Construção de componentes: JavaBeans

- Em Java, Bean significa Componente
- Tem dois modelos de componentes em Java:
  - JavaBeans normais para componentes baseados em eventos
    - Frequentemente (mas não necessariamente) feitos para GUI
  - Enterprise JavaBeans (EJB) para Server Components
- Não há relação entre os dois modelos
- Aqui, usaremos:
  - "JavaBeans" (ou "Bean") para significar JavaBeans normais
  - "EJB" para significar Enterprise JavaBeans
- Esta seção fala de JavaBeans
  - Usaremos o Bean Development Kit (BDK) para construir aplicações
  - Podemos também usar o Bean Builder, mais recente que o BDK, mas tão cheio de bugs quanto o BDK
  - Se quiser fuçar o Bean Builder, veja o diretório tvexample

Introdução a JavaBeans

- O modelo de componentes JavaBeans foi bolado para permitir que componentes reutilizáveis pudessem ser compostos em outros JavaBeans, applets e componentes usando ferramentas visuais
  - Embora seja possível fazer Beans não visuais, a *composição* é feita visualmente
- JavaBeans apareceu no JDK1.1
- Conceitos básicos
  - Uma ferramenta visual descobre as propriedades,

métodos e eventos de um Bean usando introspeção (*introspection*, ou "reflexão em componentes Bean")

- Introspeção: olhar dentro do Bean
- Dizemos que as propriedades, métodos e eventos são *expostos* pelo Bean

- Há duas formas de usar introspeção para descobrir essas coisas:
  - Usando a API *Reflection* (que permite descobrir todos os métodos, atributos, etc. de uma classe) e usando nomes padronizados de métodos para carregar a semântica desejada
  - Detalhando a informação de propriedades, métodos e eventos numa classe especial BeanInfo (Bean Information)
- Propriedades correspondem a atributos de aparência e de comportamento que podem ser mudados em tempo de Design
  - Propriedades são expostas pelos Beans para que possam ser customizadas em tempo de design
  - A customização pode usar:
    - Editores de propriedades
    - Customizadores de Beans (são *wizards* de customização mais sofisticados)
- Beans usam eventos para se comunicarem com outros Beans
  - Um Bean que quer receber eventos (Listener Bean) cadastra seu interesse junto ao Bean que dispara o evento (Source Bean)
  - As ferramentas visuais podem examinar o Bean e verificar quais eventos podem ser disparados (enviados) e quais eventos podem ser tratados (recebidos) pelo Bean
- A persistência permite que um Bean salve seu estado e o restaure adiante
  - JavaBeans usa *Object Serialization* para implementar a persistência

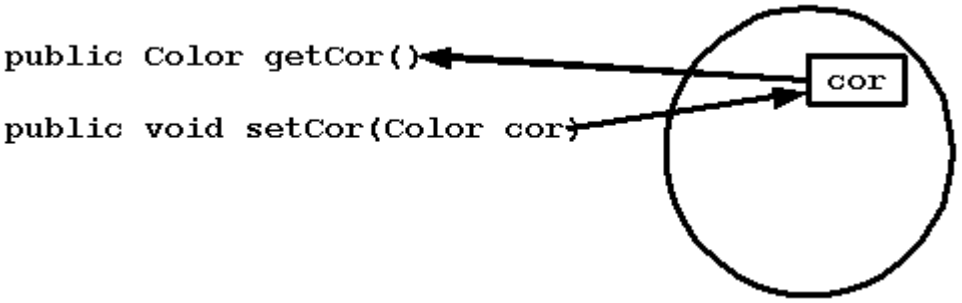
- Os métodos de um Bean não são diferentes de outros métodos em Java
  - Todos os métodos públicos são exportados e podem ser chamados por outros Beans
- Criar e manipular Beans é muito simples e pode ser feito por programadores humanos ou por ferramentas de design

Conceitos básicos de JavaBeans

Propriedades

- Simple Properties representam tipos de dados simples
  - Podem ser nativos (int, String, ...) ou não
  - Métodos devem se chamar get<NomePropriedade> e set<NomePropriedade>

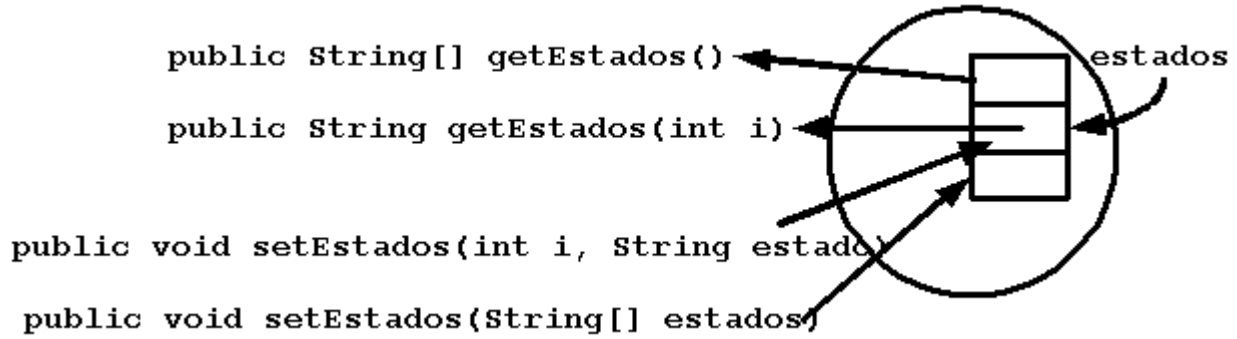
```
public Color getCor();
public void setCor(Color cor);
```



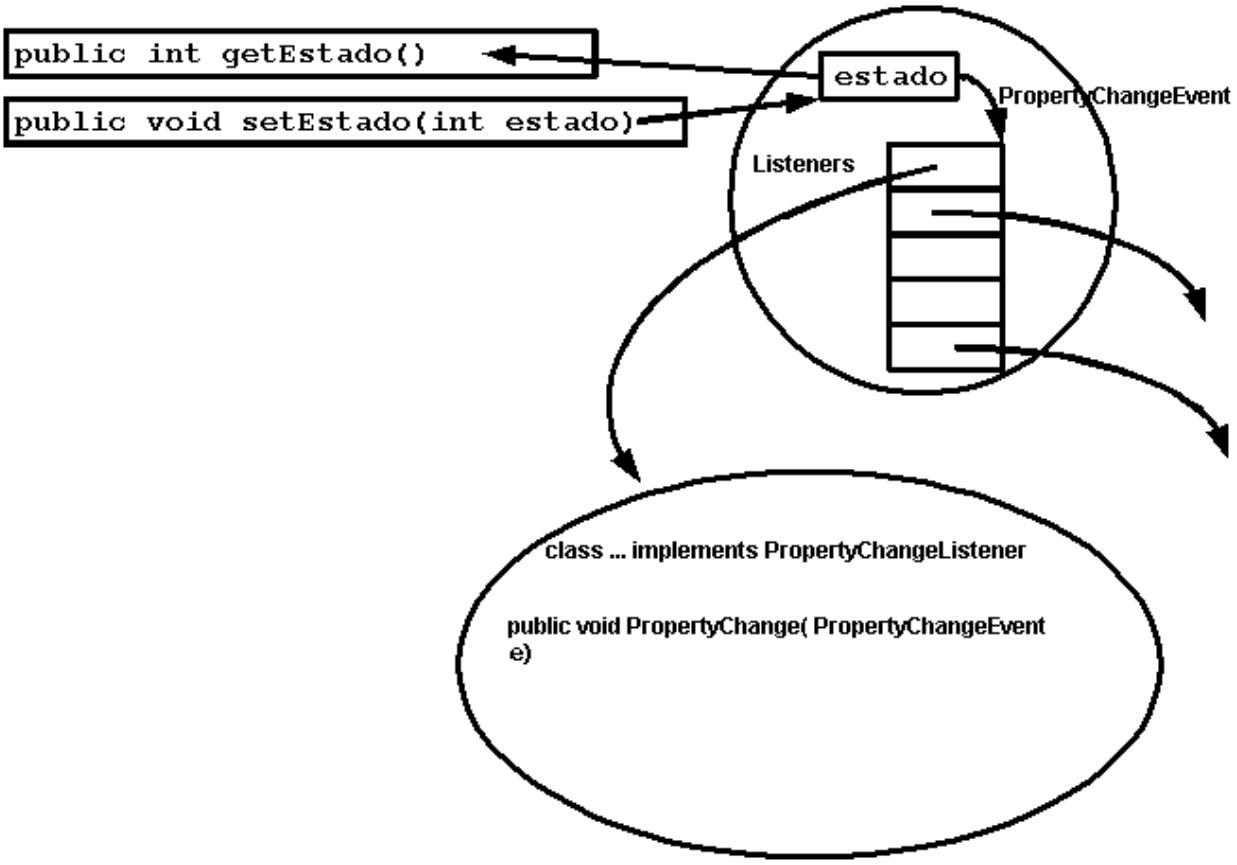
  - Desta forma, a ferramenta visual infere que existe uma propriedade chamada "cor" que pode ser lida (get) e alterada (set)
- Indexed Properties contêm um array de valores possíveis
  - Métodos getter e setter são como para Simple Properties mas manipulam um array, ou recebem um índice a mais

```
public String[] getEstados()
public String getEstados(int índice)
public void setEstados(String[] estados)
```

```
public void setEstados(int índice, String estado)
```

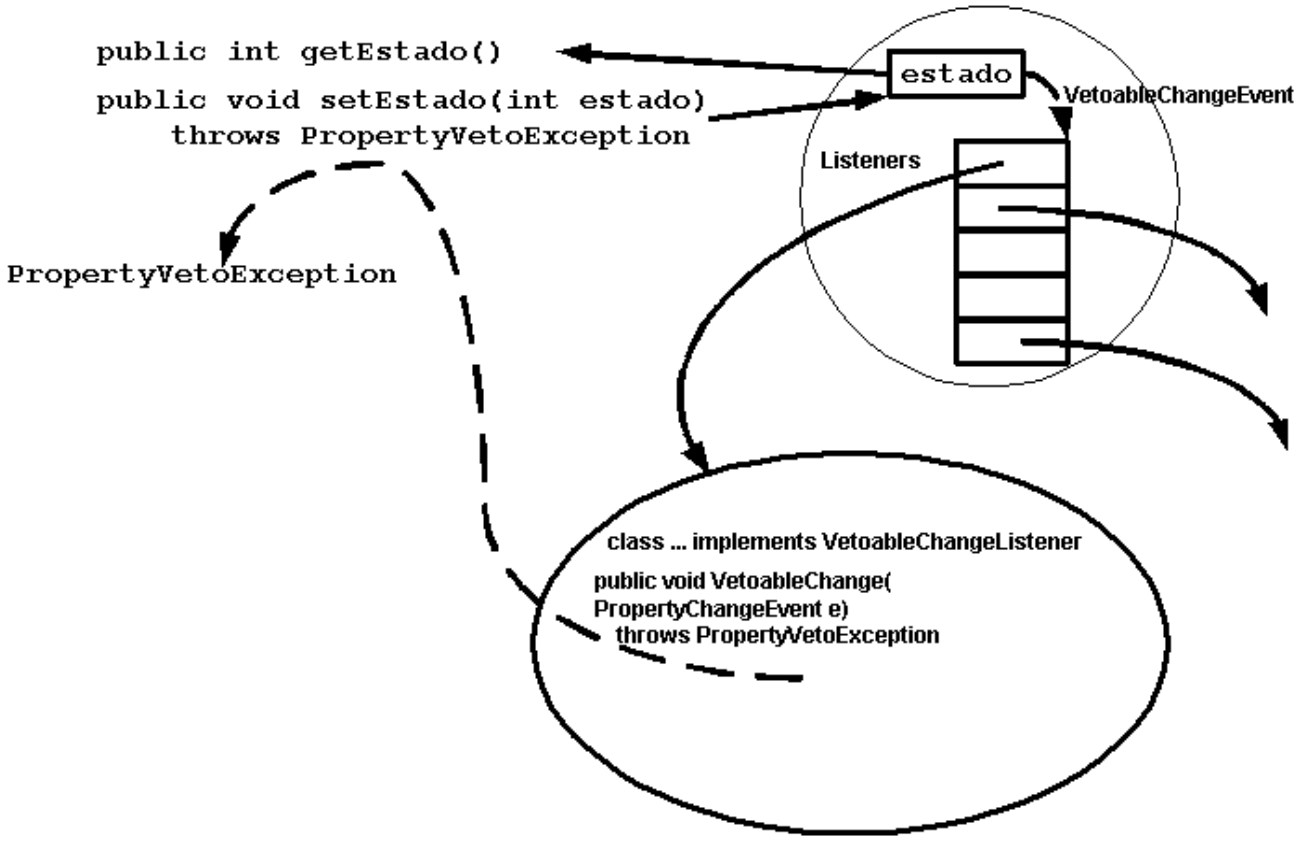


- Bound Properties (Propriedades Amarradas) avisam outros objetos (Beans, por exemplo) de mudanças nos seus valores
  - Uma mudança de valor gera um evento PropertyChangeEvent
  - Listeners deste evento serão notificados da mudança



- Constrained Properties (Propriedades Restritas) são como Bound Properties mas os Listeners podem *vetar* a mudança

- Uma mudança aqui gera um evento VetoableChangeEvent



Eventos

- JavaBeans usam o modelo de Eventos padrão do Java 1.1
  - Ver detalhes aqui
- Resumindo:
  - Quem dispara um evento é um Event Source
  - Um objeto interessado neste evento de cadastra junto ao Source e passa a ser um Listener
  - Todos os listeners são avisados do evento (através de um *callback* feito pelo Source)
  - O evento pode encapsular ou não informação especial para os listeners
  - Uma referência à fonte do evento (o objeto Source) sempre está disponível dentro do evento

O modelo de eventos no JavaBeans

- Os eventos possíveis são:
  - PropertyChangeEvent (para mudanças em Bound Properties)
  - VetoableChangeEvent (para mudanças em Constrained Properties)
- Os Beans Source devem implementar alguns métodos dependendo dos eventos envolvidos
  - Quem gera eventos PropertyChangeEvent deve implementar os seguintes métodos:

```
public void addPropertyChangeListener(
                        PropertyChangeListener pcl);
public void removePropertyChangeListener(
                        PropertyChangeListener pcl);
```

  - Quem gera eventos VetoableChangeEvent deve implementar os seguintes métodos:

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

  - Quem gera eventos customizados deve implementar os seguintes métodos

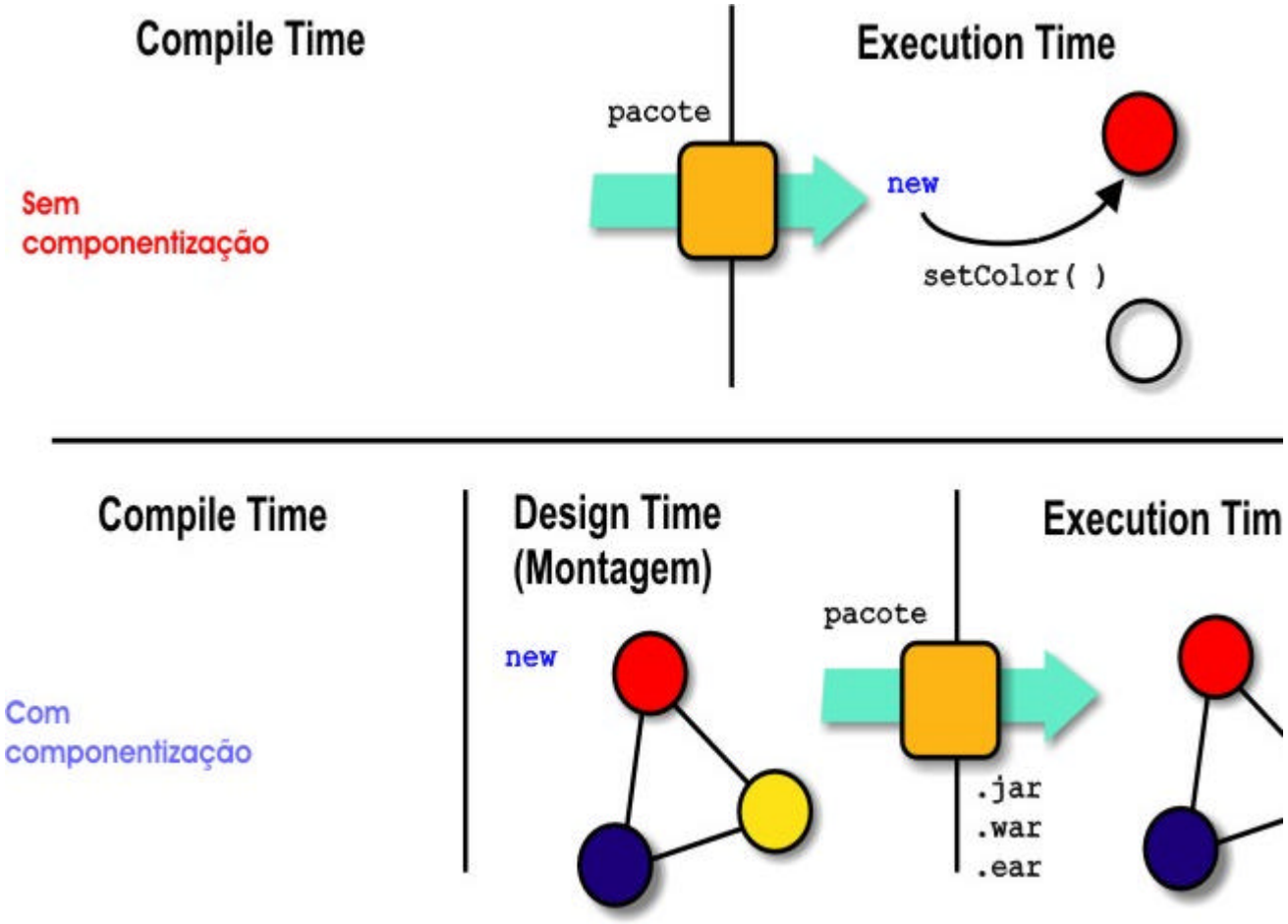
```
public void add<Custom>Listener(<Custom>Listener cl);
public void remove<Custom>Listener(<Custom>Listener cl);
```

  - Os Beans Listener devem implementar a interface apropriada
    - PropertyChangeListener
      - Deve implementar o método PropertyChange (PropertyChangeEvent e);
    - VetoableChangeListener
      - Deve implementar o método VetoableChange (VetoableChangeEvent e);
    - <Custom>Listener

- Deve implementar o método public Xpto (<CustomEvent> e);

Um Bean mínimo

- Tem um construtor default ("no-arg" constructor)
  - Para poder instanciar um Bean na ferramenta visual
- Implementa a interface Serializable
  - Para poder salvar o Bean customizado em tempo de design
  - Para entender melhor isso, veja a diferença entre a forma de instanciar objetos sem e com componentização (ver figura abaixo)
    - Quando usamos componentização, objetos podem ser instanciados em tempo de design (usando componentes como factories) e seus atributos podem ser alterados em tempo de design
    - É necessário ter serialização para poder obter os objetos novamente em tempo de execução



Classes especiais para ajudar

- PropertyChangeSupport e VetoableChangeSupport
  - Ajudam a mandar os eventos para todos os listeners e outras tarefas semelhantes

Packaging de JavaBeans

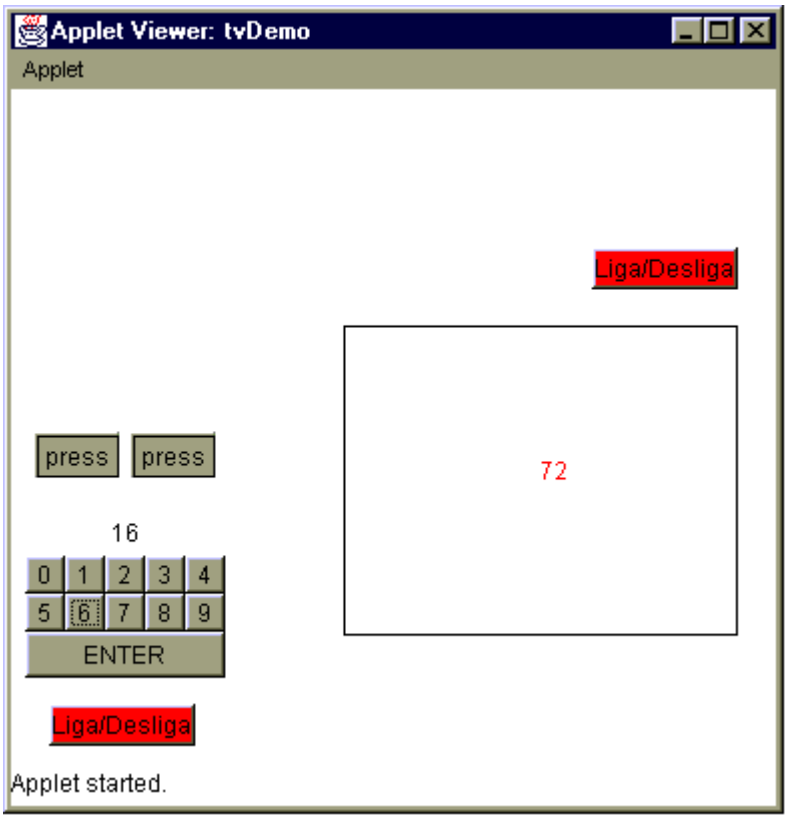
- Usa um arquivo jar
  - Compatível com ZIP
  - O primeiro arquivo tem que ser um "manifest file"
  - Conteúdo típico de um manifest file:

```
Manifest-Version: 1.0
Name: tv/BotaoOnOff.class
Java-Bean: True
Created-By: 1.2 (Sun Microsystems Inc.)

Name: tv/Limites.class
Java-Bean: True
```

```
Name: tv/TVDisplay.class
Java-Bean: True

Name: tv/KeyPad.class
Java-Bean: True

Name: tv/TVEstado.class
Java-Bean: True
```

Uma Demonstração

- Veremos os detalhes de JavaBeans através de um exemplo
  - Controle Remoto de uma TV
  - Demo: Clique aqui
  - Se a demo acima não funcionar (devido a um browser antigo), faça o seguinte:
    - Vá para o diretório tv/tvDemoApplet
    - Digite: appletviewer tvDemoApplet.html



O design e código do exemplo Demo

Elementos básicos

- Para a TV
  - Um botão liga/desliga com luzinha mostrando o estado on/off
  - Botões Up/Down de mudança de canal
    - Não modelados aqui para simplificar
  - Uma tela
  - Um display do número de canal
- Para o controle remoto
  - Um botão liga/desliga com luzinha mostrando o estado on/off
  - Botões Up/Down de mudança de canal
  - Uma ilha numérica (keypad)
- Poderíamos adicionar controle de volume, etc. mas isso foi deixado como exercício

Design de Beans úteis para resolver o problema

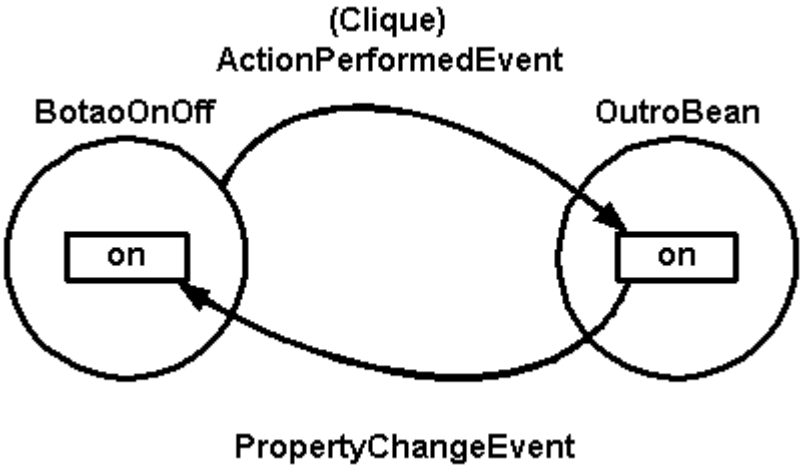
- Vamos bolar Beans úteis para muitas situações e não só para atender a este exemplo
  - Este é o maior problema da solução original apresentada na referência "Java Application Frameworks" de Govoni
  - Govoni fornece Beans pouco genéricos

O Bean BotaoOnOff

- Tem uma propriedade "on"
  - Tipo boolean
  - Ao set alterada com setOn(boolean estado), a luzinha muda de cor
  - É uma propriedade simples
    - Não avisa ninguém quando muda de valor
  - Nossa implementação pode até deixá-la como propriedade Bound para ter mais flexibilidade no

reuso em outras situações (onde haja listeners)

- É listener de outras propriedades "on"
  - Ao receber um PropertyChangeEvent, chama setOn ()
- Observar que clicar no botão *não* chama setOn()
  - Alguém tem que se amarrar ao ActionPerformedEvent do clique de botão (o dispositivo sendo ligado/desligado), mudar uma propriedade "on" particular da qual este botão seria listener
  - Assim, temos certeza que o estado "on" do botão reflete o estado de algum dispositivo sendo controlado
  - Também permite que mais de um botão controle o dispositivo e que cada botão saiba o estado real do dispositivo



- Código do Bean BotaoOnOff segue:
  - Observe que deixamos a propriedade "on" bound mesmo não precisando neste exemplo (generalidade)

```
package tv; // poderia estar num package mais genérico

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

/**
 * Um botão que acende quando está "on"
```

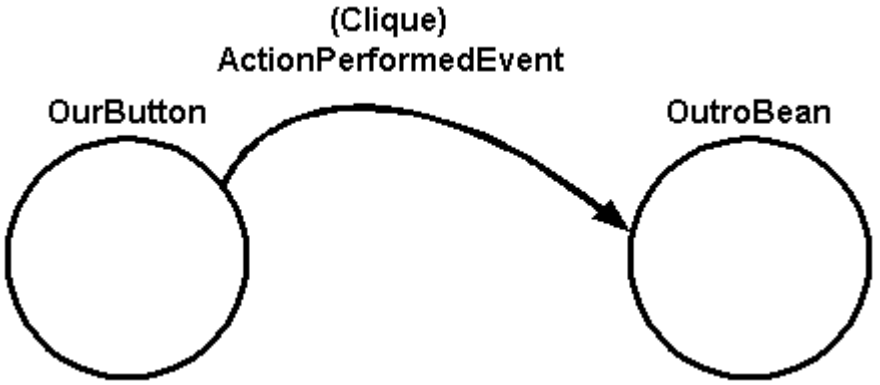
```
 */
public class BotaoOnOff extends Button
             implements Serializable, PropertyChangeListener

  private boolean   on; // estado do botão
  private PropertyChangeSupport pcs;

  // Construtor
  public BotaoOnOff() {
    // Cada bean tem um construtor default ("no-arg")
    super("Liga/Desliga");
    pcs = new PropertyChangeSupport(this);
    on = false;
  }

  // Método getter para propriedade "on"
  public boolean isOn() {
    return on;
  }

  // Método setter para propriedade "on"
  public void setOn(boolean on) {
    // Dispara a mudança para os listeners, se houver
    pcs.firePropertyChange("on", new Boolean(this.on),
                                 new Boolean(on));
    this.on = on;
    mudaVisual();
  }

  // Muda o visual da "lâmpada on/off"
  private void mudaVisual() {
    setBackground(isOn() ? Color.red : Color.gray);
  }

  // Se este bean for listener da propriedade "on" de outro
  // entao precisamos deste método
  public void propertyChange(PropertyChangeEvent pce) {
    String propriedadeMudada = pce.getPropertyName();
    Object novoValor = pce.getNewValue();
    if(propriedadeMudada.equals("on")) {
      this.setOn(((Boolean)novoValor).booleanValue());
    }
  }

  // Método de cadastro de listeners
  // usa delegação para implementar
  public void addPropertyChangeListener(PropertyChangeListen
    pcs.addPropertyChangeListener(l);
```

```
  }

  // Método de remoção de listeners
  // usa delegação para implementar
  public void removePropertyChangeListener(PropertyChangeLis
    pcs.removePropertyChangeListener(l);
  }
}
```

- Exercício: Para deixar o botão mais genérico ainda, poderíamos permitir que um clique do botão mudasse diretamente a propriedade "on". Forneça um bom design para isso. Lembre que este comportamento é apenas uma possibilidade e não deve ser obrigatório. Em que situação seria útil usar essa funcionalidade?

O botão comum

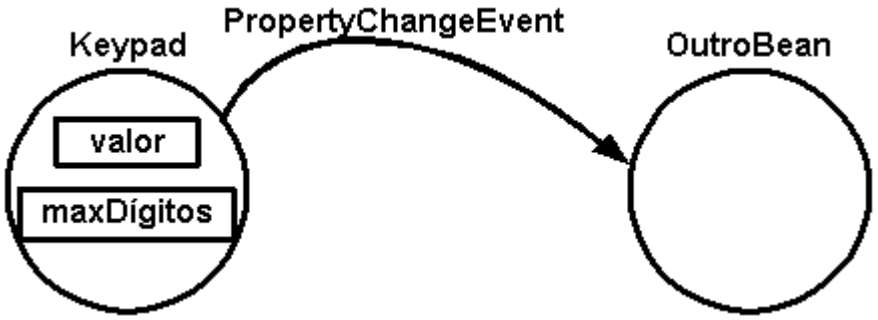
- Gera o evento ActionPerfomedEvent quando clicado
- Não tem propriedade
- Usaremos o botão OurButton já fornecido no Beans Development Kit (BDK)



O bean KeyPad

- Tem 10 botões numéricos
  - O próprio Bean atende aos cliques desses botões e não os repassa para fora
- Tem uma área para mostrar o valor temporário sendo digitado
- Tem um botão ENTER
  - Para mudar a propriedade "valor" que recebe o valor temporário

- A propriedade "valor" é bound para avisar outros Beans da mudança
- Tem uma propriedade simples inteira informando o número máximo de dígitos que o keypad manipula
  - Setado em tempo de design
  - Default: 3



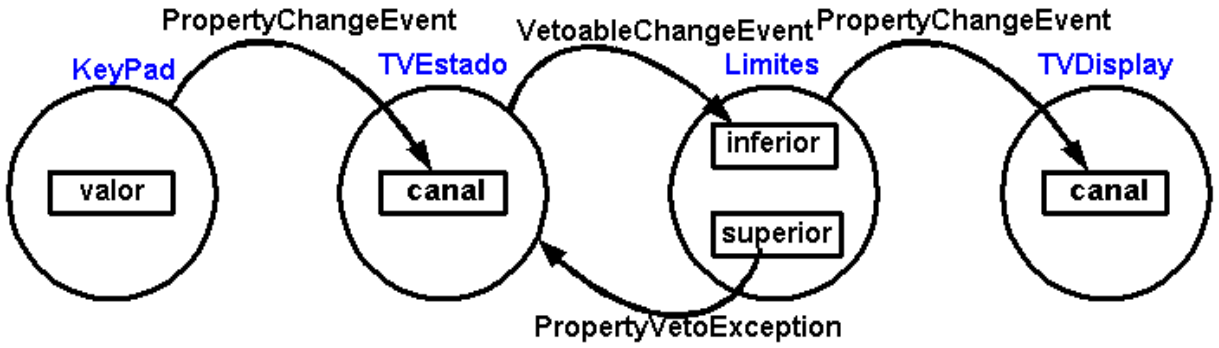
- Código do Bean KeyPad segue:

```
package tv;

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

/**
 * Um keypad
 */
public class KeyPad extends Container implements ActionListe
  private int   valor = 0;      // propriedade do bean
  private int   maxDígitos = 3; // propriedade: número max d
  private String  valorDisplay;   // valor do display antes
  private PropertyChangeSupport pcs;
  private Button botãoEnter = new Button("ENTER");
  private Label display = new Label("");

  // Construtor
  public KeyPad() {
    pcs = new PropertyChangeSupport(this);
    Panel p = new Panel(); // para os botões de 0 a 9
    p.setLayout(new GridLayout(2,5));
    setLayout(new BorderLayout()); // para conter o keypad i

    // cria os botões
    for(int i = 0; i < 10; i++) {
      Button b = new Button(Integer.toString(i));
```

```
    p.add(b);
    b.addActionListener(this);
  }
  botãoEnter.addActionListener(this);
  add("South", botãoEnter);
  add("Center", p);
  add("North", display);
  display.setAlignment(Label.CENTER);
  valorDisplay = "";
}

// getter do valor
public int getValor() {
  return valor;
}

// setter do valor
public void setValor(int valor) {
  pcs.firePropertyChange("valor", new Integer(this.valor),
                                  new Integer(valor));
  this.valor = valor;
  valorDisplay = "";
  display.setText(valorDisplay);
}

// getter de maxDígitos
public int getMaxDígitos() {
  return maxDígitos;
}

// setter de maxDígitos
public void setMaxDígitos(int maxDígitos) {
  this.maxDígitos = maxDígitos;
}

// Mudança do display
public void mudaDisplay(String digito) {
  valorDisplay += digito;
  // só permite maxDígitos digitos de display
  if(valorDisplay.length() > maxDígitos) {
    valorDisplay = digito;
  }
  display.setText(valorDisplay);
}

// cadastro para mudança de propriedade
public void addPropertyChangeListener(PropertyChangeListen
  pcs.addPropertyChangeListener(l);
```

```
  }

  // remoção de cadastro para mudança de propriedade
  public void removePropertyChangeListener(PropertyChangeLis
    pcs.removePropertyChangeListener(l);
  }

  // trata o evento de clique de qualquer botão
  public void actionPerformed(ActionEvent ae) {
    if(ae.getSource().equals(botãoEnter)) {
      try {
        setValor(Integer.parseInt(valorDisplay));
      } catch(NumberFormatException nfe) {
        // "não deveria" acontecer
        nfe.printStackTrace();
        return;
      }
    } else {
      mudaDisplay(ae.getActionCommand());
    }
  }
}
```

- Separamos a TV (fora os botões) em 3 partes
  - Tem design mais simples para resolver o problema mas fizemos assim para:
    - Dividir a parte gráfica (TVDisplay) do miolo da TV (TVEstado)
    - Para poder demonstrar Constrained Properties
      - A mudança de canal pode ser vetada se estiver fora de faixa (digamos de 1 a 99)
      - Usamos um Bean Limites para isso



- Exercício: Mudar o design acima de forma ao Bean Limites se comunicar apenas com TVEstado e de

forma a TVDisplay receber o evento de TVEstado

O Bean Limites

- Recebe um VetoableChangeEvent e veta se o valor não estiver nos limites apropriados
- Os limites inferior e superior são propriedades simples
- Ao aceitar um valor, a propriedade bound "valor" é alterada e os listeners são avisados
- O código segue:

```
package tv;

import java.beans.*;
import java.io.*;

/**
 * Verifica se um valor inteiro cai entre dois limites
 * e lança exceção se não aceitável
 */
public class Limites
       implements VetoableChangeListener, Serializable {

  private int inferior; // propriedade: limite inferior para
  private int superior; // propriedade: limite inferior para
  private int valor;    // propriedade: o valor do Bean
  private PropertyChangeSupport pcs;

  public Limites() {
    pcs = new PropertyChangeSupport(this);
    inferior = 1;   // valores default
    superior = 99;
  }

  // getter de inferior
  public int getInferior() {
    return inferior;
  }

  // setter de inferior
  public void setInferior(int inferior) {
    this.inferior = inferior;
  }

  // getter de superior
  public int getSuperior() {
```

```
  return superior;
}

// setter de superior
public void setSuperior(int superior) {
  this.superior = superior;
}

// verifica validade do valor
public boolean isValorValido(int valor) {
  return inferior <= valor && valor <= superior;
}

// getter de valor
public int getValor() {
  return valor;
}

public void setValor(int valor) {
  pcs.firePropertyChange("valor", new Integer(this.valor),
                                  new Integer(valor));
  this.valor = valor;
}

// vetoableChange - é aqui que recebemos o evento
// de mudança de valor e verificamos se é aceitável
public void vetoableChange(PropertyChangeEvent pce)
            throws PropertyVetoException {

  String propriedadeMudada = pce.getPropertyName();
  Object novoValor = pce.getNewValue();
  if(propriedadeMudada.equals("valor")) {
    int valorProposto = ((Integer)novoValor).intValue();
    if(isValorValido(valorProposto)) {
      setValor(valorProposto);
    } else {
      throw new PropertyVetoException(
                    "Valor fora dos limites", pce);
    }
  }
}

// cadastro para mudança de propriedade
public void addPropertyChangeListener(PropertyChangeListen
  pcs.addPropertyChangeListener(l);
}

// remoção de cadastro para mudança de propriedade
```

```
  public void removePropertyChangeListener(PropertyChangeLis
    pcs.removePropertyChangeListener(l);
  }
}
```

O Bean TVDisplay

- Trata da parte gráfica da TV
  - Liga e desliga a tela
  - Mostra o canal selecionado
- Propriedades simples "on" e "canal"
  - Podem ser bound para ter generalidade (uso futuro)
- O código segue:

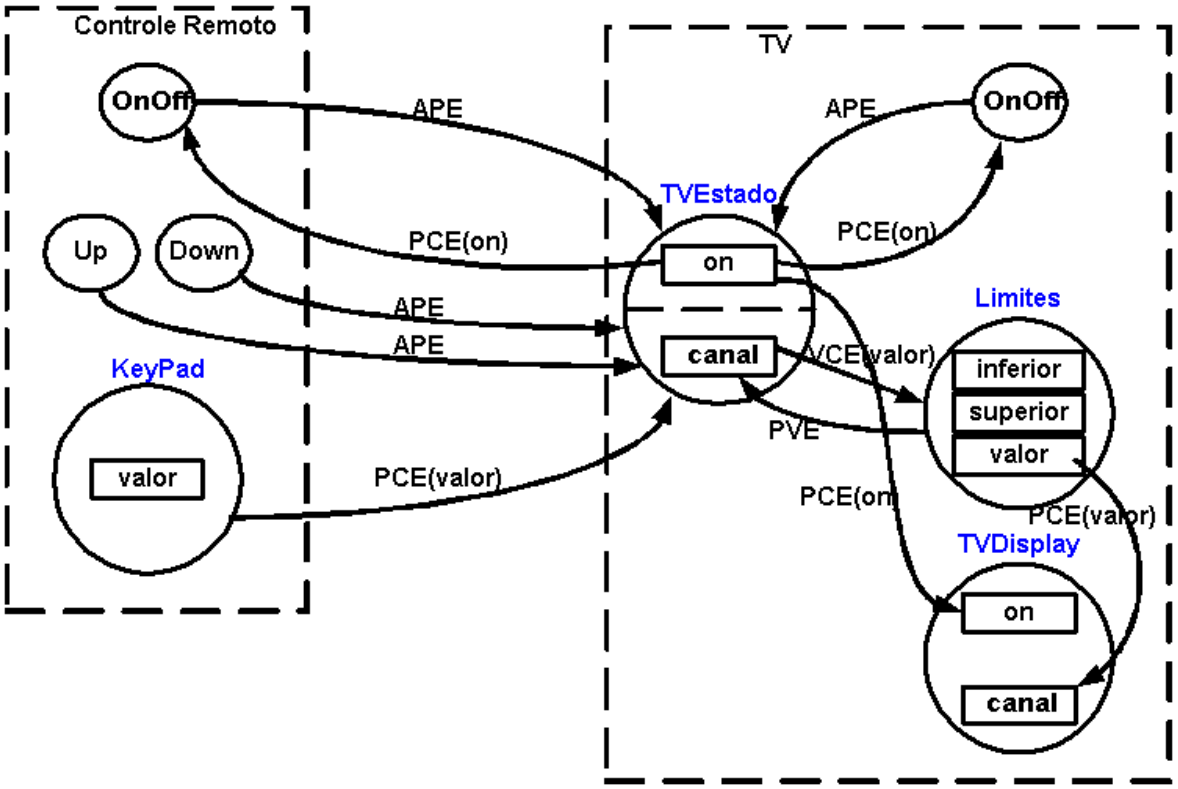
```
package tv;

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

/**
 * Um display de TV simples
 */
public class TVDisplay extends Canvas
             implements PropertyChangeListener, Serializable
  private int       canal;  // propriedade
  private boolean   on;     // propriedade

  private PropertyChangeSupport pcs;

  // Construtor
  public TVDisplay() {
    super();
    pcs = new PropertyChangeSupport(this);
    canal = 0;
    on = false;
  }

  public void paint(Graphics g) {
    int altura = getSize().height;
    int largura = getSize().width;
    int x = 0;
```

```
    int y = 0;
    int offset = 1;
    g.setColor(Color.black);
    g.drawRect(x, y, largura-offset, altura-offset);
    g.setColor(corDaTela());
    g.fillRect(x+1, y+1, largura-offset-1, altura-offset-1);
    if(isOn()) {
      g.setColor(Color.green);
      g.drawString(String.valueOf(canal), largura/2, altura/
    }
  }

  private Color corDaTela() {
    // cor da tela depende do canal sintonizado
    Random nr = new Random(canal);
    return isOn() ? new Color(nr.nextInt(256),
                              nr.nextInt(256),
                              nr.nextInt(256))
                  : Color.gray;
  }

  // getter para "on"
  public boolean isOn() {
    return on;
  }

  // setter para "on"
  public void setOn(boolean on) {
    if(on != this.on) {
      pcs.firePropertyChange("on", new Boolean(this.on),
                                   new Boolean(on));
      this.on = on;
    }
    repaint();
  }

  // getter para o canal
  public int getCanal() {
    return canal;
  }

  // setter para canal
  public void setCanal(int canal) {
    pcs.firePropertyChange("canal", new Integer(this.canal),
                                    new Integer(canal));
    this.canal = canal;
    repaint();
  }
```

```
  // trata da mudança da propriedade canal de outro bean
  public void propertyChange(PropertyChangeEvent pce) {
    String propriedadeMudada = pce.getPropertyName();
    Object novoValor = pce.getNewValue();
    // mudança de "valor" de outro Bean altera o canal
    if(propriedadeMudada.equals("valor")) {
        this.setCanal(((Integer)novoValor).intValue());
    }
    if(propriedadeMudada.equals("on")) {
        this.setOn(((Boolean)novoValor).booleanValue());
    }
  }

  // cadastro para mudança de propriedade
  public void addPropertyChangeListener(PropertyChangeListen
    pcs.addPropertyChangeListener(l);
  }

  // remoção de cadastro para mudança de propriedade
  public void removePropertyChangeListener(PropertyChangeLis
    pcs.removePropertyChangeListener(l);
  }
}
```

O Bean TVEstado

- Propriedade bound "on" com métodos
  - getOn()
  - setOn()
  - onOff() // toggle da propriedade
- Propriedade Constrained "canal" com métodos
  - getCanal()
  - setCanal()
  - incrementaCanal()
  - decrementaCanal()



- O código segue:

```
package tv;

import java.beans.*;
import java.io.*;

/**
 * Manutenção do estado da TV
 */
public class TVEstado implements Serializable {
  private int       canal;  // propriedade: canal da TV
  private boolean   on;     // propriedade: TV ligada ou não
  private PropertyChangeSupport pcs;
  private VetoableChangeSupport vcs;

  // Construtor
  public TVEstado() {
    pcs = new PropertyChangeSupport(this);
    vcs = new VetoableChangeSupport(this);
    canal = 0;
    on = false;
  }

  // getter para propriedade "on"
  public boolean isOn() {
    return on;
```

```
}

// setter para propriedade "on"
public void setOn(boolean on) {
  if(on != this.on) {
    pcs.firePropertyChange("on", new Boolean(this.on),
                                 new Boolean(on));
    this.on = on;
  }
}

// getter para propriedade "canal"
public int getCanal() {
  return canal;
}

// setter para propriedade "canal"
public void setCanal(int canal) throws PropertyVetoExcepti
  vcs.fireVetoableChange("valor", new Integer(this.canal),
                                  new Integer(canal));
  // se for vetado, não chegaremos aqui
  this.canal = canal;
}

// Incrementa canal
public void incrementaCanal() {
  mudaCanal(1);
}

// Decrementa canal
public void decrementaCanal() {
  mudaCanal(-1);
}

private void mudaCanal(int incremento) {
  try {
    setCanal(getCanal() + incremento);
  } catch(PropertyVetoException e) {
    // mensagem de visualização para testes
    System.out.println("Vetado!");
  }
}

// Método de mudança on/off
public void onOff() {
  setOn(!isOn());
}
```

```
  // cadastro para mudança de propriedade
  public void addPropertyChangeListener(PropertyChangeListen
    pcs.addPropertyChangeListener(l);
  }

  // remoção de cadastro para mudança de propriedade
  public void removePropertyChangeListener(PropertyChangeLis
    pcs.removePropertyChangeListener(l);
  }

  // cadastro para mudança "vetável" de propriedade
  public void addVetoableChangeListener(VetoableChangeListen
    vcs.addVetoableChangeListener(l);
  }

  // remoção de cadastro para mudança "vetável" de proprieda
  public void removeVetoableChangeListener(VetoableChangeLis
    vcs.removeVetoableChangeListener(l);
  }
}
```

Como o BDK "cola" os Beans

- Vamos ver um pequeno exemplo com uso de BDK
- Crie um manifest file manifest.txt como mostrado abaixo

```
Manifest-Version: 1.0
Name: tv/BotaoOnOff.class
Java-Bean: True
Created-By: 1.2 (Sun Microsystems Inc.)

Name: tv/Limites.class
Java-Bean: True

Name: tv/TVDisplay.class
Java-Bean: True

Name: tv/KeyPad.class
Java-Bean: True

Name: tv/TVEstado.class
Java-Bean: True
```

- Empacote os Beans num arquivo jar com o comando:

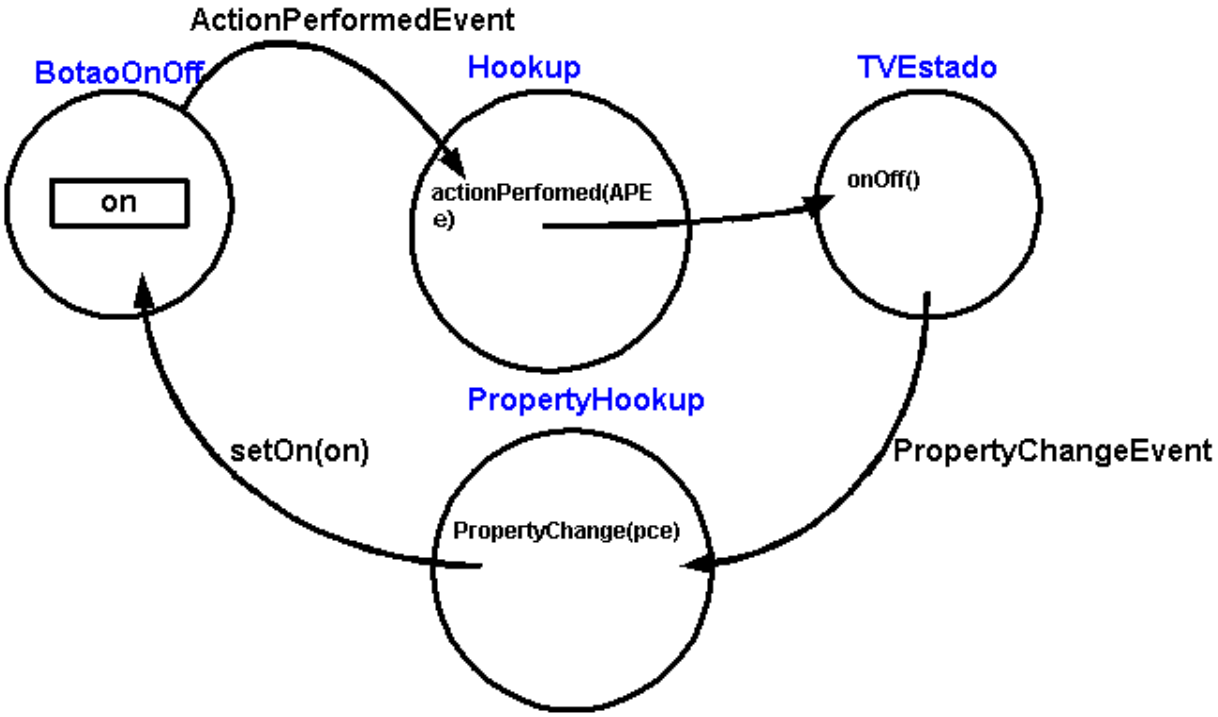
```
jar cfm tv.jar manifest.txt tv/BotaoOnOff.class
        tv/TVEstado.class tv/Limites.class
        tv/KeyPad.class tv/TVDisplay.class
```

- Coloque os Beans (o arquivo tv.jar) no diretório XXX\beans\jars
  - Se o BDK tiver sido instalado em XXX
  - Assim o BDK toma conhecimento dos novos Beans
  - Advertência: devido a um bug do BDK, ao instalar o JDK, não deixar espaço em branco no nome do diretório de instalação do JDK
    - Prometo mudar todo este material para usar o Bean Builder em vez do velho BDK, assim que arrumar um tempinho ...
- Execute o BDK
- Arraste um BotaoOnOff e um TVEstado na área de design
- Ligue o evento ActionPerformedEvent do BotaoOnOff para o método onOff() do Bean TVEstado
- O BDK vai ter criado a seguinte "Adapter class" para colar as coisas:
  - Isto é, para que uma chamada a actionPerformed causada pelo clique chame, na realidade, onOff() de TVEstado

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import tv.TVEstado;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_15e5be4184
        implements java.awt.event.ActionListener,
                   java.io.Serializable {

    public void setTarget(tv.TVEstado t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent a
```

```
        target.onOff();
    }

    private tv.TVEstado target;
}
```

- A classe é instanciada pelo BDK e o setTarget já chamado com parâmetro indicando o Bean TVEstado
- O addActionListener do botão (na realidade de java.awt.Button) é chamado pelo BDK para ter a classe Hookup como Listener do botão
- Agora, no TVEstado, use BindProperty/on e ligue com o Botão/on
  - Neste caso, a classe de hookup utilizada não precisa ser gerada em tempo de design, por que seu código é fixo e já existe dentro do BDK (chama-se PropertyHookup)
  - O PropertyHookup vai receber o evento PropertyChange e vai chamar setOn do Botão
- Claro que uma alternativa adicional seria de não usar "Bind property" no BDK e amarrar o evento PropertyChange de TVEstado ao método setOn do Botão
  - Neste caso, o hookup teria seu código gerado e compilado em tempo de design
  - A vantagem de usar "Bind property" e o PropertyHookup é que esta última classe consegue verificar se a propriedade sendo mudada é a correta e só chamar o destino quando realmente deve
- Tudo está pronto: clique no botão para ver o efeito visual
- Ao salvar os Beans (serializados), tudo está pronto para funcionar:



Construindo a aplicação

- Execute os seguintes passos no BDK:
  - Criação de objetos:

1. Arraste um BotaoOnOff para o controle remoto e altere o label se desejar
2. Arraste dois OurButton para o controle remoto e altere os labels para Up e Down
3. Arraste um KeyPad para o controle remoto e altere o número de dígitos se desejar
4. Arraste um BotaoOnOff para a TV e altere o label se desejar
5. Arraste um TVEstado, TVDisplay e Limites para a TV
6. Altere os limites inferior e superior de Limites se desejar

  - Conexões

1. Liga/Desliga Remoto/ActionPerformed para TVEstado/OnOff (gera hookup)
2. Liga/Desliga TV/ActionPerformed para TVEstado/OnOff (gera hookup)
3. TVEstado/BindProperty/on para Liga/desliga Remoto/on
   (o BDK uso um hookup interno da classe PropertyHookup)
4. TVEstado/BindProperty/on para Liga/desliga TV/on
5. TVEstado/BindProperty/on para TVDisplay/on
6. Up/ActionPerformed para TVEstado/incrementaCanal (gera hookup)
7. Down/ActionPerformed para TVEstado/decrementaCanal (gera hookup)
8. Keypad/BindProperty/valor para TVEstado/canal
9. TVEstado/Event/VetoableChange para Limites/vetoableChange (gera hookup)
10. Limites/BindProperty/valor para TVDisplay/canal

- Agora, manda contruir um applet que faz tudo que pedimos

- Você pode ver o resultado aqui
  - Se não tiver um browser que aceite Java 2, use o appletviewer:

```
appletviewer tvDemoApplet.html
```

- Observe que fizemos o controle remoto e a TV na

mesma aplicação

- Na prática, poderíamos separa os dois usando objetos distribuídos (usando RMI ou CORBA)

Features Avançados de JavaBeans

- Coisas que não usamos para manter a simplicidade

Property Editors

- Para editar propriedades especiais quando o ambiente visual não oferece um editor default ou você não gosta do editor default

Uso da classe BeanInfo

- Para expor apenas os features desejados em vez de depender de reflexão
- Para associar um ícone a um Bean
- Para especificar um Customizer
  - Vide adiante
- Para segregar features em categorias "normal" e "expert"
- Para prover um nome de display mais descritivo ou mais informação acerca de um feature do Bean

Customizadores

- Um Wizard para customizar um Bean complexo
- Um "Wizard" transforma uma GUI onde podemos clicar em qualquer lugar (complexo) numa GUI baseada em perguntas e respostas com muito menos escolhas a cada passo (menos complexo)
  - Parece mais uma receita de bolo para quem não está acostumado com algo

Bridge ActiveX

- Permite converter um Bean num Controle ActiveX