explorer; no Solaris, execute o script de *shell* **run.sh** a partir de seu *shell* de

comandos. [*Nota:* se você não estiver utilizando um desses sistemas operacionais, você terá de executar a **BeanBox** como você normalmente executaria um aplicativo Java em sua plataforma.]

Esta seção utiliza capturas de tela de um ambiente Windows. A Fig. 25.1 mostra as quatro janelas da **BeanBox** — **ToolBox**, **BeanBox** , **Properties** e **Method Tracer** — que aparecem quando o aplicativo é iniciado.

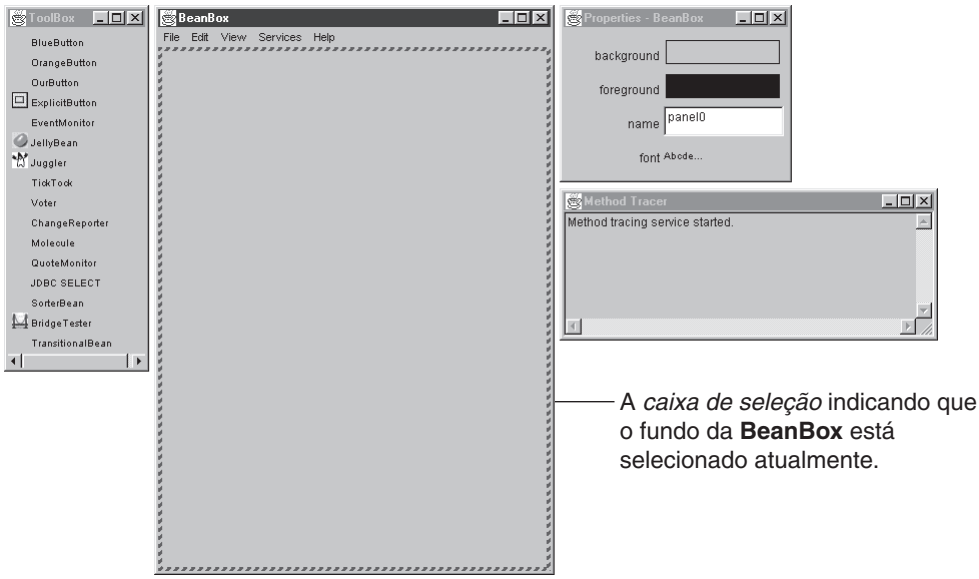


Fig. 25.1 As janelas **ToolBox**, **BeanBox**, **Properties** e **Method Tracer** do **BeanBox**.

A janela **ToolBox** contém 16 JavaBeans de demonstração que você pode utilizar para aprender sobre **BeanBox** e os princípios básicos de programação com JavaBeans. Eles também podem ser utilizados para interagir com qualquer novo *bean* que você estiver testando. A janela **BeanBox** é utilizada para testar um *bean*. A janela **Properties** permite que o programador personalize o *bean* atualmente selecionado. As ferramentas de desenvolvimento freqüentemente referenciam a janela **Properties** (ou sua versão dessa janela) como *folha de propriedades* (*property sheet*). A janela **Method Tracer** exibe mensagens simples de depuração e ajuda a rastrear chamadas de método (não trabalhamos essa janela neste livro).

Inicialmente, o fundo da janela **BeanBox** está *selecionado,* como indica o tracejado da *caixa de seleção* em torno do fundo da janela na Fig. 25.1. O fundo da janela da **BeanBox** é, na verdade, um objeto da classe **`java.awt.Panel`**. As propriedades do *bean* atualmente selecionado que podem ser manipuladas pelo programador são exibidas na janela **Properties**. Para o painel de fundo da janela **BeanBox**, as propriedades que podemos configurar são ***background***, ***foreground***, ***name*** e ***font***.

Clique no retângulo à direita da propriedade ***background*** na janela **Properties** para personalizar a cor de fundo do painel de fundo da **BeanBox**. Isso exibe o *editor de propriedades* ***ColorEditor*** (Fig. 25.2). Um editor de propriedades permite que o programador personalize um valor de propriedade.

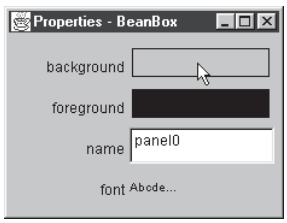
Selecione **yellow** como a cor de fundo. Observe que a cor de fundo da janela da **BeanBox** é alterada para amarelo imediatamente. Clique em **Done** na janela do **ColorEditor** para fechar o diálogo do **ColorEditor**.

Em seguida, colocaremos um JavaBean na janela da **BeanBox**. Na janela **ToolBox**, clique no *bean* do **`ExplicitButton`**. O cursor de seu mouse deve mudar para um cursor em forma cruz. Posicione o cursor do mouse sobre a janela da **BeanBox** onde você quer que o centro de seu *bean* seja localizado e clique com o mouse para posicionar o *bean*. A Fig. 25.3 mostra a janela da **BeanBox** com o novo *bean* **`ExplicitButton`** selecionado e a janela **Properties** contendo as propriedades do **`ExplicitButton`**. Se você clicar no botão, ele parecerá funcionar exatamente como um **`JButton`** funcionou anteriormente neste livro.

A janela **Properties** *expõe as propriedades do bean*. As propriedades expostas do *bean* podem ser alteradas durante o projeto em um ambiente de desenvolvimento a fim de personalizar o *bean* para utilizar em um programa específico.

Observação de engenharia de software 25.2

Um benefício de trabalhar em um ambiente de desenvolvimento de beans prontos é que o ambiente apresenta visualmente as propriedades do bean para o programador a fim de facilitar sua modificação e personalização durante o projeto.



Clique no retângulo à direita da propriedade **background** na janela **Properties** (à esquerda) para abrir o editor de propriedades **ColorEditor** (à direita).

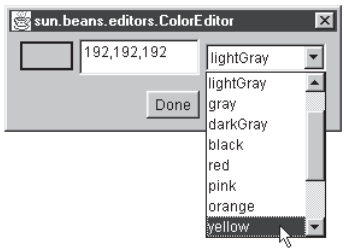


Fig. 25.2 Alterando a propriedade **background** para a **BeanBox**.

Na janela **Properties**, um das propriedades modificáveis é o ***label*** do **`ExplicitButton`**. Como utilizaremos esse botão para iniciar uma animação, gostaríamos de mudar sua propriedade **`label`** para "**`Start the Animation`**" em vez do valor atual "**`press`**". Clique no campo de texto à direita da propriedade **`label`** e mude o texto no botão para "**`Start the Animation`**". Pressione a tecla *Enter* quando estiver pronto. Observe que o rótulo no botão se altera e o botão se redimensiona automaticamente para ajustar o novo rótulo, que é mais longo que o rótulo original. A Fig. 25.4 mostra o botão e seu novo rótulo.

A seguir, gostaríamos de mover o botão. Se o botão não estiver selecionado (isto é, a caixa de seleção não aparece em torno do botão), clique no botão para selecioná-lo. [*Nota:* se o clique em um *bean* na janela da **BeanBox** não o seleciona, você pode precisar simplesmente clicar fora do limite do *bean* para selecioná-lo.] Posicione o cursor do mouse sobre a borda superior, inferior, esquerda ou direita da caixa de seleção do botão. O *cursor de movimentação* deve aparecer como mostra a Fig. 25.5.

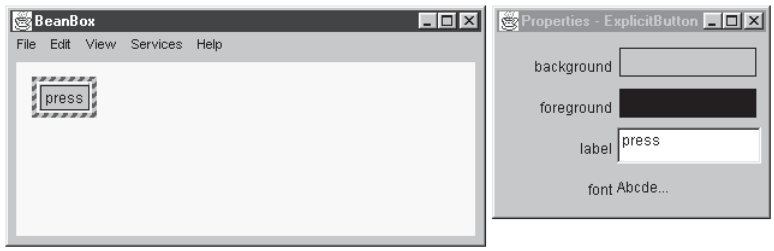


Fig. 25.3 A janela **BeanBox** com o *bean* **`ExplicitButton`** selecionado.

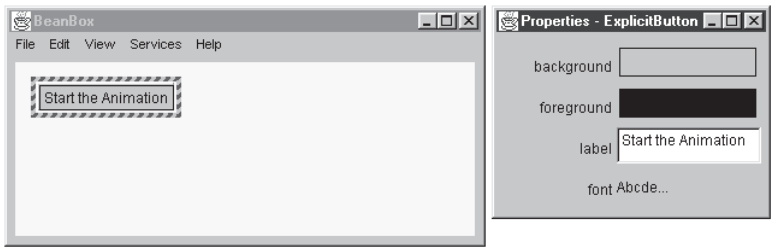


Fig. 25.4 O **`ExplicitButton`** com seu novo rótulo.

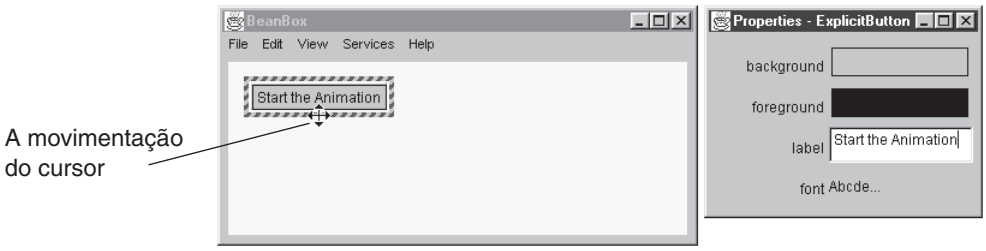


Fig. 25.5 A movimentação do cursor.

Mantenha o botão do mouse pressionado e arraste o botão na tela para o lado direito da janela da **BeanBox**. A janela deve ficar parecida com a Fig. 25.6. Observe que, enquanto você arrasta o botão, um retângulo vermelho indicando os limites do botão segue o cursor do mouse. Isso fornece um feedback visual que o ajuda a mover ou redimensionar um *bean*.

Em seguida gostaríamos de redimensionar o botão. Se o botão não estiver atualmente selecionado (isto é, a caixa de seleção não aparece em torno do botão), clique no botão para selecioná-lo. Posicione o cursor do mouse sobre o canto superior esquerdo, superior direito, inferior esquerdo ou inferior direito da caixa de seleção para o botão. O *cursor de redimensionamento* deve aparecer como mostra a Fig. 25.7.

Mantenha o botão do mouse pressionado e arraste o canto da caixa de seleção para aumentar a altura do botão. A janela deve aparecer como na Fig. 25.8. Observe que, enquanto você arrasta o mouse, um retângulo vermelho que indica os novos limites do botão segue o cursor do mouse. Isso fornece feedback visual que ajuda a redimensionar o botão com precisão.

Em seguida, adicione outro botão na **BeanBox**. Esse botão será utilizado para interromper a animação. Repita os passos precedentes para criar outro `ExplicitButton` com o rótulo "`Stop the Animation`". A Fig. 25.9 mostra a janela da **BeanBox** com os dois botões.

Em seguida, adicionaremos um *bean* de animação na **BeanBox**, para personalizar o *bean* e configurar os eventos de botão que permitirão iniciar e parar a animação. Na **ToolBox**, selecione o *bean* `Juggler` e adicione-o à janela **BeanBox**. A janela **BeanBox** e as propriedades do *bean* `Juggler` são mostradas na Fig. 25.10.

Observe que a animação de `Juggler` inicia imediatamente depois que o *bean* `Juggler` é *solto* sobre a janela da **BeanBox** (isto é, depois que você clica na **BeanBox** para posicionar o *bean*).

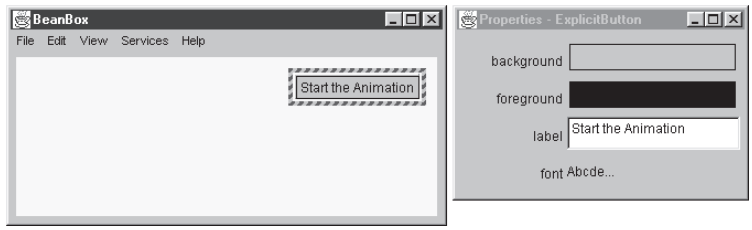


Fig. 25.6 O `ExplicitButton` depois de ser movido.

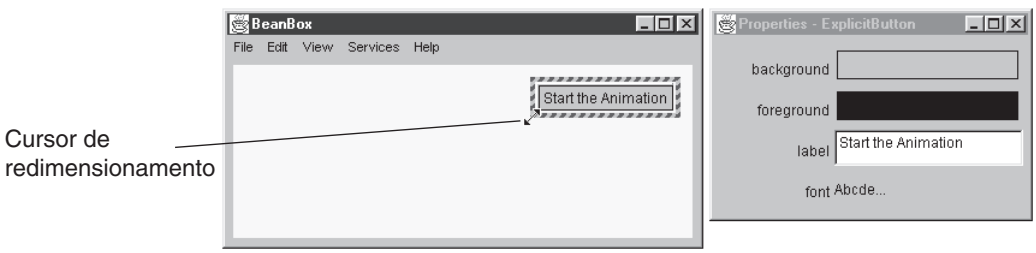


Fig. 25.7 O cursor de redimensionamento.

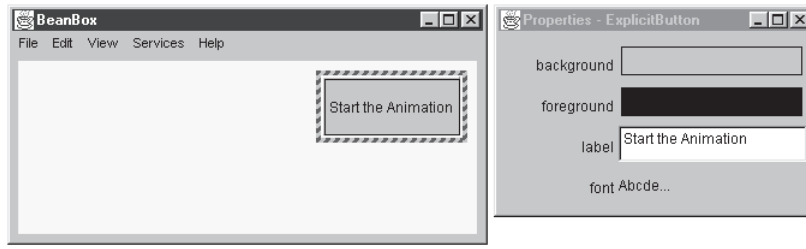


Fig. 25.8 O **`ExplicitButton`** depois de ser redimensionado.

Fig. 25.9 A janela **BeanBox** com dois *beans* **`ExplicitButton`**.

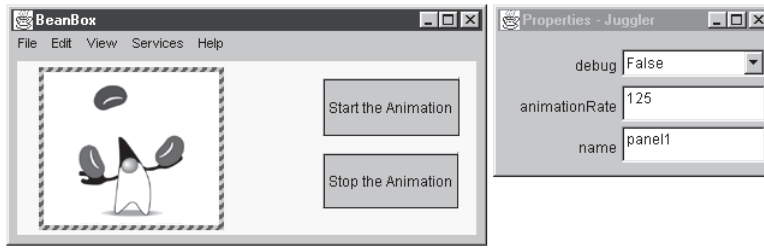


Fig. 25.10 A janela **BeanBox** com o novo *bean* **`Juggler`**.

Observação de engenharia de software 25.3

Um benefício de trabalhar em um ambiente de desenvolvimento de beans prontos é que os beans em geral executam no ambiente de desenvolvimento. Isso permite visualizar e testar a funcionalidade de seu programa imediatamente no ambiente de projeto em vez de utilizar o ciclo de programação padrão de editar, compilar e executar.

Além disso, repare que a janela **Properties** agora exibe as propriedades do **`Juggler`**. Uma das propriedades que podemos configurar para o **`Juggler`** é sua **`animationRate`**. A **`animationRate`** é na realidade o intervalo de adormecimento para o *thread* que exibe as imagens na animação (o tempo-padrão de adormecimento é 125 milissegundos). Diminuir o **`animationRate`** aumenta a velocidade de animação. Aumentar a **`animationRate`** diminui a velocidade de animação. Tente alterar a propriedade **`animationRate`**. Lembre-se de pressionar a tecla *Enter* depois de alterar o valor no campo de texto para configurar a nova **`animationRate`**.

A seguir, "engancharemos" os eventos dos botões na animação para iniciar e parar a animação. Como a animação já está executando, iniciamos com o botão de parada.

O *menu* ***Edit*** na janela **BeanBox** fornece acesso aos eventos suportados por um *bean* que é uma *origem de evento* (isto é, qualquer *bean* que pode notificar um *ouvinte* de que um evento ocorreu utilizando o tratamento de evento-padrão modelo mostrado por todo este livro). De fato, componentes GUI Swing são todos *beans*. Selecione o botão "**`Stop the Animation`**", então clique no menu **Edit** e posicione o mouse sobre o *item de menu* ***Events***. Um submenu aparece (Fig. 25.11) contendo itens de menu para cada tipo de evento suportado pelo *bean* selecionado atualmente. Os dois eventos suportados são ***button push*** — quando um usuário pressiona o botão — e ***bound property change*** — notificando um ouvinte quando o valor de uma propriedade muda (discutiremos

propriedades associadas em detalhes na Seção 25.8). Posicione o mouse sobre o item de menu **button push**. Aparece um submenu contendo o método **actionPerformed**. Como você viu em tratamento de eventos GUI, **actionPerformed** é o método invocado em um objeto **ActionListener** quando ocorre um **ActionEvent** (como um pressionamento de botão). Os nomes de item de menu exibidos para cada tipo de evento no item de menu **Events** do menu **Edit** podem ser personalizados para cada *bean* (mostramos como fazer isso na Seção 25.9).

Clique no item de menu **actionPerformed** para indicar que você gostaria de especificar o que acontece quando o botão é pressionado (isto é, parar a animação). À medida que você move o mouse pela janela **BeanBox** depois de selecionar o evento, você verá uma linha vermelha seguindo o mouse. Esta *linha seletora do alvo* (Fig. 25.12) ajuda especificar o *alvo do evento* — o objeto sobre o qual pretendemos chamar um método quando o botão for pressionado. Gostamos de chamar isso de *programação de ligar os pontos* — os pontos são os *beans* e a linha seletora do alvo ajuda a ligar os pontos.

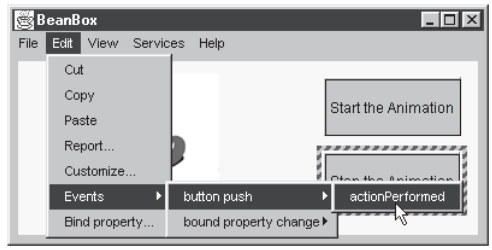


Fig. 25.11 Selecionando o evento de pressionar botão para um **ExplicitButton**.

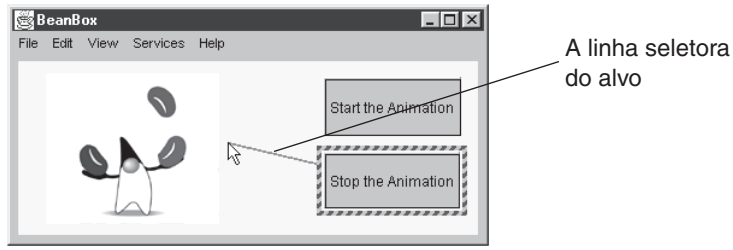


Fig. 25.12 A linha seletora do alvo.

Posicione o mouse sobre o **Juggler** e clique para especificar o **Juggler** como o alvo do evento. Isso exibe a *janela* ***EventTargetDialog*** listando os métodos **public** que podem ser chamados no alvo (Fig. 25.13). Dessa lista, você pode selecionar o método do alvo que será chamado quando o usuário clicar no botão **Stop the Animation**.

O **Juggler** fornece para nós dois métodos importantes nesse exemplo — o método ***startJuggling*** inicia a animação e o método ***stopJuggling*** pára a animação. Selecione **stopJuggling** da lista, depois pressione o botão **OK** para completar o *enganchamento de evento (event hookup)*. O **EventTargetDialog** exibe a mensagem "Generating and compiling adapter class" ("Gerando e compilando classe adaptadora") para indicar que a **BeanBox** está escrevendo uma nova definição de classe (chamada de *classe de enganchamento* ou *classe adaptadora de evento*). Um objeto dessa nova classe é automaticamente criado e registrado como o **ActionListener** para o botão **Stop the Animation**. Neste ponto, você pode clicar no botão **Stop the Animation** para interromper a animação. Quando o evento ocorre, o método **actionPerformed** da classe de enganchamento chama o método **stopJuggling** no alvo **Juggler**. A Fig. 25.14 ilustra a interação.

Repita o processo de enganchar um evento de pressionamento de botão ao botão **Start the Animation**. Na janela **EventTargetDialog**, selecione o método **startJuggling** como o método-alvo para chamar quando o botão é pressionado. Neste ponto, você pode clicar em **Start the Animation** para iniciar a animação.

Fig. 25.13 A janela **EventTargetDialog** exibe a lista de métodos que podem ser chamados no alvo de um evento.

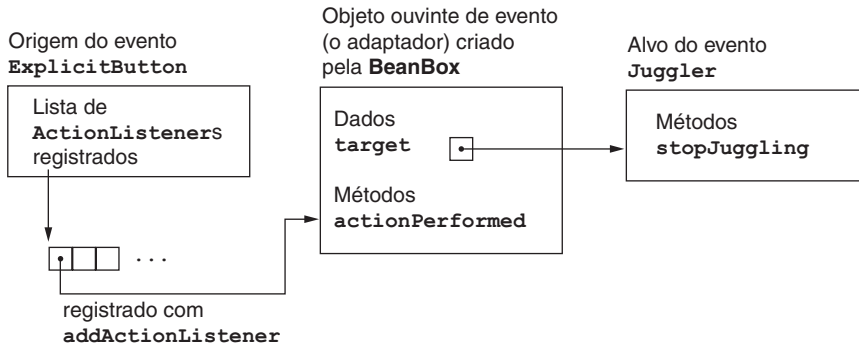


Fig. 25.14 A interação entre o `ExplicitButton` e o `Juggler`.

A **BeanBox** fornece várias maneiras de salvar seu projeto, incluindo salvar o projeto que será recarregado na **BeanBox** mais tarde (o *item de menu* ***Save...*** do menu **File**) e salvar o projeto como um *applet* Java (o *item de menu* ***MakeApplet...*** do menu **File**). Há uma terceira opção — ***SerializeComponent...*** — que pode ser utilizada para ajudar a criar um novo JavaBean. Para mais informações sobre essa opção, consulte a documentação on-line para a **BeanBox** (localizada no diretório `doc` onde você instalou o BDK).

Para salvar o projeto de modo que ele possa ser recarregado na **BeanBox** mais tarde, selecione o item de menu **Save...** do menu **File**. Isso exibe a *caixa de diálogo* ***Save BeanBox File*** na Fig. 25.15. Essa opção utiliza serialização de objetos para salvar os *beans* no projeto. Atribuímos ao nosso arquivo o nome `OurJuggler.ser` (estamos utilizando a *extensão* `.ser` para indicar que o projeto é um arquivo serializado). Não há nenhum requisito de atribuição de nome de arquivo para essa opção, então você pode nomear o arquivo da maneira como quiser. Por default seu arquivo será salvo no diretório `beanbox` na estrutura de diretórios de BDK. Se você quiser salvar o arquivo em outra localização, utilize o diálogo para alterar o diretório antes de clicar no botão **Save** para salvar o projeto.

Para ver se seu arquivo foi salvo corretamente, selecione o *item de menu* ***Clear*** do menu **File** para esvaziar a **BeanBox**. Em seguida, selecione o *item de menu* ***Load...*** do menu **File** para exibir o *diálogo* ***Load saved BeanBox*** (Fig. 25.16). Selecione o arquivo que você salvou previamente (`OurJuggler.ser`), então clique no botão **Open** para recarregar o projeto. Observe que a animação inicia automaticamente quando o projeto é recarregado. Isso porque o *bean* `Juggler` é realmente um *applet*. Quando um *bean applet* é recarregado, ele começa a executar com seu método `start` (exatamente como um *applet* faz quando é recarregado em um navegador da Web). O método `start` de `Juggler` cria um novo *thread* para reiniciar a animação.

Para salvar seu projeto como um *applet*, selecione o *item de menu* ***MakeApplet...*** do menu **File**. Isso exibe o *diálogo* ***Make an Applet*** da Fig. 25.17. O nome-padrão da classe de *applet* é `MyApplet`, mas você pode personalizar o nome clicando no *botão* ***Choose JAR File...*** para exibir o *diálogo* ***Choose JAR File*** na Fig. 25.18. Quando a **BeanBox** cria um *applet* de seu projeto, ela armazena os arquivos `.class` em um *Java Archive File*

*(*arquivo *JAR).* Os JavaBeans são armazenados em arquivos JAR (discutiremos arquivos JAR mais adiante neste capítulo). Quando você escolhe um nome diferente para o arquivo JAR, o nome da classe de *applet* é alterado. Atribua ao arquivo JAR o nome **`OurJuggler.jar`** digitando o nome no campo **File name.**

Além disso, você pode alterar o diretório em que o arquivo JAR será armazenado nesse diálogo. Depois de escolher o nome do arquivo e do diretório, clique em **Save** para retornar para o diálogo **Make an Applet** na Fig. 25.19. Por default, nosso *applet* será armazenado no diretório.

```
BDK\beanbox\tmp\myApplet
```

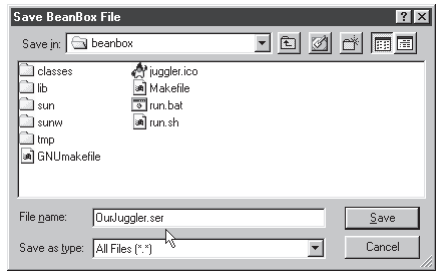


Fig. 25.15 O diálogo **Save BeanBox File**.

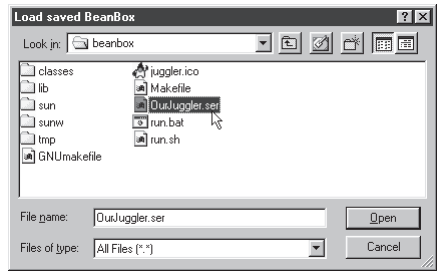


Fig. 25.16 O diálogo **Load saved BeanBox**.

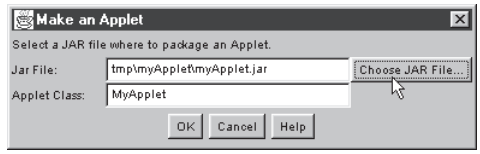


Fig. 25.17 O diálogo **Make an Applet**.

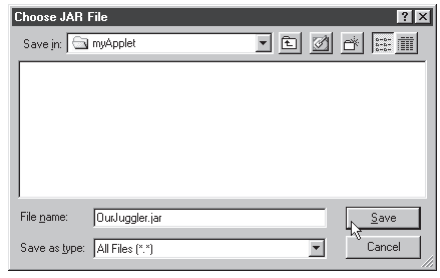


Fig. 25.18 O diálogo **Choose JAR File**.

Fig. 25.19 O diálogo **Make an Applet** depois de se alterar o nome de arquivo padrão.

Para executar o *applet* no **appletviewer**, vá para a linha de comando e mude para o diretório onde você salvou o *applet*. Se você denominou o arquivo como especificado previamente, haverá um arquivo chamado **OurJuggler.html** que pode ser carregado no **appletviewer**. O *applet* carregado é mostrado na Fig. 25.20. Observe que o *applet* não tem o fundo amarelo da **BeanBox**. Isso porque o contêiner da **BeanBox** à qual anexamos os componentes não é salvo como parte do *applet*. O *applet* já é um contêiner e é capaz de armazenar os *beans* que são parte do *applet*.

Inspecione o código-fonte para o arquivo de HTML do *applet* gerado pela **BeanBox** (Fig. 25.21). Repare nas linhas 12 a 15

```
archive="./OurJuggler.jar,./support.jar
   ,./juggler.jar
   ,./buttons.jar
"
```

Essa é a *propriedade* **archive** do *tag* **<applet>**. Ela especifica uma lista separada por vírgulas dos arquivos JAR contendo o código que é utilizado para executar esse *applet*. Cada arquivo JAR para um *bean* que utilizamos é listado, além de **OurJuggler.jar**, que contém o código para classe **OurJuggler**. Repare também na linha 16

```
code="OurJuggler"
```

que especifica o nome da classe de *applet* que irá iniciar a execução de nosso *applet*.

Se você quiser visualizar o código-fonte para o *applet* que foi escrito pela **BeanBox**, o diretório **OurJuggler_files** onde o *applet* foi salvo contém todo o código-fonte gerado pela **BeanBox**.

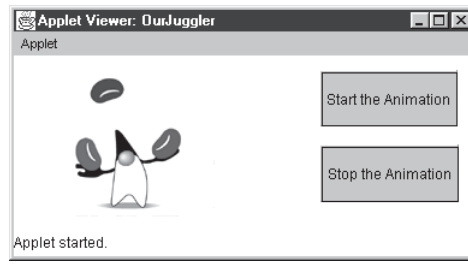


Fig. 25.20 O *applets* **OurJuggler** executando no **appletviewer**.

```
1  <html>
2  <head>
3  <title>Test page for OurJuggler as an APPLET</Title>
4  </head>
5  <body>
6  <h1>Test for OurJuggler as an APPLET</h1>
7  This is an example of the use of the generated
8  OurJuggler applet.  Notice the Applet tag requires several
9  archives, one per JAR used in building the Applet
10 <p>
```

Fig. 25.21 O arquivo **OurJuggler.html** gerado pela **BeanBox** (parte 1 de 2).

```
<applet
    archive="./OurJuggler.jar,./support.jar
        ,./juggler.jar
        ,./buttons.jar
    "
    code="OurJuggler"
    width=382
    height=150
>
Trouble instantiating applet OurJuggler!!
</applet>
```

Fig. 25.21 O arquivo `OurJuggler.html` gerado pela **BeanBox** (parte 2 de 2).

25.3 Preparando uma classe para ser um JavaBean

O próximo exemplo e as seções a seguir apresentam uma animação de um logotipo da Deitel & Associates, Inc. (`LogoAnimator`) como visto no Capítulo 16. Anteriormente, a animação foi demonstrada como um aplicativo independente que executa em um `JFrame`. Nesta seção, demonstramos o `LogoAnimator` como um aplicativo independente e como um *bean* para utilizar na **BeanBox**. O código de programa na Fig. 25.22 e as quatro capturas de tela demonstram o *bean* executando como um aplicativo independente. Nossa classe `LogoAnimator` é uma subclasse de `JPanel`, portanto ela tem propriedades e eventos herdados da classe `JPanel`. Quando um *bean* de `LogoAnimator` é carregado na **BeanBox**, tiramos proveito de algumas dessas propriedades e eventos predefinidos, como a cor de fundo e eventos de mouse. Esta seção também demonstra interações entre *beans* de demonstração da **BeanBox** e nosso *bean* `LogoAnimator`.

```
// Fig. 25.22: LogoAnimator.java
// Bean de animação
package jhtp3beans;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class LogoAnimator extends JPanel
       implements ActionListener, Serializable {
   protected ImageIcon images[];
   protected int totalImages = 30,
                 currentImage = 0,
                 animationDelay = 50; // retardo de 50 milissegundos
   protected Timer animationTimer;

   public LogoAnimator()
   {
      setSize( getPreferredSize() );

      images = new ImageIcon[ totalImages ];

      URL url;

      for ( int i = 0; i < images.length; ++i ) {
         url = getClass().getResource(
                  "deitel" + i + ".gif" );
         images[ i ] = new ImageIcon( url );
```

Fig. 25.22 O `LogoAnimator` como um aplicativo independente (parte 1 de 3).

```
31          }
32
33          startAnimation();
34       }
35
36       public void paintComponent( Graphics g )
37       {
38          super.paintComponent( g );
39
40          if ( images[ currentImage ].getImageLoadStatus() ==
41               MediaTracker.COMPLETE ) {
42             g.setColor( getBackground() );
43             g.drawRect(
44                0, 0, getSize().width, getSize().height );
45             images[ currentImage ].paintIcon( this, g, 0, 0 );
46             currentImage = ( currentImage + 1 ) % totalImages;
47          }
48       }
49
50       public void actionPerformed( ActionEvent e )
51       {
52          repaint();
53       }
54
55       public void startAnimation()
56       {
57          if ( animationTimer == null ) {
58             currentImage = 0;
59             animationTimer = new Timer( animationDelay, this );
60             animationTimer.start();
61          }
62          else  // continua a partir da última imagem exibida
63             if ( ! animationTimer.isRunning() )
64                animationTimer.restart();
65       }
66
67       public void stopAnimation()
68       {
69          animationTimer.stop();
70       }
71
72       public Dimension getMinimumSize()
73       {
74          return getPreferredSize();
75       }
76
77       public Dimension getPreferredSize()
78       {
79          return new Dimension( 160, 80 );
80       }
81
82       public static void main( String args[] )
83       {
84          LogoAnimator anim = new LogoAnimator();
85
86          JFrame app = new JFrame( "Animator test" );
87          app.getContentPane().add( anim, BorderLayout.CENTER );
```

Fig. 25.22 O `LogoAnimator` como um aplicativo independente (parte 2 de 3).

```
88
89          app.addWindowListener(
90             new WindowAdapter() {
91                public void windowClosing( WindowEvent e )
92                {
93                   System.exit( 0 );
94                }
95             }
96          );
97
98          app.setSize( anim.getPreferredSize().width + 10,
99                       anim.getPreferredSize().height + 30 );
100         app.show();
101      }
102   }
```

Fig. 25.22 O **LogoAnimator** como um aplicativo independente (parte 3 de 3).

Há duas modificações menores nesse programa que nos permitirão utilizar a classe como um JavaBean. Primeiro, repare que adicionamos uma instrução **package** (linha 3) ao arquivo da classe **LogoAnimator**. Normalmente, as classes que representam um *bean* são as primeiras colocadas em um pacote. Lembre-se de que você deve compilar suas classes empacotadas utilizando a opção **-d** com o compilador Java, como em

```
javac -d . LogoAnimator.java
```

Como mostrado acima, o "**.**" representa o diretório em que o pacote **jhtp3beans** deve ser colocado ("**.**" representa o diretório atual, que utilizamos aqui por questão de simplicidade). Depois de a classe ser compilada, você pode empacotar a classe em um JavaBean armazenado como um *Java Archive File (arquivo JAR)* que termina com a *extensão* ***.jar***. Um único arquivo JAR pode conter muitos JavaBeans. Os arquivos JAR são criados com o *utilitário* ***jar*** que vem com o JDK. Discutiremos a criação do arquivo JAR na próxima seção.

A segunda modificação está na linha 12, onde a classe especifica que ela implementa a interface **Serializable** para suportar a *persistência* — salvar um objeto *bean* em seu estado atual para futura utilização. Os objetos de nossa classe **LogoAnimator** podem ser serializados com **ObjectOutputStream**s e **ObjectInputStream**s (mostrados no Capítulo 17, "Arquivos e fluxos)". Implementar a interface **Serializable** permite que programadores utilizem uma ferramenta de desenvolvimento para salvar seu *bean* personalizado *serializando o bean* para um arquivo.

Observação de engenharia de software 25.4

Todos os JavaBeans devem implementar a interface ***Serializable*** *para suportar persistência.*

Exceto por essas modificações menores no **LogoAnimator**, mostrado aqui para tornar a animação serializável, o código é o mesmo que o da animação mostrada no Capítulo 16. A próxima seção discute a criação de um JavaBean do **LogoAnimator**. A seção a seguir discute a utilização do JavaBean na **BeanBox**.

25.4 Criando um JavaBean: Java Archive Files e o utilitário `jar`

Para utilizar uma classe como um JavaBean, ela deve primeiro ser colocada em um *Java Archive File* (arquivo JAR). Antes de criar o arquivo JAR, primeiro criamos um arquivo de texto chamado ***manifest.tmp***. O *arquivo manifesto* (como é chamado) é utilizado pelo utilitário **jar** para descrever o conteúdo do arquivo JAR. Isso é importante para ambientes integrados de desenvolvimento que suportam JavaBeans. Quando um arquivo JAR con-

tendo um JavaBean (ou um conjunto de JavaBeans) é carregado em um IDE, o IDE examina o *arquivo manifesto* para determinar as classes no JAR que representam JavaBeans. Essas classes são disponibilizadas para o programador de uma maneira visual, como você viu anteriormente na visão geral de **BeanBox** neste capítulo. O utilitário de repositórios de arquivo Java **`jar`** utiliza **`manifest.tmp`** para criar um arquivo chamado ***`MANIFEST.MF`*** que é incluído no *diretório* ***`META-INF`*** do arquivo JAR. Todos os ambientes de desenvolvimento sabem procurar o arquivo **`MANIFEST.MF`** no diretório **`META-INF`** do arquivo JAR. Além disso, o interpretador Java pode executar um aplicativo diretamente a partir de um arquivo JAR se o arquivo manifesto indicar qual classe no JAR contém **`main`**. O arquivo **`manifest.tmp`** para o **`LogoAnimator`** é mostrado na Fig. 25.23.

Observação de engenharia de software 25.5

Você deve definir um arquivo manifesto que descreve o conteúdo de um arquivo JAR se pretende utilizar o bean em um ambiente de desenvolvimento integrado compatível com JavaBean ou se pretende executar um aplicativo diretamente a partir de um arquivo JAR.

```
1  Main-Class: jhtp3beans.LogoAnimator
2
3  Name: jhtp3beans/LogoAnimator.class
4  Java-Bean: True
```

Fig. 25.23 O arquivo **`manifest.tmp`** para o bean **`LogoAnimator`**.

Nesse arquivo manifesto particular, incluímos a linha 1, que especifica que a classe **`jhtp3beans.LogoAnimator`** é a classe que contém o método **`main`** para executar o *bean* como um aplicativo. Quando a classe **`main`** de um aplicativo é armazenada em um JavaBean, o aplicativo pode ser executado diretamente do *bean* utilizando o interpretador Java com a opção de linha de comando **`-jar`** como segue:

```
java -jar LogoAnimator.jar
```

O interpretador automaticamente verá o arquivo manifesto para a classe especificada com a *propriedade de arquivo manifesto* ***`Main-Class:`*** e iniciará a execução com o método **`main`** dessa classe. O aplicativo também pode ser executado a partir do arquivo JAR que não contém um manifesto com o comando

```
java -cp LogoAnimator.jar jhtp3beans.LogoAnimator
```

em que **`-cp`** indica o *caminho da classe* (isto é, o arquivo JAR em que o interpretador deve procurar as classes). A opção **`-cp`** é seguida pelo arquivo JAR contendo a classe de aplicativo. O último argumento da linha de comando é o nome explícito da classe (incluindo o nome do pacote) para a classe do aplicativo.

Com Java 2, muitas plataformas constroem automaticamente o comando precedente quando o usuário executa o aplicativo Java como fariam com qualquer outro aplicativo nessa plataforma. Por exemplo, no Microsoft Windows, o usuário pode executar um aplicativo Java a partir de um arquivo JAR dando clique duplo no nome de arquivo JAR no Windows Explorer.

A linha 3 do arquivo manifesto especifica o ***`Name:`*** do arquivo contendo a classe do *bean* (incluindo a extensão de nome de arquivo **`.class`**) utilizando seu nome de pacote e de classe. Observe que os pontos (**`.`**) comumente utilizados em um nome de pacote são substituídos por barras normais (**`/`**) para o **Name:** no arquivo manifesto. A linha 4 especifica que a classe nomeada na linha 3 é, de fato, um JavaBean (***`Java-Bean: True`***). É possível ter em um arquivo JAR classes que não são JavaBeans. Essas classes são geralmente utilizadas para suportar os JavaBeans no repositório de arquivos. Por exemplo, um *bean* de lista encadeada talvez tenha uma classe de nodo de lista vinculada de suporte cujos objetos são utilizados para representar cada nodo na lista. Cada classe listada no arquivo manifesto deve ser separada de todas as outras classes por uma linha em branco. Se a classe é um *bean*, sua linha **Name:** deve ser imediatamente seguida pela sua linha **Java-Bean:**.

Observação de engenharia de software 25.6

No arquivo manifesto, o nome de um bean é especificado com a propriedade ***`Name:`*** *seguida pelo nome do pacote completo e o nome de classe do bean. Os pontos (`.`) normalmente utilizados para separar os nomes de pacote dos nomes de classe são substituídos por barras normais (`/`) nessa linha do arquivo manifesto.*

Observação de engenharia de software 25.7

Se uma classe específica representa um bean, a linha que se segue à propriedade **`Name:`** *dessa classe deve especificar* **`Java-Bean: True`**. *Caso contrário, os IDEs não reconhecerão a classe como um bean.*

Observação de engenharia de software 25.8

Se uma classe contendo **`Main`** *é incluída em um arquivo JAR, essa classe pode ser utilizada pelo interpretador para executar o aplicativo diretamente do arquivo JAR especificando a propriedade* **`Main-Class:`** *em uma linha isolada no início do arquivo manifesto. O nome completo do pacote e o nome completo da classe devem ser especificados com o ponto normal (* **`.`** *) separando os nomes de pacote e o nome da classe.*

Erro comum de programação 25.1

Não especificar um arquivo manifesto ou especificar um arquivo manifesto com sintaxe incorreta ao criar um arquivo JAR é um erro. As ferramentas de desenvolvimento não reconhecerão os beans no arquivo JAR.

A seguir, criamos o arquivo JAR para o *bean* de **`LogoAnimator`**. Isso é realizado com o utilitário **`jar`** na linha de comando (como o Prompt do MS-DOS ou shell de UNIX). O comando

```
jar cfm LogoAnimator.jar manifest.tmp jhtp3beans\*.*
```

cria o arquivo JAR. [*Nota:* esse comando utiliza **`\`** como o separador de diretório no prompt do MS-DOS. O UNIX utilizaria **`/`** como o separador de diretório.] No comando precedente, **`jar`** é o utilitário de repositório de arquivos Java utilizado para criar arquivos JAR. Em seguida, estão as opções para o utilitário ***`jar`***, ***`cfm`***. A letra ***`c`*** indica que estamos criando um arquivo JAR. A letra ***`f`*** indica que o próximo argumento na linha de comando (**`LogoAnimator.jar`**) é o nome do arquivo JAR a ser criado. A letra ***`m`*** indica que o próximo argumento na linha de comando é o arquivo **`manifest.tmp`**, que é utilizado pelo **`jar`** para criar o arquivo **`MANIFEST.MF`** no diretório **`META-INF`** do JAR. Seguindo as opções, o nome do arquivo JAR e o arquivo **`manifest.tmp`** são os arquivos reais que serão incluídos no arquivo JAR. Especificamos **`jhtp3beans\*.*`**, para indicar que todos os arquivos no diretório **`jhtp3beans`** devem ser incluídos no arquivo JAR. O diretório de pacote **`jhtp3beans`** contém os arquivos **`.class`** para o **`LogoAnimator`** e suas classes de suporte, bem como as imagens utilizadas na animação. [*Nota:* você pode incluir arquivos selecionados especificando o caminho e o nome de arquivo para cada arquivo individual.] É importante que a estrutura de diretórios no arquivo JAR corresponda à estrutura de diretórios utilizada no arquivo **`manifest.tmp`**. Portanto, executamos o comando **`jar`** a partir do diretório em que **`jhtp3beans`** está localizado.

Para confirmar se os arquivos foram arquivados diretamente, você pode utilizar o comando

```
jar tvf LogoAnimator.jar
```

que produz a listagem na Fig. 25.24. No comando precedente, as opções para o utilitário **`jar`** são ***`tvf`***. A letra ***`t`*** indica que o conteúdo (*table of contents*) do JAR deve ser listado. A letra ***`v`*** indica que a saída deve ser verbosa (a saída verbosa inclui o tamanho do arquivo em bytes e a data e hora que cada arquivo foi criado, além da estrutura de diretórios e nome de arquivo). A letra ***`f`*** especifica que o próximo argumento na linha de comando é o arquivo JAR a utilizar.

Tente executar o aplicativo **`LogoAnimator`** com o comando

```
java -jar LogoAnimator.jar
```

Você verá que a animação aparece em sua própria janela na tela.

```
     0 Sun Mar 14 11:36:16 EST 1999 META-INF/
   163 Sun Mar 14 11:36:16 EST 1999 META-INF/MANIFEST.MF
  4727 Thu Feb 15 00:37:04 EST 1996 jhtp3beans/deitel0.gif
  4858 Thu Feb 15 00:39:32 EST 1996 jhtp3beans/deitel1.gif
  4374 Thu Feb 15 00:55:46 EST 1996 jhtp3beans/deitel10.gif
  4634 Thu Feb 15 00:56:52 EST 1996 jhtp3beans/deitel11.gif
  4852 Thu Feb 15 00:58:00 EST 1996 jhtp3beans/deitel12.gif
  4877 Thu Feb 15 00:59:10 EST 1996 jhtp3beans/deitel13.gif
  4926 Thu Feb 15 01:00:20 EST 1996 jhtp3beans/deitel14.gif
  4765 Thu Feb 15 01:01:32 EST 1996 jhtp3beans/deitel15.gif
  4886 Thu Feb 15 01:05:16 EST 1996 jhtp3beans/deitel16.gif
  4873 Thu Feb 15 01:06:12 EST 1996 jhtp3beans/deitel17.gif
  4739 Thu Feb 15 01:07:18 EST 1996 jhtp3beans/deitel18.gif
  4566 Thu Feb 15 01:08:24 EST 1996 jhtp3beans/deitel19.gif
  4819 Thu Feb 15 00:41:06 EST 1996 jhtp3beans/deitel2.gif
  4313 Thu Feb 15 01:09:48 EST 1996 jhtp3beans/deitel20.gif
  3910 Thu Feb 15 01:10:46 EST 1996 jhtp3beans/deitel21.gif
  3076 Thu Feb 15 01:12:02 EST 1996 jhtp3beans/deitel22.gif
  3408 Thu Feb 15 01:13:16 EST 1996 jhtp3beans/deitel23.gif
  4039 Thu Feb 15 01:14:06 EST 1996 jhtp3beans/deitel24.gif
  4393 Thu Feb 15 01:15:02 EST 1996 jhtp3beans/deitel25.gif
  4626 Thu Feb 15 01:16:06 EST 1996 jhtp3beans/deitel26.gif
  4852 Thu Feb 15 01:17:18 EST 1996 jhtp3beans/deitel27.gif
  4929 Thu Feb 15 01:18:18 EST 1996 jhtp3beans/deitel28.gif
  4914 Thu Feb 15 01:19:16 EST 1996 jhtp3beans/deitel29.gif
  4769 Thu Feb 15 00:42:52 EST 1996 jhtp3beans/deitel3.gif
  4617 Thu Feb 15 00:43:54 EST 1996 jhtp3beans/deitel4.gif
  4335 Thu Feb 15 00:47:14 EST 1996 jhtp3beans/deitel5.gif
  3967 Thu Feb 15 00:49:40 EST 1996 jhtp3beans/deitel6.gif
  3200 Thu Feb 15 00:50:58 EST 1996 jhtp3beans/deitel7.gif
  3393 Thu Feb 15 00:52:32 EST 1996 jhtp3beans/deitel8.gif
  4006 Thu Feb 15 00:53:48 EST 1996 jhtp3beans/deitel9.gif
   420 Sun Mar 14 11:36:16 EST 1999 jhtp3beans/LogoAnimator$1.class
  3338 Sun Mar 14 11:36:16 EST 1999 jhtp3beans/LogoAnimator.class
```

Fig. 25.24 O conteúdo de **`LogoAnimator.jar`**.

25.5 Adicionando *beans* à BeanBox

Agora que o **`LogoAnimator`** está empacotado em um arquivo JAR como um JavaBean, podemos utilizar o novo *bean* na **BeanBox**. Há duas maneiras de carregar o *bean* na **BeanBox**— coloque o arquivo JAR no diretório **`BDK1.1\jars`** ou utilize a opção **LoadJar...** do menu **File** da **BeanBox** para localizar o arquivo JAR em seu sistema e carregá-lo na **ToolBox** da **BeanBox**. Se você colocar o arquivo JAR no diretório **`BDK1.1\jars`**, ele será automaticamente carregado na **ToolBox** quando a **BeanBox** for executada. A Fig. 25.25 mostra a opção **LoadJar...** do menu **File**, o diálogo **Load beans from JAR File** e a **ToolBox** com **`LogoAnimator`** carregado.

Para adicionar um **`LogoAnimator`** à área de projeto da **BeanBox**, clique no *bean* de **`LogoAnimator`** na **ToolBox** para mover o cursor em forma de cruz sobre a área de projeto da **BeanBox** e clique no lugar em que você quer que o centro do **`LogoAnimator`** apareça. Assim que você clica, a **BeanBox** cria um objeto do tipo **`LogoAnimator`** que imediatamente começa a carregar e exibir as imagens da animação. Observe que a janela **Properties** agora contém as propriedades do **`LogoAnimator`**. A Fig. 25.26 mostra a área de projeto de **BeanBox** com o **`LogoAnimator`** e a janela **Properties** com as propriedades de **`LogoAnimator`**.

Observe que a janela **Properties** mostra várias propriedades **`LogoAnimator`**. Essas propriedades foram todas herdadas da classe **`JPanel`**. No exemplo da Seção 25.7, adicionaremos nossa própria propriedade para controlar a velocidade da animação. Nosso *bean* de **`LogoAnimator`** já pode ser configurado, mesmo sem termos criado ainda nossas próprias propriedades. Uma vez que utilizamos imagens GIF transparentes na animação, você

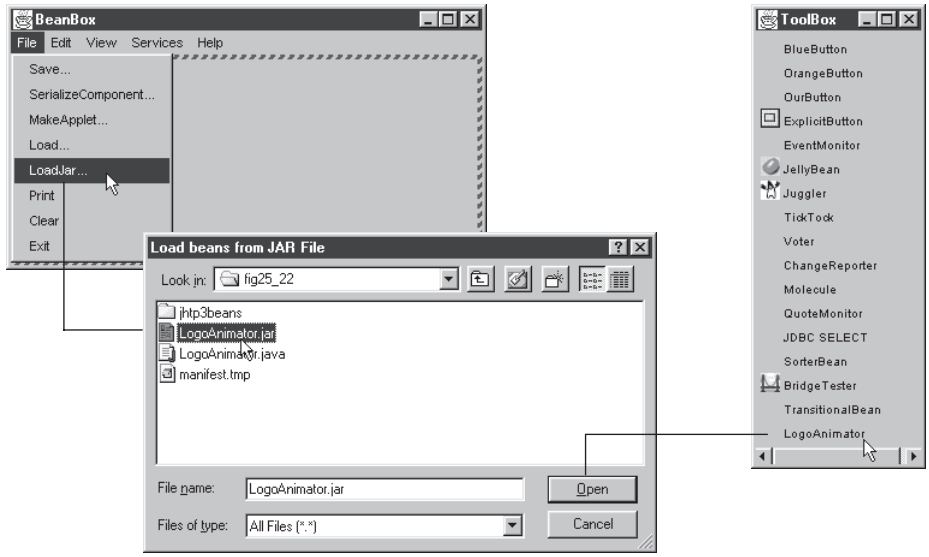


Fig. 25.25 Carregando um *bean* na **Toolbox** da **BeanBox**.

pode escolher a cor do **background** para a animação. Simplesmente clique no retângulo à direita da propriedade do **background** na janela **Properties**. Isso exibe o editor de propriedades **ColorEditor** da **BeanBox**. Aqui você pode escolher uma nova cor de fundo para a animação. A Fig. 25.27 mostra o editor de propriedades **ColorEditor** e o **`LogoAnimator`** na área de projeto da **BeanBox** com sua nova cor de fundo.

25.6 Associando *beans* com eventos na BeanBox

Lembre-se de que nosso *bean* de **`LogoAnimator`** fornece os métodos **`stopAnimation`** e **`startAnimation`** que permitem que a animação seja parada e reiniciada. Agora conectaremos dois botões com o **`LogoAnimator`** — **Stop** e **Start**. Como vimos anteriormente, a **BeanBox** vem com vários botões. Anexaremos dois **`ExplicitButton`**s ao **`LogoAnimator`**.

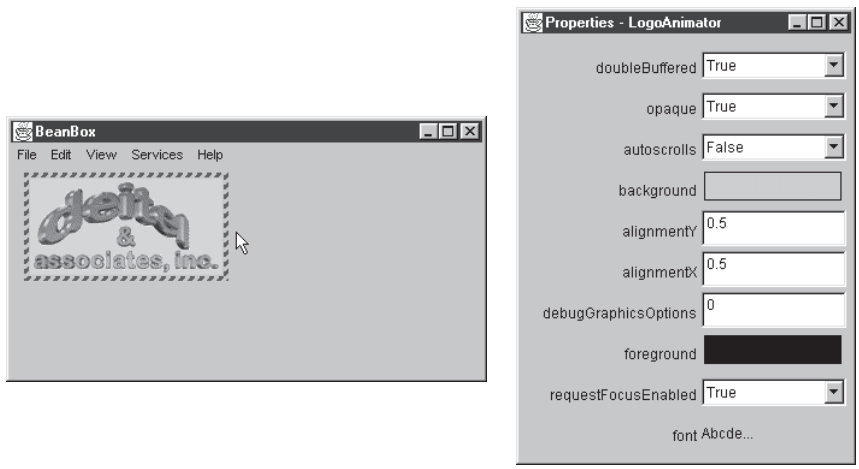


Fig. 25.26 O **`LogoAnimator`** na área de projeto da **BeanBox**.

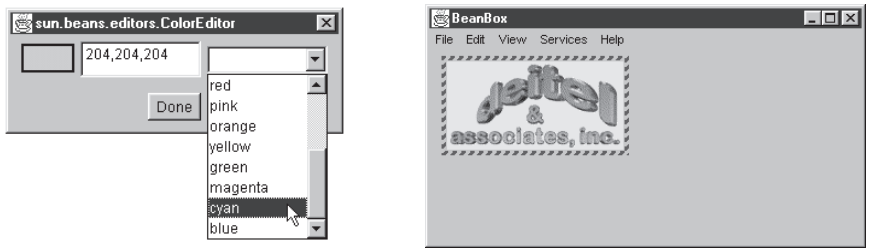


Fig. 25.27 Alterando a cor de fundo do **`LogoAnimator`**.

Coloque dois **`ExplicitButton`**s na área de projeto **BeanBox**. Altere o rótulo em um **`ExplicitButton`** para **`Start`** e o outro para **`Stop`**. A Fig. 25.28 mostra a **BeanBox** com o **`LogoAnimator`** e dois **`ExplicitButton`**s.

Fig. 25.28 Dois **`ExplicitButton`**s para iniciar e interromper a animação.

A seguir, selecione o botão **`Stop`**. No menu **Edit** da **BeanBox**, selecione o evento **button-push** como mostra a Fig. 25.29. Posicione o ponteiro do mouse sobre o **`LogoAnimator`** e clique para exibir a janela **EventTargetDialog**. Em seguida, selecionamos o método no alvo (isto é, o **`LogoAnimator`**) que será chamado quando o usuário clicar no botão **`Stop`**. Role para baixo até **`stopAnimation`**, selecione essa opção e então pressione o botão **OK** para completar a enganchamento do evento. Repita o processo para o botão **Start** e selecione o método **`startAnimation`** como o método alvo a chamar quando o botão é pressionado. Depois de enganchar os dois eventos de botão ao **`LogoAnimator`**, clique no botão **Stop** para ver a animação parar, depois clique no botão **Start** para reiniciar a animação.

25.7 Adicionando propriedades a um JavaBean

Nesta seção, demonstraremos a adição de uma propriedade **`animationDelay`** em nosso **`LogoAnimator`** para controlar a velocidade da animação. Para esse propósito, estendemos a classe **`LogoAnimator`** para criar a classe **`LogoAnimator2`**. O novo código para nossa propriedade é definido pelo método **`setAnimationDelay`** (linha 12) e pelo método **`getAnimationDelay`** (linha 18) na Fig. 25.30.

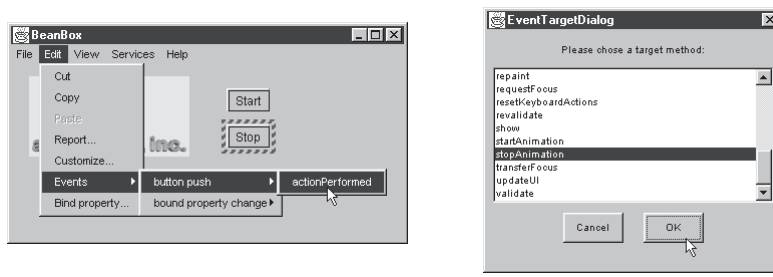


Fig. 25.29 Configurando o evento de pressionar botão para o botão **Stop**.

```
1  // Fig. 25.30: LogoAnimator2.java
2  // Bean de animação com a propriedade animationDelay
3  package jhtp3beans;
4
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8
9  public class LogoAnimator2 extends LogoAnimator {
10    // os dois métodos seguintes são
11    // para a propriedade animationDelay
12    public void setAnimationDelay( int value )
13    {
14       animationDelay = value;
15       animationTimer.setDelay( animationDelay );
16    }
17
18    public int getAnimationDelay()
19    {
20       return animationTimer.getDelay();
21    }
22
23    public static void main( String args[] )
24    {
25       LogoAnimator2 anim = new LogoAnimator2();
26
27       JFrame app = new JFrame( "Animator test" );
28       app.getContentPane().add( anim, BorderLayout.CENTER );
29
30       app.addWindowListener(
31          new WindowAdapter() {
32             public void windowClosing( WindowEvent e )
33             {
34                System.exit( 0 );
35             }
36          }
37       );
38
39       app.setSize( anim.getPreferredSize().width + 10,
40                    anim.getPreferredSize().height + 30 );
41       app.show();
42    }
43 }
```

Fig. 25.30 `LogoAnimator2` com a propriedade `animationDelay`.

Para criar a propriedade de **`animationDelay`**, definimos os métodos **`setAnimationDelay`** e **`getAnimationDelay`**. Uma propriedade de *leimtura/gravação* de um *bean* é definida como um par de métodos *set/get* da seguinte forma:

```
public void set NomeDaPropriedade ( TipoDeDados value )
public TipoDeDados get NomeDaPropriedade ()
```

Esses métodos são freqüentemente referidos como "método *set* de propriedade" e "método *get* de propriedade", respectivamente.

Observação de engenharia de software 25.9

Uma propriedade de leitura/gravação do JavaBean é definida por um par de métodos set/get *em que o método* set *retorna* **`void`** *e aceita um argumento e o método* get *retorna o mesmo tipo do argumento correspondente do método*

set *e não aceita nenhum argumento. Também é possível ter propriedades somente de leitura (definidas apenas com um método* get*) e propriedades somente de escrita (definidas apenas com um método* set*).*

Observação de engenharia de software 25.10

Para uma propriedade com o nome **`propertyName`***, o par de métodos set/get correspondente seria* **`setPropertyName`**/**`getPropertyName`** *por default. Observe que a primeira letra de* **`propertyName`** *é escrita em maiúscula nos nomes de método set/get.*

Se a propriedade é um tipo de dados **`boolean`**, o par de métodos *set/get* normalmente é definido como

```
public void set NomeDaPropriedade ( boolean value )
public boolean is NomeDaPropriedade ()
```

onde o nome de método *get* começa com a palavra **`is`** em vez de **`get`**.

Quando uma ferramenta de desenvolvimento examina um *bean*, ela procura nos métodos do *bean* pares de métodos *set/get* que possam representar propriedades (algumas ferramentas de desenvolvimento também expõem propriedades de leitura). Esse é um processo conhecido como *introspecção*. Se um par adequado de métodos *set/get* for localizado durante o processo de introspecção, a ferramenta de desenvolvimento expõe esse par de métodos como uma propriedade no *bean*. No primeiro **`LogoAnimator`**, o par de métodos

```
public void setBackground( Color c )
public Color getBackground()
```

que foi herdado da classe **`JPanel`** permitiu à **BeanBox** expor a propriedade **`background`** na janela **Properties** para personalização. Observe que a convenção para atribuição de nomes para o par de métodos *set/get* utiliza uma primeira letra maiúscula para o nome de propriedade, mas a propriedade exposta na folha de propriedades é mostrada com uma primeira letra minúscula.

Observação de engenharia de software 25.11

Quando uma ferramenta de desenvolvimento examina um bean, se ela localiza um par de métodos set/get *que corresponde ao padrão de propriedade do JavaBeans, ela expõe esse par de métodos como uma propriedade no bean.*

Lembre-se de que você deve empacotar a classe **`LogoAnimator2`** como um JavaBean para carregá-la na **BeanBox** ou em uma ferramenta de desenvolvimento. Primeiro compile a classe

```
javac -d . LogoAnimator2.java
```

Isso coloca o diretório do pacote **`jhtp3beans`** no diretório ("**`.`**") atual. Em seguida, empacote a classe em um arquivo JAR

```
jar cfm LogoAnimator2.jar manifest.tmp jhtp3beans\*.*
```

O arquivo manifesto para esse exemplo é mostrado na Fig. 25.31. A linha 1 especifica o nome do arquivo de classe (**`jhtp3beans\LogoAnimator2.class`**) que representa o *bean*. A linha 2 especifica que a classe nomeada na linha 1 é um JavaBean. A linha 3 especifica que **`jhtp3beans.LogoAnimator2`** também é a **`Main-Class`** para esse aplicativo (quando for executado como um aplicativo).

Na Fig. 25.32, mostramos um *bean* de **`LogoAnimator2`** na **BeanBox** com a janela **Properties**. Observe que a propriedade **`animationDelay`** agora está exposta na janela **Properties**. Tente alterar o valor da propriedade para ver seu efeito na velocidade da animação (você deve pressionar *Enter* depois de alterar o valor para efetuar a alteração). Valores menores fazem com que a animação rode mais rapidamente e valores maiores fazem com que a animação rode mais lentamente. Tente digitar 1000 para ver um frame da animação por segundo.

```
1  Main-Class: jhtp3beans.LogoAnimator2
2
3  Name: jhtp3beans/LogoAnimator2.class
4  Java-Bean: True
```

Fig. 25.31 O arquivo manifesto para o *bean* **`LogoAnimator2`**.

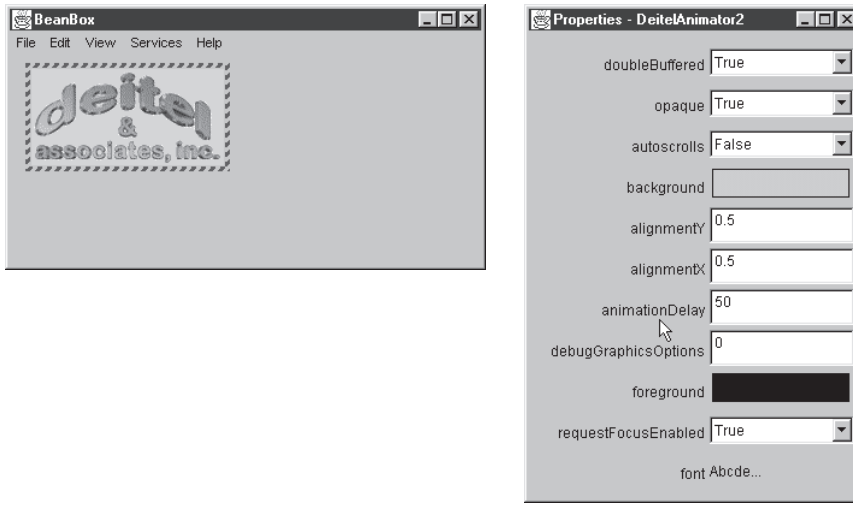


Fig. 25.32 O *bean* **`LogoAnimator2`** com a propriedade **`animationDelay`** exposta na folha de propriedades.

25.8 Criando um JavaBean com uma propriedade associada

Uma *propriedade associada* faz com que o objeto que possui a propriedade notifique outros objetos quando houver uma alteração no valor da propriedade associada. Isso é realizado utilizando recursos de tratamento de eventos-padrão demonstrados anteriormente no texto — todos os ***`PropertyChangeListener`s*** registrados são notificados quando o valor de propriedade muda. Para suportar esse recurso, o *pacote* ***`java.beans`*** fornece a interface **`PropertyChangeListener`** de modo que ouvintes possam ser configurados para receber notificações de alteração na propriedade, a classe ***`PropertyChangeEvent`*** para fornecer informações para um **`PropertyChangeListener`** sobre alterações no valor da propriedade e a classe ***`PropertyChangeSupport`*** para fornecer os serviços de registro e notificação de ouvinte (isto é, manter a lista de ouvintes e notificá-los quando um evento ocorre). Cada tipo é discutido no contexto de sua utilização na Fig. 25.33.

Observação de engenharia de software 25.12

Uma propriedade associada faz com que o objeto que possui a propriedade notifique outros objetos de que houve uma alteração no valor dessa propriedade.

O próximo exemplo apresenta nosso novo componente GUI (**`SliderFieldPanel`**), que estende **`JPanel`** e inclui um objeto do tipo **`JSlider`** e um objeto do tipo **`JTextField`**. Quando o valor **`JSlider`** muda, nosso novo componente GUI atualiza automaticamente o **`JTextField`** com o novo valor. Além disso, quando a um novo valor entra no **`JTextField`** e o usuário pressiona a tecla *Enter*, o **`JSlider`** é automaticamente reposicionado na localização apropriada. Nosso propósito ao definir esse novo componente é vincular um deles com a animação **`LogoAnimator2`** para controlar a velocidade da animação. Quando o valor **`SliderFieldPanel`** muda, queremos alterar a velocidade da animação. A Fig. 25.33 apresenta o código para a classe **`SliderFieldPanel`** e mostra uma captura de tela da aparência do nosso novo componente na **BeanBox**.

```
1 // Fig. 25.33: SliderFieldPanel.java
2 // Uma subclasse de JPanel contendo um JSlider e um JTextField
3 package jhtp3beans;
4
5 import javax.swing.*;
6 import javax.swing.event.*;
7 import java.io.*;
```

Fig. 25.33 Definição da classe **`SliderFieldPanel`** (parte 1 de 4).

```
 8 import java.awt.*;
 9 import java.awt.event.*;
10 import java.beans.*;
11
12 public class SliderFieldPanel extends JPanel
13                              implements Serializable {
14    private JSlider slider;
15    private JTextField field;
16    private Box boxContainer;
17    private int currentValue;
18
19    // objeto para suportar alterações da propriedade de limite
20    private PropertyChangeSupport changeSupport;
21
22    public SliderFieldPanel()
23    {
24       // criar PropertyChangeSupport para propriedades associadas
25       changeSupport = new PropertyChangeSupport( this );
26
27       slider =
28          new JSlider( SwingConstants.HORIZONTAL, 1, 100, 1 );
29       field = new JTextField(
30                  String.valueOf( slider.getValue() ), 5 );
31
32       boxContainer = new Box( BoxLayout.X_AXIS );
33       boxContainer.add( slider );
34       boxContainer.add( Box.createHorizontalStrut( 5 ) );
35       boxContainer.add( field );
36
37       setLayout( new BorderLayout() );
38       add( boxContainer );
39
40       slider.addChangeListener(
41          new ChangeListener() {
42             public void stateChanged( ChangeEvent e )
43             {
44                setCurrentValue( slider.getValue() );
45             }
46          }
47       );
48
49       field.addActionListener(
50          new ActionListener() {
51             public void actionPerformed( ActionEvent e )
52             {
53                setCurrentValue(
54                   Integer.parseInt( field.getText() ) );
55             }
56          }
57       );
58    }
59
60    // métodos para adicionar e remover PropertyChangeListeners
61    public void addPropertyChangeListener(
62       PropertyChangeListener listener )
63    {
64       changeSupport.addPropertyChangeListener( listener );
65    }
```

Fig. 25.33 Definição da classe `SliderFieldPanel` (parte 2 de 4).

```
66
67    public void removePropertyChangeListener(
68       PropertyChangeListener listener )
69    {
70       changeSupport.removePropertyChangeListener( listener );
71    }
72
73    // propriedade minimumValue
74    public void setMinimumValue( int min )
75    {
76       slider.setMinimum( min );
77
78       if ( slider.getValue() < slider.getMinimum() ) {
79          slider.setValue( slider.getMinimum() );
80          field.setText( String.valueOf( slider.getValue() ) );
81       }
82    }
83
84    public int getMinimumValue()
85    {
86       return slider.getMinimum();
87    }
88
89    // propriedade maximumValue
90    public void setMaximumValue( int max )
91    {
92       slider.setMaximum( max );
93
94       if ( slider.getValue() > slider.getMaximum() ) {
95          slider.setValue( slider.getMaximum() );
96          field.setText( String.valueOf( slider.getValue() ) );
97       }
98    }
99
100   public int getMaximumValue()
101   {
102      return slider.getMaximum();
103   }
104
105   // propriedade currentValue
106   public void setCurrentValue( int current )
107   {
108      int oldValue = currentValue;
109      currentValue = current;
110      slider.setValue( currentValue );
111      field.setText( String.valueOf( currentValue ) );
112      changeSupport.firePropertyChange(
113         "currentValue", new Integer( oldValue ),
114         new Integer( currentValue ) );
115   }
116
117   public int getCurrentValue()
118   {
119      return slider.getValue();
120   }
121
122   // propriedade fieldWidth
```

Fig. 25.33 Definição da classe `SliderFieldPanel` (parte 3 de 4).

```
123    public void setFieldWidth( int cols )
124    {
125       field.setColumns( cols );
126       boxContainer.validate();
127    }
128
129    public int getFieldWidth()
130    {
131       return field.getColumns();
132    }
133
134    public Dimension getMinimumSize()
135    {
136       return boxContainer.getMinimumSize();
137    }
138
139    public Dimension getPreferredSize()
140    {
141       return boxContainer.getPreferredSize();
142    }
143 }
```

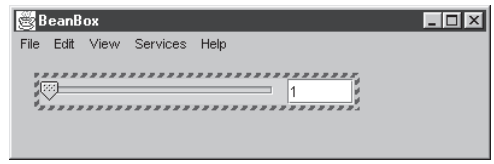


Fig. 25.33 Definição da classe **SliderFieldPanel** (parte 4 de 4).

A classe **SliderFieldPanel** inicia especificando que ela será parte do pacote **jhtp3beans** (linha 3). A classe é uma subclasse de **JPanel**, então podemos adicionar um **JSlider** e um **JTextField** a ela. Os objetos de classe **SliderFieldPanel** podem, então, ser adicionados a outros contêineres.

As declarações de linhas 14 a 20 especificam variáveis de instância do tipo **JSlider**(**slider**) e **JTextField**(**field**), que representam os subcomponentes que o usuário utilizará para configurar o valor do **SliderFieldPanel**, um **Box** (**boxContainer**) que gerenciará o leiaute de objetos de nossa classe, um **int** (**currentValue**) que armazena o valor atual do **SliderFieldPanel** e um **PropertyChangeSupport** (**changeSupport**) que fornecerá os serviços de registro e notificação de ouvinte.

No construtor, a linha 25

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

cria o objeto **PropertyChangeSupport**. O argumento **this** especifica que um objeto dessa classe (**SliderFieldPanel**) é a origem do **PropertyChangeEvent**. As linhas 40 a 47 do construtor registram o **ChangeListener** para **slider**. Quando o valor de **slider** muda, a linha 44 chama **setCurrentValue** para atualizar **field** e notificar os **PropertyChangeListener**s registrados da alteração no valor. De maneira semelhante, as linhas 49 a 57 registram o **ActionListener** para **field**. Quando o valor de **field** muda, as linhas 53 e 54 chamam **setCurrentValue** para atualizar **slider** e notificar os **PropertyChangeListener**s registrados da alteração no valor.

Para suportar o registro de ouvintes para alterações em uma propriedade associada ao nosso **SliderFieldPanel**, definimos os métodos **addPropertyChangeListener** (linhas 61–65) e **removePropertyChangeListener** (linhas 67–71). Cada um desses métodos chama o método correspondente no objeto **PropertyChangeSupport** **changeSupport**. Esse objeto fornecerá os serviços de notificação de evento quando o valor de propriedade se alterar.

Observação de engenharia de software 25.13

Para definir um evento para um bean, você deve fornecer uma interface de ouvinte e uma classe de evento, e o bean deve definir métodos que permitam adicionar e remover ouvintes. Para eventos de propriedade associada, a interface

*de ouvinte e a classe de evento já estão definidas (**PropertyChangeListener** e **PropertyChangeEvent**, respectivamente). Um bean que suporta eventos de propriedade associada deve definir o método **addPropertyChangeListener** e o método **removePropertyChangeListener** para fornecer serviços de registro de ouvinte.*

A classe **SliderFieldPanel** fornece várias propriedades. A propriedade **minimumValue** é definida pelo método **setMinimumValue** (linha 74) e pelo método **getMinimumValue** (linha 84). A propriedade **maximumValue** é definida pelo método **setMaximumValue** (linha 90) e pelo método **getMaximumValue** (linha 100). A propriedade **fieldWidth** é definida pelo método **setFieldWidth** (linha 123) e pelo método **getFieldWidth** (linha 129). Os métodos **getMinimumSize** (linha 134) e **getPreferredSize** (linha 139) são definidos para retornar o tamanho mínimo e o tamanho preferido do objeto **Box boxContainer** que gerencia o leiaute do **JSlider** e **JTextField**.

Observação de engenharia de software 25.14

*Se um bean aparecerá como parte de uma interface com o usuário, o bean deve definir o método **getPreferredSize**, que não aceita nenhum argumento e retorna um objeto **Dimension** contendo a largura e a altura preferidas do bean. Isso ajuda o gerenciador de leiaute a dimensionar o bean.*

Os métodos **setCurrentValue** (linha 106) e **getCurrentValue** (linha 117) definem *a propriedade associada **currentValue***. Quando a propriedade associada se altera, os **PropertyChangeListener**s registrados devem ser notificados da alteração. A especificação do JavaBeans requer que cada ouvinte de propriedade associada seja informado dos valores de propriedade antigos e novos quando notificado da alteração (os valores podem ser **null** se eles não forem necessários). Por essa razão, a linha 108 salva o valor anterior da propriedade. A linha 109 configura o novo valor da propriedade. As linhas 110 e 111 asseguram que o **JSlider** e o **JTextField** mostrem os novos valores apropriados. As linhas 112 a 114 invocam o método **firePropertyChange** do objeto **PropertyChangeSupport** para notificar cada **PropertyChangeListener** registrado. O primeiro argumento é um **String** contendo o nome da propriedade que se alterou — **currentValue**. O segundo argumento é o valor antigo da propriedade. O terceiro argumento é o novo valor da propriedade.

Observação de engenharia de software 25.15

***PropertyChangeListener**s são notificados de um evento de alteração de propriedade com o antigo e o novo valor da propriedade. Se esses valores não forem necessários, eles podem ser **null**.*

Observação de engenharia de software 25.16

*A classe **PropertyChangeSupport** é fornecida como uma conveniência para implementar o suporte de registro e notificação de ouvinte para eventos de alteração de propriedade.*

Lembre-se de que você deve empacotar a classe **SliderFieldPanel** como um JavaBean para carregá-la na **BeanBox** ou em uma ferramenta de desenvolvimento. Primeiro compile a classe

```
javac -d . SliderFieldPanel.java
```

Isso coloca o diretório do pacote **jhtp3beans** no diretório atual ("**.**"). Em seguida, arquive a classe em um arquivo JAR

```
jar cfm SliderFieldPanel.jar manifest.tmp jhtp3beans\*.*
```

O arquivo manifesto para esse exemplo é mostrado na Fig. 25.34. A linha 1 especifica o nome do arquivo de classe (**jhtp3beans\SliderFieldPanel.class**) que representa o *bean*. A linha 2 especifica que a classe nomeada na linha 1 é um JavaBean. Não há uma linha **Main-Class:** nesse arquivo porque o **SliderFieldPanel** não é um aplicativo.

Para demonstrar a funcionalidade da propriedade associada, coloque um *bean* de **SliderFieldPanel** e um *bean* **Juggler** na **BeanBox**. Selecione o *bean* de **SliderFieldPanel**, configure sua propriedade **maximumValue** como 1000 e configure seu **currentValue** como 125 (a velocidade de animação padrão para o **Juggler**). Em seguida, selecione o item **Bind property...** do menu **Edit**. O **PropertyNameDialog** aparece na Fig. 25.35.

Localize o nome da propriedade associada (**currentValue**), selecione-a e clique em **OK**. Uma linha vermelha de seletor de alvo aparece guiando a partir de seu **SliderFieldPanel**. Conecte essa linha com o **Juggler** e clique no mouse. O diálogo na Fig. 25.36 aparece.

```
1  Name: jhtp3beans/SliderFieldPanel.class
2  Java-Bean: True
```

Fig. 25.34 Arquivo manifesto para o JavaBean **`SliderFieldPanel`**.

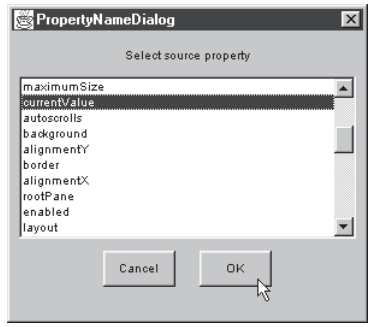


Fig. 25.35 Selecionando a propriedade associada do **PropertyNameDialog**.

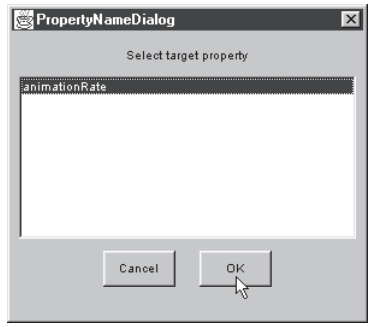


Fig. 25.36 Selecionando a propriedade associada do **PropertyNameDialog**.

O **`PropertyNameDialog`** mostra apenas as propriedades de alvo que têm o mesmo tipo de dados que a propriedade associada. Selecione a propriedade **`animationRate`** do **`Juggler`** (a única exibida) e clique em **OK**. A propriedade **`animationRate`** está agora associada com a propriedade **`currentValue`** do **`SliderFieldPanel`**. A Fig. 25.37 mostra o **`SliderFieldPanel`** e o **`Juggler`** na **BeanBox**.

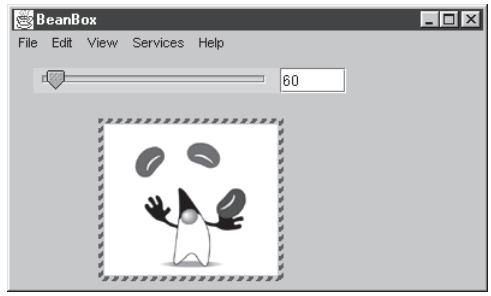


Fig. 25.37 A janela da **BeanBox** com os *beans* **`SliderFieldPanel`** e **`Juggler`**.

Tente ajustar o controle deslizante para ver a alteração na velocidade da animação. Mova o controle deslizante para a esquerda para ver a velocidade da animação aumentar e mova o controle deslizante para a direita para ver a velocidade de animação diminuir. Além disso, tente digitar um novo valor no campo de texto e pressionar *Enter* para alterar a velocidade de animação.

25.9 Especificando a classe `BeanInfo` para um JavaBean

Como mencionado anteriormente, o mecanismo de introspecção de Java pode ser utilizado por uma ferramenta de desenvolvimento para expor automaticamente propriedades, métodos e eventos de um JavaBean se o programador seguir *padrões* adequados de projeto de JavaBean (como as convenções de atribuição de nomes especiais discutidas para pares de método *set/get* que definem uma propriedade de um *bean*). As ferramentas de desenvolvimento utilizam as classes e interfaces do pacote ***`java.lang.reflect`*** para realizar serviços de introspecção. Para os JavaBeans que não seguem os padrões de projeto do JavaBean ou para aqueles JavaBeans em que o programador quer personalizar o conjunto exposto de propriedades, métodos e eventos, o programador pode fornecer uma classe que implementa a interface ***`BeanInfo`*** (pacote **`java.beans`**). A classe **`BeanInfo`** descreve para a ferramenta de desenvolvimento como apresentar os recursos do *bean* para o programador.

Observação de engenharia de software 25.17

Propriedades, métodos e eventos de um JavaBean podem ser expostos automaticamente por uma ferramenta de desenvolvimento se o programador seguir os padrões adequados de projeto de JavaBean.

Observação de engenharia de software 25.18

*Cada classe **`BeanInfo`** deve implementar a interface **`BeanInfo`**. Essa interface descreve os métodos utilizados por uma ferramenta de desenvolvimento para determinar os recursos de um bean descrito pela classe **`BeanInfo`** desse bean.*

Observação de engenharia de software 25.19

*Uma classe **`BeanInfo`** pode ser utilizada para descrever um JavaBean para uma ferramenta de desenvolvimento de modo que a ferramenta possa apresentar os recursos do bean para um programador. Isso é útil para os JavaBeans que não seguem os padrões de projeto do JavaBean ou para aqueles JavaBeans em que o programador quer personalizar o conjunto exposto de propriedades, métodos e eventos.*

No último exemplo, você pode ter reparado, quando selecionou o *bean* de **`SliderFieldPanel`** na **BeanBox**, que 14 propriedades diferentes foram expostas e 10 categorias de eventos (veja o item de menu **Events** do menu **Edit** da **BeanBox**) foram expostas. Para esse *bean*, queremos que o programador veja apenas as propriedades **`fieldWidth`**, **`currentValue`**, **`minimumValue`** e **`maximumValue`** (as outras propriedades foram herdadas da classe **`JPanel`** e não são verdadeiramente relevantes para nosso *bean*). Além disso, o único evento que queremos que o programador utilize para nosso componente é o evento da propriedade associada.

A Fig. 25.38 apresenta a classe **`SliderFieldPanelBeanInfo`** para personalizar as propriedades e eventos expostos na **BeanBox** (ou qualquer outra ferramenta de desenvolvimento) para nosso *bean* **`SliderFieldPanel`**. As capturas de tela na Fig. 25.38 mostram os recursos expostos do JavaBean **`SliderFieldPanel`**.

Observação de engenharia de software 25.20

*Por default, a classe **`BeanInfo`** tem o mesmo nome que o bean e termina com **`BeanInfo`**.*

Observação de engenharia de software 25.21

*Por default, a classe **`BeanInfo`** é incluída no mesmo arquivo JAR que o JavaBean **`SliderFieldPanel`**. Quando o bean é carregado, a **BeanBox** (ou ferramenta de desenvolvimento) determina se o arquivo JAR contém uma classe **`BeanInfo`** para um bean. Se uma classe **`BeanInfo`** é localizada, ela é utilizada para determinar os recursos expostos do bean. Caso contrário, a introspecção-padrão é utilizada para determinar os recursos expostos do bean.*

Observação de engenharia de software 25.22

*Por default, a classe **`BeanInfo`** é colocada no mesmo **`package`** que o bean que ela descreve.*

```
1  // Fig. 25.38: SliderFieldPanelBeanInfo.java
2  // A classe BeanInfo para SliderFieldPanel
3  package jhtp3beans;
4
5  import java.beans.*;
6
7  public class SliderFieldPanelBeanInfo extends SimpleBeanInfo {
8     public final static Class beanClass =
9        SliderFieldPanel.class;
10
11    public PropertyDescriptor[] getPropertyDescriptors()
12    {
13       try {
14          PropertyDescriptor fieldWidth =
15             new PropertyDescriptor( "fieldWidth", beanClass );
16          PropertyDescriptor currentValue =
17             new PropertyDescriptor(
18                "currentValue", beanClass );
19          PropertyDescriptor maximumValue =
20             new PropertyDescriptor(
21                "maximumValue", beanClass );
22          PropertyDescriptor minimumValue =
23             new PropertyDescriptor( "minimumValue", beanClass );
24
25          // assegura que PropertyChangeEvent ocorre para essa
26          currentValue.setBound( true ); // propriedade
27
28          PropertyDescriptor descriptors[] = { fieldWidth,
29             currentValue, maximumValue, minimumValue };
30
31          return descriptors;
32       }
33       catch ( IntrospectionException ie ) {
34          throw new RuntimeException( ie.toString() );
35       }
36    }
37
38    // o índice para a propriedade de currentValue
39    public int getDefaultPropertyIndex()
40    {
41       return 1;
42    }
43
44    public EventSetDescriptor[] getEventSetDescriptors() {
45       try {
46          EventSetDescriptor changed =
47             new EventSetDescriptor( beanClass,
48             "propertyChange",
49             java.beans.PropertyChangeListener.class,
50             "propertyChange");
51
52          changed.setDisplayName(
53             "SliderFieldPanel value changed" );
54
55          EventSetDescriptor[] descriptors = { changed };
56
57          return descriptors;
```

Fig. 25.38 Demonstrando a classe `SliderFieldPanelBeanInfo` na **BeanBox** (parte 1 de 2).

```
58         }
59         catch (IntrospectionException e) {
60            throw new RuntimeException(e.toString());
61         }
62      }
63   }
```

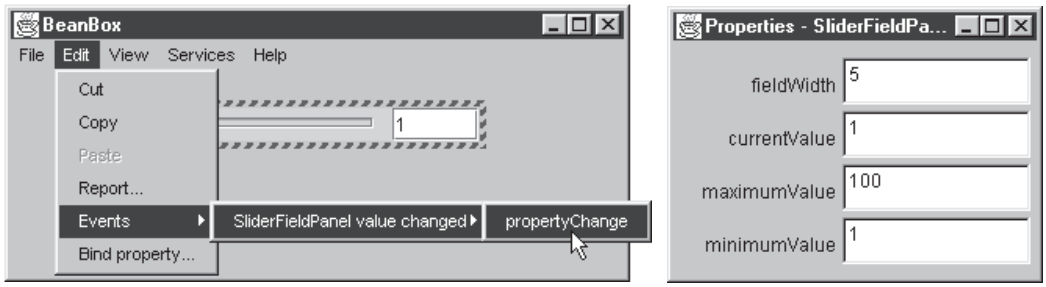


Fig. 25.38 Demonstrando a classe `SliderFieldPanelBeanInfo` na **BeanBox** (parte 2 de 2).

Cada classe **`BeanInfo`** deve implementar a interface **`BeanInfo`**. Essa interface descreve os métodos utilizados por uma ferramenta de desenvolvimento para determinar os recursos expostos do *bean* descrito por sua classe **`BeanInfo`** correspondente. Como uma conveniência, o pacote **`java.beans`** inclui a classe ***`SimpleBeanInfo`***, que fornece uma implementação-padrão de cada método da interface **`BeanInfo`**. O programador pode estender essa classe e redefinir seletivamente seus métodos para implementar uma classe **`BeanInfo`** adequada. A classe **`SliderFieldPanelBeanInfo`** estende a classe **`SimpleBeanInfo`** (linha 7).

Observação de engenharia de software 25.23

*A classe **`SimpleBeanInfo`** fornece uma implementação-padrão de cada método da interface **`BeanInfo`**. O programador pode redefinir seletivamente os métodos dessa classe para implementar uma classe **`BeanInfo`** adequada.*

Nesse exemplo, redefinimos os métodos ***`getPropertyDescriptors`***, ***`getDefaultPropertyIndex`*** e ***`getEventSetDescriptors`*** de **`BeanInfo`**.

As linhas 8 e 9

```
public final static Class beanClass =
   SliderFieldPanel.class;
```

definem uma referência (**`beanClass`**) para um objeto ***`Class`***. A classe **`Class`** (definida no pacote **`java.lang`**) permite a um programa referenciar uma definição de classe (e também fornece muitos outros recursos interessantes). A classe **`Class`** é utilizada pelo mecanismo de introspecção para referenciar a classe que será procurada por recursos descritos na classe **`BeanInfo`**. As linhas 8 e 9 fazem com que o arquivo **`SliderFieldPanel.class`** seja carregado na memória. A referência é declarada **`final`** porque não deve ser modificada na definição de classe. A referência é declarada **`static`** porque apenas uma instância do **`Class`** é necessária.

As linhas 11 a 36 redefinem o método **`getPropertyDescriptors`** para retornar um *array* de objetos ***`PropertyDescriptor`***. Cada **`PropertyDescriptor`** indica uma propriedade específica que deve ser exposta por uma ferramenta de desenvolvimento. Há várias maneiras de construir um **`PropertyDescriptor`**. Nesse exemplo, cada uma das chamadas de construtor **`PropertyDescriptor`** tem a forma

```
new PropertyDescriptor( "nomeDaPropriedade", classeDoBean );
```

onde *nomeDaPropriedade* é um **`String`** que especifica o nome de uma propriedade definida pelo par de métodos **`set`***NomeDaPropriedade* e **`get`***NomeDaPropriedade*. Observe que o **`String`** *nomeDaPropriedade* deve iniciar com uma letra minúscula e os métodos de propriedade correspondentes começam o nome de propriedade com uma primeira letra maiúscula. Definimos **`PropertyDescriptor`**s para **`fieldWidth`**, **`currentValue`**, **`minimumValue`** e **`maximumValue`**.

Observação de engenharia de software 25.24

Se os métodos set/get *para uma propriedade não utilizam a convenção para atribuição de nomes do JavaBeans para propriedades, há dois outros construtores* **`PropertyDescriptor`** *em que os nomes reais de método são passados. Isso permite às ferramentas de desenvolvimento utilizar métodos de propriedade não-padrão a fim de expor uma propriedade para manipulação pelo programador em tempo de projeto. Isso é particularmente útil no ajustamento de uma classe como um JavaBean quando essa classe não foi originalmente projetada e implementada utilizando padrões de projeto JavaBeans.*

A linha 26

```
currentValue.setBound( true );
```

especifica que a propriedade **`currentValue`** é uma propriedade associada. Algumas ferramentas de desenvolvimento tratam em seu aspecto visual eventos de propriedade associada separadamente de outros eventos. Por exemplo, a **BeanBox** tem o item **Bind property...** do menu **Edit** para permitir ao programador enganchar um evento de propriedade associada. Observe que a **BeanBox** também fornece uma entrada para o evento de propriedade associada no item **Events** do menu **Edit**.

As linhas 28 e 29 criam o *array* **`PropertyDescriptor`** que é retornado pelo método na linha 31. Observe o tratador de exceção para ***`IntrospectionException`***`s`. Se um construtor **`PropertyDescriptor`** é incapaz de confirmar a propriedade no objeto **`Class`** correspondente que representa a definição de classe, o construtor dispara uma **`IntrospectionException`**. Uma vez que a classe **`BeanInfo`** e seus métodos são realmente utilizados pela ferramenta de desenvolvimento em tempo de projeto (isto é, durante o desenvolvimento do programa no IDE), a **`RuntimeException`** disparada pelo tratador **`catch`** normalmente seria capturada pela ferramenta de desenvolvimento.

As linhas 39 a 42 definem o método ***`getDefaultPropertyIndex`*** para retornar o valor 1, para indicar que a propriedade na posição 1 no *array* de **`PropertyDescriptor`** que retornou de **`getPropertyDescriptors`** é a *propriedade-padrão* para que os desenvolvedores personalizem em uma ferramenta de desenvolvimento. Em geral, a propriedade-padrão é automaticamente selecionada quando você clica em um *bean*. Nesse exemplo, a propriedade **`currentValue`** é a propriedade padrão.

As linhas 44 a 62 redefinem o método ***`getEventSetDescriptors`*** para retornar um *array* de objetos ***`EventSetDescriptor`*** que descrevem para uma ferramenta de desenvolvimento os eventos suportados por esse *bean*. As linhas 46 a 50

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

definem um objeto **`EventSetDescriptor`** para o **`PropertyChangeEvent`** associado com a propriedade associada **`currentValue`**. Os quatro argumentos para o construtor descrevem o evento que deve ser exposto pela ferramenta de desenvolvimento. O primeiro argumento é o objeto **`Class`** (**`beanClass`**) para representar a *origem do evento* (o *bean* que gera o evento). O segundo argumento é um **`String`** representando o nome do *conjunto de eventos* (por exemplo, o conjunto de eventos **`mouse`** inclui **`mousePressed`**, **`mouseClicked`**, **`mouseReleased`**, **`mouseEntered`** e **`mouseExited`**). Nesse exemplo, o nome do conjunto de eventos é ***`propertyChange`***.

Observação de engenharia de software 25.25

Ao utilizar padrões de projeto JavaBeans, o nome do conjunto de eventos é parte de todos os nomes de tipo de dados e nomes de método utilizados para processar o evento. Por exemplo, os tipos e métodos para o conjunto de eventos ***`propertyChange`*** *são:* ***`PropertyChangeListener`*** *(a interface que um objeto deve implementar para ser notificado de um evento nesse conjunto de eventos),* ***`PropertyChangeEvent`*** *(o tipo passado para um método ouvinte de um evento nesse conjunto de eventos),* ***`addPropertyChangeListener`*** *(o método chamado para adicionar um ouvinte de um evento nesse conjunto de eventos),* ***`removePropertyChangeListener`*** *(o método chamado para remover um ouvinte de um evento nesse conjunto de eventos) e* ***`firePropertyChange`*** *(o método chamado para notificar ouvintes quando um evento nesse conjunto de eventos ocorre — esse método é identificado dessa maneira apenas por convenção).*

O terceiro argumento é o objeto **`Class`** para representar a interface de ouvinte de evento implementada por ouvintes desse evento. Esse argumento é especificado com

```
java.beans.PropertyChangeListener.class
```

que carrega o arquivo "**`.class`**" e automaticamente cria um objeto anônimo do tipo **`Class`**. Por fim, o último argumento é um **`String`** representando o nome do método ouvinte a ser chamado (**`propertyChange`**) quando esse evento ocorre.

Observação de engenharia de software 25.26

***`EventSetDescriptor`**s podem ser construídos com outros argumentos para expor eventos que não seguem os padrões de projeto convencionais do JavaBeans .*

Uma vantagem de um **`EventSetDescriptor`** é personalizar o nome do conjunto de eventos a exibir na ferramenta de desenvolvimento. As linhas 52 e 53

```
changed.setDisplayName(
   "SliderFieldPanel value changed" );
```

chamam o método ***`setDisplayName`*** no **`EventSetDescriptor`** para indicar que seu nome exibido deve ser "**`SliderFieldPanel value changed`**" (veja a captura de tela na Fig. 25.38).

Boa prática de programação 25.1

Personalizar o nome do conjunto de eventos exibido por uma ferramenta de desenvolvimento pode tornar a finalidade desse conjunto de eventos mais compreensível para o programador que utiliza o bean.

As linhas 55 e 57 criam o *array* **`EventSetDescriptor`** e o retornam, respectivamente. Observe o tratador de exceções para **`IntrospectionException`**s na linha 59. Se um construtor **`EventSetDescriptor`** for incapaz de confirmar o evento no objeto **`Class`** correspondente que representa a definição de classe, o construtor dispara (**`throws`**) uma **`IntrospectionException`**.

25.10 Recursos sobre JavaBeans na World Wide Web

Se você tem acesso à Internet e à World Wide Web, há um grande número de recursos sobre JavaBean disponíveis para você. O melhor lugar para começar é na fonte — o site da Web de Java **`http://java.sun.com`** da Sun Microsystems, Inc.

`http://java.sun.com/beans/`

Essa é a *JavaBeans Home Page* no site da Web da Sun Microsystems, Inc. Aqui, você pode descarregar o *Beans Development Kit (BDK)* e outros programas de software relacionados com *beans*. Outros recursos do site incluem: documentação e especificações do JavaBeans, uma lista de perguntas feitas com freqüência, uma visão geral de ambientes integrados de desenvolvimento (*Integrated Development Environments*) que suportam o desenvolvimento de JavaBeans, treinamento e suporte, próximos eventos relacionados com JavaBeans, um diretório pesquisável de componentes JavaBeans, uma área de suporte para vender seus JavaBeans e uma variedade de recursos on-line para comunicar-se com outros programadores sobre JavaBeans.

`http://java.sun.com/beans/spec.html`

Visite esse site para descarregar as especificações dos JavaBeans.

`http://java.sun.com/beans/tools.html`

Visite esse site para as informações sobre ferramentas de desenvolvimento de JavaBeans.

`http://java.sun.com/beans/directory/`

Visite esse site para um diretório pesquisável de *beans* disponíveis.

`http://java.sun.com/products/hotjava/bean/index.html`

Descarregue uma versão de avaliação do *HotJava HTML Component*. Esse JavaBean permite que os desenvolvedores forneçam capacidades de renderizar HTML em qualquer aplicativo Java. O *bean* suporta protocolo HTTP 1.1, HTML 3.2 e muitos recursos-padrão de navegadores da Web.

Resumo

- A **BeanBox** é um contêiner de teste para seus JavaBeans.
- Depois de instalar o *JavaBeans Development Kit*, você pode executar a **BeanBox** localizando o diretório **`BDK1.1`** onde você instalou o *JavaBeans Development Kit*. O subdiretório **`beanbox`** desse diretório contém arquivos de inicialização para Windows (**`run.bat`**) e Solaris (**`run.sh`**).
- O diretório **`doc`** contém os arquivos em HTML de ajuda on-line para o **BeanBox** (utilize o arquivo **`beanbox.html`** para introdução). Os arquivos de ajuda discutem profundamente todos os recursos da **BeanBox** .
- As quatro janelas da **BeanBox** são **ToolBox**, **BeanBox** , **Properties** e **Method Tracer**.
- Um editor de propriedades permite que o programador personalize o valor de uma propriedade.
- Ao selecionar um *bean* da **ToolBox**, o cursor do mouse deve mudar para um cursor em forma de cruz. Posicione o cursor do mouse sobre a janela da **BeanBox** onde você quer que o centro de seu *bean* seja localizado e clique com o mouse para posicionar o *bean*.
- Para personalizar um *bean*, selecione o *bean* clicando nele com o mouse. Se o clique em um *bean* na janela da **BeanBox** não o seleciona, você pode precisar simplesmente clicar fora do limite do *bean* para selecioná-lo. As propriedades do *bean* atualmente selecionado são exibidas na janela **Properties**.
- O item de menu **Events** do menu **Edit** da janela **BeanBox** fornece acesso aos eventos suportados por um *bean* que é uma origem de evento (isto é, qualquer *bean* que pode notificar um ouvinte de que um evento ocorreu).
- Quando você mover o mouse pela janela **BeanBox** depois de selecionar um evento, você verá uma linha seletora vermelha de alvo acompanhando o mouse. Isso é utilizado para especificar o *alvo* do evento. Clique no objeto-alvo para exibir a janela **EventTargetDialog** listando os métodos **`public`** que podem ser chamados no alvo.
- A **BeanBox** fornece a capacidade de salvar um projeto que será recarregado na **BeanBox** mais tarde (o item **Save...** do menu **File**) ou salvar um projeto como um *applet* Java (o item **MakeApplet...** do menu **File**).
- A propriedade **`archive`** do *tag* **`<applet>`** especifica uma lista separada por vírgulas de arquivos JAR contendo o código que é utilizado para executar o *applet*.
- As classes que representam um *bean* são colocadas em um pacote utilizando a opção **`-d`** com o compilador Java.
- Todos os JavaBeans devem implementar a interface **`Serializable`** para que possam ser salvos a partir de uma ferramenta de desenvolvimento depois de serem personalizados pelo programador.
- Para utilizar uma classe como um JavaBean, ela deve primeiro ser colocada em um arquivo JAR. Um arquivo de texto chamado **`manifest.tmp`** é utilizado pelo utilitário **`jar`** para descrever o conteúdo do arquivo JAR. O utilitário **`jar`** utiliza o arquivo **`manifest.tmp`** para criar um arquivo chamado **`MANIFEST.MF`** que é incluído no diretório **`META-INF`** do repositório de arquivos.
- Quando um arquivo JAR contendo um JavaBean (ou um conjunto de JavaBeans) é carregado em um IDE, o IDE examina o arquivo manifesto para determinar as classes no JAR que representam JavaBeans. Essas classes são disponibilizadas para utilização pelo programador de uma maneira visual.
- Quando uma classe de aplicativo contendo **`main`** é armazenada em um JAR, o aplicativo pode ser executado diretamente do JAR utilizando o interpretador Java com a opção de linha de comando **`-jar`** como segue:

```
java -jar LogoAnimator.jar
```

  O interpretador examina em um arquivo manifesto dentro de um JAR a classe especificada como a **`Main-Class`** para iniciar a execução com seu método **`main`**.
- Com Java 1.2 (a plataforma do Java 2), muitas plataformas pressupõem que um arquivo JAR contém um aplicativo e tentam executá-lo.
- O comando

```
jar cfm NomeDoArquivoJAR.jar manifest.tmp arquivos
```

  cria o arquivo JAR. No comando precedente, **`jar`** é o utilitário de repositórios de arquivo Java utilizado para criar arquivos JAR. A seguir, estão as opções para o utilitário **`jar cfm`**. A letra **`c`** indica que estamos criando um arquivo JAR. A letra **`f`** indica que o próximo argumento é o nome do arquivo JAR. A letra **`m`** indica que o próximo argumento é o arquivo **`manifest.tmp`** que será utilizado por **`jar`** para criar **`MANIFEST.MF`** no diretório **`META-INF`** do JAR. Depois das opções, o nome de arquivo JAR e o arquivo **`manifest.tmp`** são os *arquivos* reais para incluir no arquivo JAR.
- A estrutura de diretórios no arquivo JAR deve corresponder à estrutura de diretórios utilizada no arquivo **`manifest.tmp`**. Ambos devem corresponder à estrutura de pacote.
- Há duas maneiras de carregar o *bean* na **BeanBox**: coloque o arquivo JAR no diretório **`BDK1.1\jars`** ou utilize a opção

LoadJar... do menu **File** para localizar o arquivo JAR em seu sistema e carregá-lo no **ToolBox** da **BeanBox**. Se você colocar o arquivo JAR no diretório **`BDK1.1\jars`**, ele será automaticamente carregado na **ToolBox** quando a **BeanBox** for executada.

- Uma propriedade de leitura e gravação de um *bean* é definida como um par de métodos *set/get* da seguinte forma:

```
public void set NomeDaPropriedade( TipoDeDados value )
public TipoDeDados get NomeDaPropriedade()
```

Se a propriedade é do tipo de dados **`boolean`**, o par de métodos *set/get* normalmente é definido assim:

```
public void set NomeDaPropriedade( boolean value )
public boolean is NomeDaPropriedade()
```

Quando uma ferramenta de desenvolvimento examina um *bean*, se ela localiza um par de métodos *set/get* com os formatos precedentes, a ferramenta de desenvolvimento expõe esse par de métodos como uma propriedade do *bean*.

- Uma propriedade associada faz com que o objeto que possui a propriedade notifique outros objetos de que houve uma alteração no valor dessa propriedade. Todos os **`PropertyChangeListener`**s registrados são automaticamente notificados quando o valor da propriedade se altera. O pacote **`java.beans`** fornece a interface **`PropertyChangeListener`** para criar ouvintes que podem receber notificação de alteração na propriedade, a classe **`PropertyChangeEvent`** para fornecer informações para um **`PropertyChangeListener`** sobre a alteração no valor da propriedade e a classe **`PropertyChangeSupport`** para fornecer os serviços de registro e notificação de ouvinte.
- Para suportar o registro de ouvintes para alterações em uma propriedade associada, um *bean* deve fornecer os métodos **`addPropertyChangeListener`** e **`removePropertyChangeListener`**. Cada um desses métodos em geral chama o método correspondente em um objeto **`PropertyChangeSupport`** que fornece os serviços de notificação de evento quando o valor da propriedade se altera.
- Quando uma propriedade associada se altera, os **`PropertyChangeListener`**s registrados devem ser notificados da alteração. Cada ouvinte recebe os valores de propriedade antigos e novos quando notificado da alteração. O método **`firePropertyChange`** do objeto **`PropertyChangeSupport`** notifica cada **`PropertyChangeListener`** registrado.
- Quando um *bean* fornece propriedades associadas, haverá um item de menu **Bind property...** no menu **Edit** da **BeanBox**. Selecione o item **Bind property...** do menu **Edit** para exibir o **PropertyNameDialog**. Selecione a propriedade associada. Conecte a linha seletora de alvo vermelha com o alvo do evento de alteração de propriedade para selecionar o método a chamar.
- Utilizando introspecção, pode-se expor automaticamente propriedades, métodos e eventos de um JavaBean por meio de uma ferramenta de desenvolvimento se o programador segue os padrões adequados de projeto de JavaBean. Para JavaBeans que não seguem os padrões de projeto de JavaBean ou JavaBeans em que o programador quer personalizar o conjunto exposto de propriedades, métodos e eventos, o programador pode fornecer uma classe **`BeanInfo`** que descreve para a ferramenta de desenvolvimento como apresentar os recursos do *bean*.
- Cada classe **`BeanInfo`** deve implementar a interface **`BeanInfo`** que descreve os métodos utilizados por uma ferramenta de desenvolvimento para determinar os recursos de um *bean*. A classe **`SimpleBeanInfo`** fornece uma implementação-padrão de cada método na interface **`BeanInfo`**. O programador pode estender essa classe e redefinir seletivamente métodos dessa classe para implementar uma classe **`BeanInfo`** adequada.
- Redefina o método **`getPropertyDescriptors`** para retornar um *array* de objetos **`PropertyDescriptor`** em que **`PropertyDescriptor`** indica uma propriedade que uma ferramenta de desenvolvimento deve expor.
- Se os métodos *set*/*get* de uma propriedade não seguem a convenção de atribuição de nomes de JavaBeans para propriedades, há dois outros construtores **`PropertyDescriptor`** em que os nomes de método reais são passados. Isso permite que as ferramentas de desenvolvimento utilizem métodos de propriedade não-padrão a fim de expor uma propriedade para manipulação pelo programador em tempo de projeto.
- Redefina o método **`getEventSetDescriptors`** para retornar um *array* de objetos **`EventSetDescriptor`** que descrevem os eventos suportados por um *bean* para uma ferramenta de desenvolvimento.
- **`EventSetDescriptor`**s podem ser construídos com argumentos diferentes para expor eventos que não seguem os padrões de projeto de JavaBeans convencionais.

Terminologia

adicionando eventos a um *bean*
adicionando propriedades a um *bean*
alvo de evento
alvo de um evento
arquivo manifesto
BDK (*JavaBeans Development Kit*)
bean
bean de demonstração **`ExplicitButton`** **BeanBox**
bean de demonstração **`Juggler`** **BeanBox**
BeanBox

caixa de diálogo **Choose JAR File**
caixa de diálogo **EventTargetDialog**
caixa de diálogo **Load saved BeanBox**
caixa de diálogo **Make an Applet**
caixa de diálogo **PropertyNameDialog**
caixa de diálogo **Save BeanBox File**
caixa de seleção
campos e classes **`transient`** (não-serializados)
classe adaptadora
classe adaptadora de evento
classe de enganchamento
classe **`EventSetDescriptor`**
classe **`IntrospectionException`**
classe **`java.beans.EventSetDescriptor`**
classe **`java.beans.PropertyDescriptor`**
classe **`java.beans.SimpleBeanInfo`**
classe **`java.io.ObjectInputStream`**
classe **`java.io.ObjectOutputStream`**
classe **`ObjectInputStream`**
classe **`ObjectOutputStream`**
classe **`PropertyChangeEvent`**
classe **`PropertyChangeSupport`**
classe **`PropertyDescriptor`**
classe **`SimpleBeanInfo`**
colocando um *bean* na **BeanBox**
conectando *beans*
conjunto de eventos
criando um arquivo JAR
cursor de movimentação (move)
cursor de redimensionamento
"desserializar" um objeto
diretório **`BDK`**
diretório **`BDK\jars`**
editor de propriedades **ColorEditor** da **BeanBox**
enganchar um evento
evento
executar um aplicativo a partir de um JavaBean
expondo eventos de um *bean*
expondo propriedades de um *bean*
extensão **`.ser`** (arquivo serializado)
extensão de arquivo **`.jar`** (*Java archive*)
ferramenta de desenvolvimento
folha de propriedades
interface **`BeanInfo`**
interface **`java.beans.BeanInfo`**
interface **`java.io.Serializable`**
interface **`PropertyChangeListener`**
interface **`Serializable`**
introspecção
item de menu **Clear** do menu **File**
item de menu **Events** do menu **Edit**
item de menu **Load...** do menu **File**
item de menu **MakeApplet...** do menu **File**
item de menu **SerializeComponent...** do menu **File**
janela **BeanBox** da **BeanBox**
janela **Properties** da **BeanBox**
janela **ToolBox** da **BeanBox**
Java Archive File (arquivo JAR)
`java -jar` (executar um aplicativo a partir de um *bean*)
`java.beans.IntrospectionException`
`java.beans.PropertyChangeEvent`
`java.beans.PropertyChangeListener`
`java.beans.PropertyChangeSupport`
JavaBean
JavaBeans Development Kit (BDK)
linha seletora de alvo
`META-INF` diretório de um arquivo JAR
método **`addPropertyChangeListener`**
método **`firePropertyChange`**
método *get*
método *get* de propriedade
método **`getDefaultPropertyIndex`**
método **`getEventSetDescriptors`**
método **`getPropertyDescriptors`**
método **`removePropertyChangeListener`**
método *set*
método *set* de propriedade
origem de evento
ouvinte de evento
pacote **`java.beans`**
pacote **`java.lang.reflect`**
persistência
personalize um JavaBean
programação "ligue os pontos"
projeto-padrão
propriedade
propriedade **`archive`** do *tag* **`<applet>`**
propriedade associada de um *bean*
serialização de objeto
serializando um *bean*
subdiretório **`beanbox`** do diretório **`BDK`**
utilitário **`jar`** (*Java Archive File*)
visualizando o conteúdo de um arquivo JAR

Erro comum de programação

25.1 Não especificar um arquivo manifesto ou especificar um arquivo manifesto com sintaxe incorreta ao criar um arquivo JAR é um erro. As ferramentas de desenvolvimento não reconhecerão os *beans* no arquivo JAR.

Boa prática de programação

25.1 Personalizar o nome do conjunto de eventos exibido por uma ferramenta de desenvolvimento pode tornar a finalidade desse conjunto de eventos mais compreensível para o programador que utiliza o *bean*.

Observações de engenharia de software

25.1 A **BeanBox** não é uma ferramenta de desenvolvimento. Em vez disso, a **BeanBox** permite que os programadores visualizem como um *bean* será exibido e utilizado por uma ferramenta de desenvolvimento.

25.2 Um benefício de trabalhar em um ambiente de desenvolvimento de *beans* prontos é que o ambiente apresenta visualmente as propriedades do *bean* para o programador a fim de facilitar sua modificação e personalização durante o projeto.

25.3 Um benefício de trabalhar em um ambiente de desenvolvimento de *beans* prontos é que os *beans*, em geral, executam no ambiente de desenvolvimento. Isso permite visualizar e testar a funcionalidade de seu programa imediatamente no ambiente de projeto em vez de utilizar o ciclo de programação padrão de editar, compilar e executar.

25.4 Todos os JavaBeans devem implementar a interface **`Serializable`** para suportar persistência.

25.5 Você deve definir um arquivo manifesto que descreve o conteúdo de um arquivo JAR se pretende utilizar o *bean* em um ambiente de desenvolvimento integrado compatível com JavaBean ou se pretende executar um aplicativo diretamente a partir de um arquivo JAR.

25.6 No arquivo manifesto, o nome de um *bean* é especificado com a propriedade **`Name:`** seguida pelo nome do pacote completo e o nome de classe do *bean*. Os pontos (**`.`**) normalmente utilizados para separar os nomes de pacote dos nomes de classe são substituídos por barras normais (**`/`**) nessa linha do arquivo manifesto.

25.7 Se uma classe específica representa um *bean*, a linha que se segue à propriedade **`Name:`** dessa classe deve especificar **`Java-Bean: True`**. Caso contrário, os IDEs não reconhecerão a classe como um *bean*.

25.8 Se uma classe contendo **`Main`** é incluída em um arquivo JAR, essa classe pode ser utilizada pelo interpretador para executar o aplicativo diretamente do arquivo JAR especificando a propriedade **`Main-Class:`** em uma linha isolada no início do arquivo manifesto. O nome completo do pacote e o nome completo da classe devem ser especificados com o ponto normal (**`.`**) separando os nomes de pacote e o nome da classe.

25.9 Uma propriedade de leitura/gravação do JavaBean é definida por um par de métodos *set*/*get* em que o método *set* retorna **`void`** e aceita um argumento, e o método *get* retorna o mesmo tipo de argumento correspondente que o do método *set* e não aceita nenhum argumento. Também é possível ter propriedades somente de leitura (definidas apenas com um método *get*) e propriedades somente de escrita (definidas apenas com um método *set*).

25.10 Para uma propriedade com o nome **`propertyName`**, o par de métodos *set*/*get* correspondente seria **`setPropertyName`**/**`getPropertyName`** por default. Observe que a primeira letra de **`propertyName`** é escrita em maiúscula nos nomes de método *set*/*get*.

25.11 Quando uma ferramenta de desenvolvimento examina um *bean*, se ela localiza um par de métodos *set*/*get* que corresponde ao padrão de propriedade do JavaBeans, ela expõe esse par de métodos como uma propriedade do *bean*.

25.12 Uma propriedade associada faz com que o objeto que possui a propriedade notifique outros objetos de que houve uma alteração no valor dessa propriedade.

25.13 Para definir um evento para um *bean*, você deve fornecer uma interface de ouvinte e uma classe de evento, e o *bean* deve definir métodos que permitam adicionar e remover ouvintes. Para eventos de propriedade associada, a interface de ouvinte e a classe de evento já estão definidas (**`PropertyChangeListener`** e **`PropertyChangeEvent`**, respectivamente). Um *bean* que suporta eventos de propriedade associada deve definir o método **`addPropertyChangeListener`** e o método **`removePropertyChangeListener`** para fornecer serviços de registro de ouvinte.

25.14 Se um *bean* aparecerá como parte de uma interface com o usuário, o *bean* deve definir o método **`getPreferredSize`**, que não aceita nenhum argumento e retorna um objeto **`Dimension`** contendo a largura e a altura preferidas do *bean*. Isso ajuda o gerenciador de leiaute a dimensionar o *bean*.

25.15 **`PropertyChangeListener`**s são notificados de um evento de alteração de propriedade com o antigo e o novo valor da propriedade. Se esses valores não forem necessários, eles podem ser **`null`**.

25.16 A classe **`PropertyChangeSupport`** é fornecida como uma conveniência para implementar o suporte de registro e notificação de ouvinte para eventos de alteração de propriedade.

25.17 Propriedades de um JavaBean, métodos e eventos podem ser expostos automaticamente por uma ferramenta de desenvolvimento se o programador seguir os padrões adequados de projeto de JavaBean.

25.18 Cada classe **`BeanInfo`** deve implementar a interface **`BeanInfo`**. Essa interface descreve os métodos utilizados por uma ferramenta de desenvolvimento para determinar os recursos de um *bean* descrito pela classe **`BeanInfo`** desse *bean*.

25.19 Uma classe **`BeanInfo`** pode ser utilizada para descrever um JavaBean para uma ferramenta de desenvolvimento, de modo que a ferramenta possa apresentar os recursos do *bean* para um programador. Isso é útil para JavaBeans que não seguem os padrões de projeto do JavaBean ou JavaBeans em que o programador quer personalizar o conjunto exposto de propriedades, métodos e eventos.

25.20 Por default, a classe **`BeanInfo`** tem o mesmo nome que o *bean* e termina com **`BeanInfo`**.

25.21 Por default, a classe **`BeanInfo`** é incluída no mesmo arquivo JAR que o JavaBean **`SliderFieldPanel`**. Quando o *bean* é carregado, a **BeanBox** (ou ferramenta de desenvolvimento) determina se o arquivo JAR contém uma classe **`BeanInfo`** para um *bean*. Se uma classe **`BeanInfo`** é localizada, ela é utilizada para determinar os recursos expostos do *bean*. Caso contrário, a introspecção-padrão é utilizada para determinar os recursos expostos do *bean*.

25.22 Por default, a classe **`BeanInfo`** é colocada no mesmo **`package`** que o *bean* que ela descreve.

25.23 A classe **`SimpleBeanInfo`** fornece uma implementação-padrão de cada método na interface **`BeanInfo`**. O programador pode redefinir seletivamente os métodos dessa classe para implementar uma classe **`BeanInfo`** adequada.

25.24 Se os métodos *set*/*get* para uma propriedade não utilizam a convenção para atribuição de nomes do JavaBeans para propriedades, há dois outros construtores **`PropertyDescriptor`** em que os nomes reais de método são passados. Isso permite às ferramentas de desenvolvimento utilizar métodos de propriedade não-padrão a fim de expor uma propriedade para manipulação pelo programador em tempo de projeto. Isso é particularmente útil no ajustamento de uma classe como um JavaBean quando essa classe não foi originalmente projetada e implementada utilizando padrões de projeto JavaBeans.

25.25 Ao utilizar padrões de projeto JavaBeans, o nome do conjunto de eventos é parte de todos os nomes de tipo de dados e nomes de método utilizados para processar o evento. Por exemplo, os tipos e métodos para o conjunto de eventos **`propertyChange`** são: **`PropertyChangeListener`** (a interface que um objeto deve implementar para ser notificado de um evento nesse conjunto de eventos), **`PropertyChangeEvent`** (o tipo passado para um método ouvinte de um evento nesse conjunto de eventos), **`addPropertyChangeListener`** (o método chamado para adicionar um ouvinte de um evento nesse conjunto de eventos), **`removePropertyChangeListener`** (o método chamado para remover um ouvinte de um evento nesse conjunto de eventos) e **`firePropertyChange`** (o método chamado para notificar ouvintes quando um evento nesse conjunto de eventos ocorre — esse método é identificado dessa maneira apenas por convenção).

25.26 **`EventSetDescriptor`**s podem ser construídos com outros argumentos para expor eventos que não seguem os padrões de projeto do JavaBeans convencionais.

Dica de teste e de depuração

25.1 A pode ser utilizada para testar e depurar JavaBeans.

Exercícios de auto-revisão

25.1 Preencha as lacunas em cada uma das frases seguintes:

a) As quatro janelas da **BeanBox** são _____, _____, _____ e _____.
b) Um _____ permite que o programador personalize o valor de uma propriedade.
c) O item de menu _____ do menu **Edit** da janela **BeanBox** fornece acesso aos eventos suportados por um *bean* que é uma origem de evento.
d) A _____ é utilizada para especificar o alvo de um evento.
e) A propriedade _____ do *tag* **`<applet>`** especifica uma lista separada por vírgulas de arquivos JAR contendo o código que é utilizado para executar o *applet*.
f) Todos os JavaBeans devem implementar a interface _____ para poderem ser salvos a partir de uma ferramenta de desenvolvimento depois de personalizados pelo programador.
g) Todos os _____ registrados são automaticamente notificados quando o valor de uma propriedade associada se altera.
h) Quando um *bean* fornece propriedades associadas, haverá um item de menu _____ no menu **Edit** da **BeanBox**.
i) Uma ferramenta de desenvolvimento utiliza _____ para expor propriedades, métodos e eventos de um JavaBean.
j) Para JavaBeans que não seguem o padrão de projeto do JavaBean ou JavaBeans em que o programador quer personalizar o conjunto de propriedades, métodos e eventos expostos, o programador pode fornecer uma classe _____ que descreve para a ferramenta de desenvolvimento como apresentar os recursos do *bean*.
k) Um objeto _____ descreve uma propriedade que uma ferramenta de desenvolvimento deve expor.
l) Um objeto _____ descreve um evento que uma ferramenta de desenvolvimento deve expor.

Respostas dos exercícios de auto-revisão

25.1 a)**Toolbox**, **BeanBox**, **Properties**, **Method Tracer**. b) editor de propriedades. c) **Events**. d) linha de seletor de alvo. e) **`archive`**. f) **`Serializable`**. g) **`PropertyChangeListener`**s. h) **Bind property...**. i) introspecção. j) **`BeanInfo`**. k) **`PropertyDescriptor`**. l) **`EventSetDescriptor`**.

Exercícios

25.2 Experimente cada um dos JavaBeans de demonstração fornecidos com a **BeanBox**. A documentação que vem com o BDK fornece uma visão geral dos *beans* de demonstração. Ao utilizar cada *bean*, experimente o seguinte:

a) Inspecione as propriedades de cada *bean* e experimente modificá-los.
b) Inspecione os eventos suportados por cada *bean* e experimente utilizar esses eventos para enganchar vários *beans* de demonstração uns aos outros.

25.3 Utilizando o **SliderFieldPanel** como base, crie seu próprio componente GUI **ColorSelector** contendo três instâncias de nosso *bean* **SliderFieldPanel**. Todos devem ter valores no intervalo 0 a 255 para os componentes de azul, verde e vermelho de uma cor. O programador que utilizar seu *bean* deve ser capaz de configurar a cor padrão enquanto seu aplicativo executa. Torne **color** uma propriedade associada de modo que outros objetos possam ser notificados quando a cor se alterar. Teste seu *bean* na **BeanBox** alterando a cor de fundo da **BeanBox** quando uma nova cor for selecionada. Para especificar a **BeanBox** como o alvo de um evento, simplesmente aponte a linha seletora de alvo para o fundo da janela da **BeanBox** e clique.

25.4 Modifique o Exercício 25.3 para fornecer um mecanismo para visualizar a cor selecionada. Para esse propósito adicione um objeto **JPanel** ao *bean*. Teste seu *bean* na **BeanBox** alterando a cor de fundo da **BeanBox** quando uma nova cor for selecionada.

25.5 Crie uma classe **BeanInfo** para o *bean* **LogoAnimator2** que expõe somente as propriedades **background** e **animationDelay**. Teste seu *bean* na **BeanBox**.

25.6 Utilizando as técnicas gráficas discutidas no Capítulo 11, implemente sua própria subclasse de **JPanel** que suporta desenhar diversas de formas. O usuário deve ser capaz de selecionar qual forma desenhar, e então utilizar o mouse para desenhar a forma. Todas as formas devem ser mantidas como objetos em uma hierarquia de tipos de forma. Você pode utilizar as hierarquias descritas nos Exercícios 9.28 e 9.29 ou utilizar as classes predefinidas da API Java2D como discutimos no Capítulo 11. Armazene todos os objetos de forma em um **Vector**. Empacote sua classe como um JavaBean e teste-o na **BeanBox**.

25.7 Modifique sua solução do Exercício 25.6 para permitir que o usuário selecione a cor e as características de preenchimento da forma. Empacote sua classe como um JavaBean e teste na **BeanBox**.

25.8 Adicione um método **saveShapes** e um método **loadShapes** ao JavaBean no Exercício 25.7. O método **saveShapes** deve armazenar um **Vector** de objetos em um arquivo em disco e o método **loadShapes** deve carregar um **Vector** de objetos a partir de um arquivo em disco. Nenhum dos métodos devem receber nenhum argumentos — o **Vector** deve ser parte do *bean*. O arquivo deve ser manipulado com as técnicas de serialização de objeto discutidas no Capítulo 17. Quando o método **saveShapes** for chamado, exiba um diálogo **JFileChooser** que permita que o usuário selecione um arquivo em que salvar o conteúdo do **Vector**, então grave o **Vector** no arquivo. Quando o método **loadShapes** for chamado, exiba um diálogo **JFileChooser** que permite que o usuário selecione um arquivo para ler o conteúdo do **Vector**, carregue então o **Vector**. Empacote sua classe como um JavaBean e teste-o na **BeanBox**. Enganche um botão em seu *bean* de modo que, quando o botão for pressionado, o método **saveShapes** é chamado. Enganche um segundo botão em seu *bean* de modo que, quando o botão for pressionado, o método **loadShapes** é chamado.