Introdução ao Desenvolvimento de Jogos em Java

Corrigido e ampliado em 20/Fev/2009

Rafael Santos

Organização

Apoio

Mapa

Java e OO	Intro OO	Aplicações	Est. Dados	
Mais Java	GUIs	Componentes	*Layouts*	+ Componentes
	Applets	*Graphics*	*Threads*	
Jogos e Java	Versão 1	Versão 2	Versão 3	Exercícios
Extras	*Showcase*	Empacotando	Omissões	
Referências				

Ementa

- Parte 1: Java e Orientação a Objetos
  - Encapsulamento: classes, instâncias, atributos e métodos.
  - Herança e polimorfismo.
  - Aplicações executáveis.
  - Estruturas de dados.

Java e OO

Ementa

- Parte 2: Mais Java
  - Aplicações gráficas (*applets* e aplicações *desktop*).
  - Componentes de interfaces gráficas e *layout* de tela.
  - Criando componentes.
  - Programação com eventos.
  - Imagens e ícones.
  - *Threads* (linhas de execução).

Mais Java

Ementa

- Parte 3: Jogos em Java: demonstração aplicada de:
  - *Loop* do jogo.
  - *Sprites.*
  - Deteção de Colisão.
  - Criação e remoção de objetos durante o jogo.
- Exercícios.

Jogos e Java

Motivação para este curso

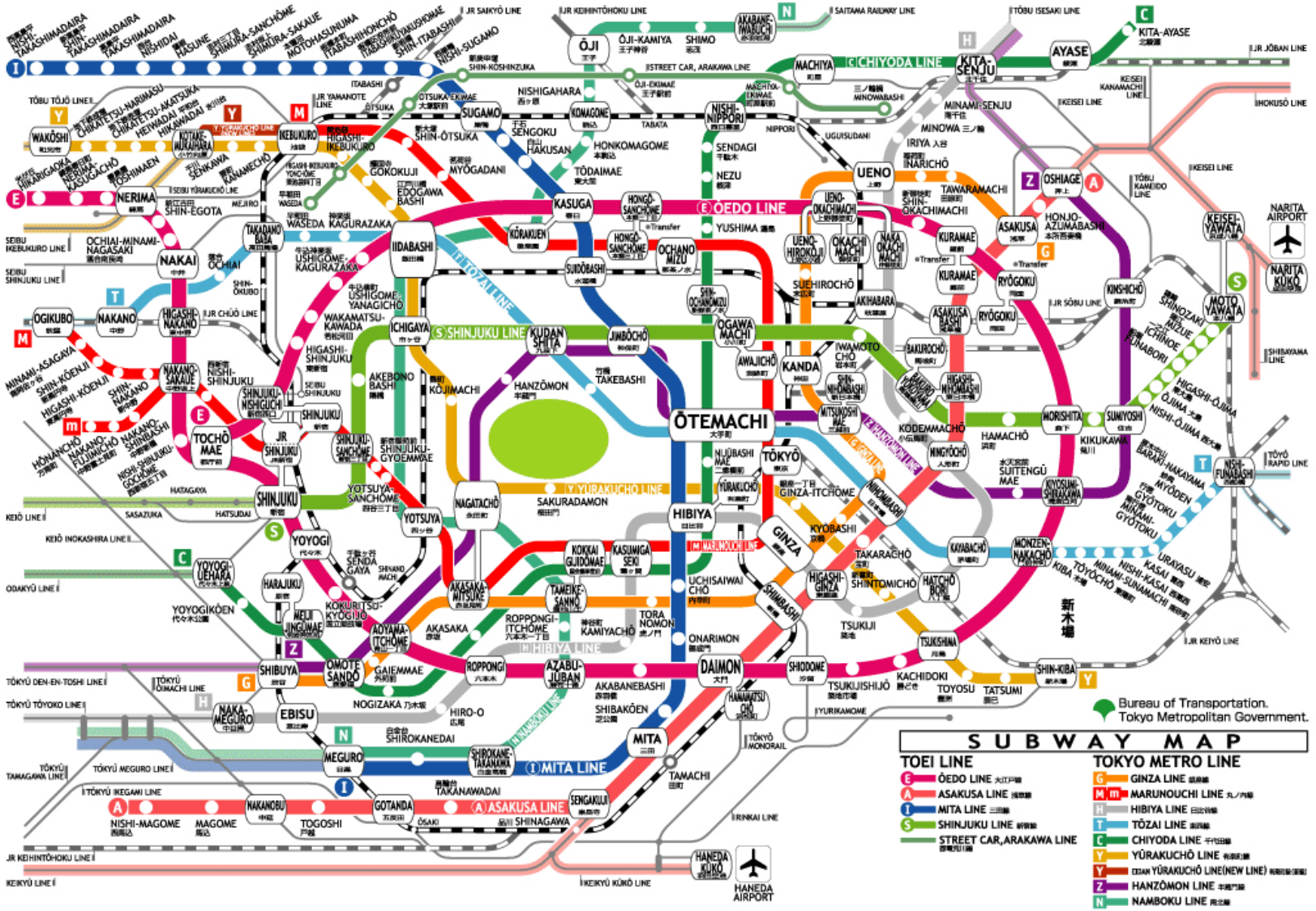
- Motivação para aprender **técnicas de programação**.
  - Java como opção para programação de jogos **simples**.
- Não veremos:
  - *Design* gráfico.
  - 3D.
  - Áudio.
  - *Deployment* e distribuição.
  - Inteligência Artificial.
- É preciso entender a **vasta distância** entre jogar um jogo e escrever um jogo.
  - O mesmo vale para aplicações: problema do ensino de programação.

(des)motivação ?

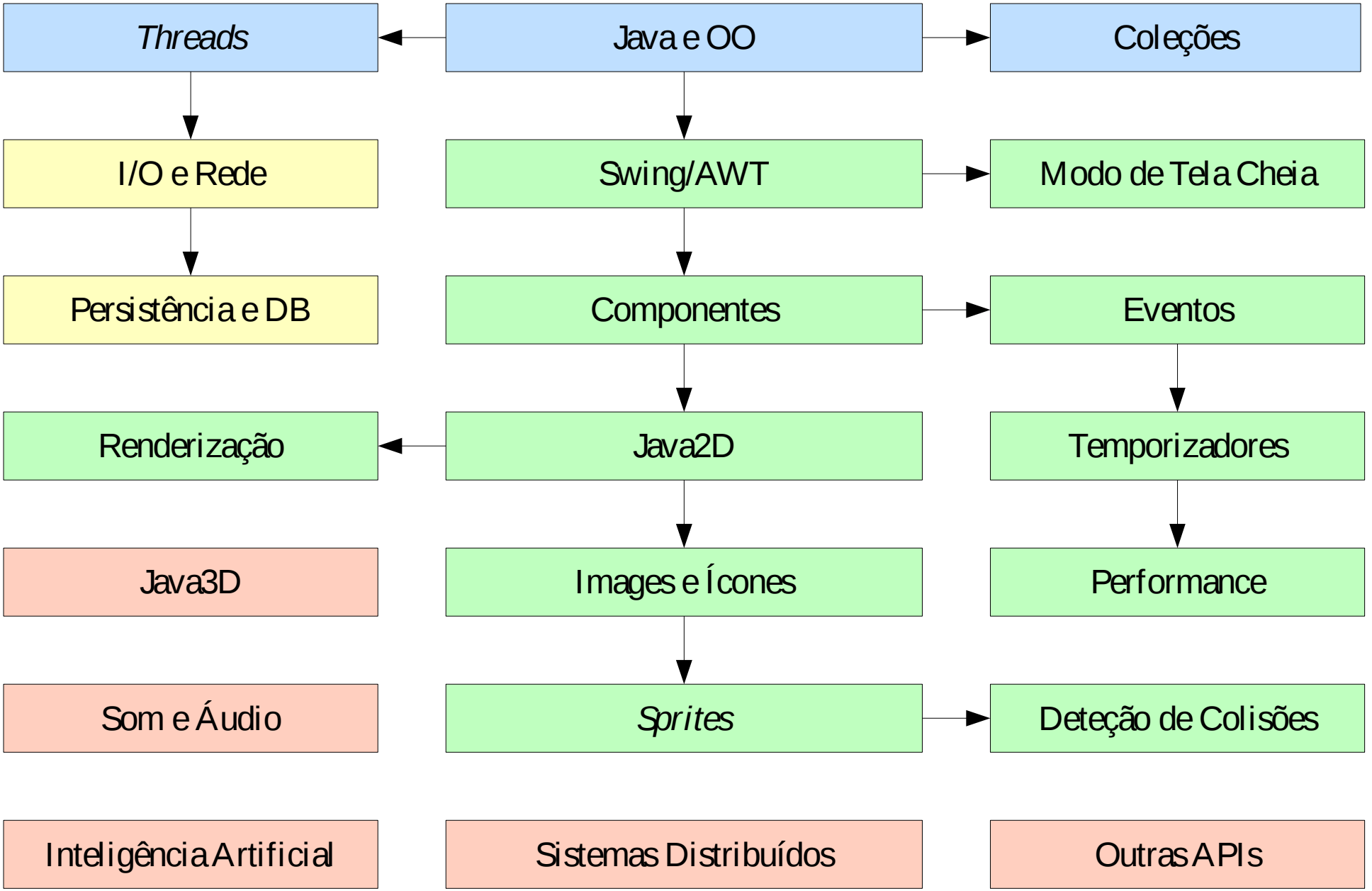
- Dá pra aprender a programar jogos em Java em algumas horas?
- Livros (veja na Amazon e nas referências!):
  - *Java 1.4 Game Programming:* ***647pp.***
  - *Java 2 Game Programming:* ***784pp.***
  - *Developing Games in Java:* ***996pp.***
  - *Killer Game Programming in Java:* ***970pp.***
  - *Pro Java 6 3D Game Development: Java 3D, JOGL, JInput and JOAL APIs:* ***528 pp.***
  - *Beginning Java Game Programming:* ***346 pp.***
  - *Practical Java Game Programming:* ***508pp.***
  - *Java Game Programming for Dummies:* ***384 pp.***

Motivação para este curso

- Mapa para estudos posteriores.



O que aprender?



Java para Jogos?

- Java é de alto nível / Java é muito lenta!
  - Abstração x velocidade.
  - Partes dependentes/independentes de arquitetura.
  - Velocidade de aplicações gráficas/Swing.
- Ninguém usa para jogos reais!
  - Veja o *Showcase*!
- Não existe interesse da Sun!
  - Novas APIs, esforços comunitários.
- Não existe suporte para consoles!
  - Verdade...
- **Lembrando sempre que o foco deste curso é programação e aprendizado...**

1 Java e Orientação a Objetos

Apesar de ser um curso introdutório, detalhes sobre Java e mecanismos básicos de orientação a objetos serão vistos de forma **superficial**.
O leitor deve ter noções básicas de programação em Java para melhor aproveitamento deste material.

Breve introdução à OO: Encapsulamento

- Atributos e funções relevantes a um domínio ou problema são *encapsulados* em uma classe de forma que:
  - Somente atributos e funções relevantes sejam representados;
  - Uma interface seja criada para que usuários/desenvolvedores tenham acesso somente a funções e atributos que devam ser acessados diretamente.

O desenvolvedor *esconde* os detalhes sobre o funcionamento da classe, mostrando apenas as *funções* que são relevantes ao objeto representado.

Breve introdução à OO: Encapsulamento

- Exemplo: uma nave em um jogo *arcade* (ex. asteróides).
  - Deve ter uma posição (par de coordenadas X e Y).
  - Deve ter um ângulo de orientação.
  - Podemos modificar o ângulo da nave em pequenos incrementos.
  - A posição deve ser modificada por uma regra interna (ex. em função de tempo, velocidade, orientação, etc.)
  - Código externo à classe pode recuperar posição da nave, mas não modificar a posição diretamente.

Classes = atributos + métodos para estes atributos.

Breve introdução à OO: Encapsulamento

```
package exemplos;

public class Nave
  {
  private int x,y;
  private double ângulo;

  public Nave() { } // inicializa variáveis com valores adequados

  public int getX() { return x; }
  public int getY() { return y; }

  public void giraHorário()
    {
    ângulo += 5;     if (ângulo > 360) ângulo = ângulo - 360;
    }
  public void giraAntiHorário()
    {
    ângulo -= 5;     if (ângulo < 0) ângulo = ângulo + 360;
    }

  public void move() { } // modifica X e Y
  public void desenha() { } // desenha nave na posição e orientação

  }
```

Breve introdução à OO: Encapsulamento

```
package exemplos;

public class Nave
  {
  private int x,y;
  private double ângulo;

  public Nave() { } // inicializa variáveis com valores adequados

  public int getX() { return x; }
  public int getY() { return y; }

  public void giraHorário()
    {
    ângulo += 5;
    if (ângulo > 360) ângulo = ângulo - 360;
    }
  public void giraAntiHorário()
    {
    ângulo -= 5;
    if (ângulo < 0) ângulo = ângulo + 360;
    }

  public void move() { } // modifica X e Y
  public void desenha() { } // desenha nave na posição e orientação
  }
```

Classes devem ser organizadas em **pacotes**.

Classes são geralmente **públicas**.

Atributos são geralmente **privados** (veremos herança adiante).

Métodos são geralmente **públicos**.

Devemos ter uma boa idéia do que são construtores e métodos e das palavras-chave usadas nestes exemplos!

Breve introdução à OO: Encapsulamento

- Criamos novos objetos ou instâncias a partir das classes usando a palavra-chave new.
  - Instâncias são materializações ou realizações das classes.
- Construtores são métodos chamados quando criamos instâncias das classes.
  - É onde devemos inicializar atributos e estruturas das classes.

Breve introdução à OO: Encapsulamento

```
package exemplos;

public class UsaNave
  {
  public static void main(String[] args)
    {
    Nave n1 = new Nave();
    n1.giraAntiHorário(); n1.move();
    System.out.println(n1.getX()+","+n1.getY());

    Nave n2 = new Nave();
    n2.x = 10;
    n2.giraAntiHorário(100);
    n2.movePara(120,80);

    }
  }
```

Permitido.

Proibido.

Breve introdução à OO: Herança

- Podemos escrever classes baseadas em outras...
  - Com métodos ou atributos adicionais.
  - Com métodos *superpostos*.
- Exemplo: uma nave em um jogo que tem algumas capacidades adicionais, representadas por atributos.
  - Devemos escrever métodos adicionais para manipular estes atributos.
  - Podemos sobreescrever ou chamar métodos já existentes.
- Não podemos remover declarações de classes ancestrais.

Breve introdução à OO: Herança

```
package exemplos;

public class NaveEspecial extends Nave
  {
  private boolean temRaios;
  private boolean temCanhão;

  public NaveEspecial()
    {
    super(); // construtor da classe ancestral
    } // inicializa variáveis com valores adequados

  public void habilitaRaios()
    {
    temRaios = true;
    }

  // Em que métodos devemos usar os atributos?
  }
```

Atributos adicionais na classe NaveEspecial.

Podemos executar métodos da classe ancestral.

Métodos adicionais na classe NaveEspecial.

Breve introdução à OO: Herança

```
package exemplos;

public class UsaNaveEspecial
  {
  public static void main(String[] args)
    {
    Nave n1 = new Nave();
    n1.giraAntiHorário(); n1.move();
    System.out.println(n1.getX()+","+n1.getY());

    NaveEspecial n2 = new NaveEspecial();
    n2.giraAntiHorário(); n2.move();
    n2.habilitaRaios();
    System.out.println(n2.getX()+","+n2.getY());

    n1.habilitaRaios(); // erro!
    }
  }
```

Permitido.

Proibido.

Breve introdução à OO: Herança

- NaveEspecial pode acessar métodos herdados de Nave:
  - giraHorário, giraAntiHorário, move, desenha, etc.
  - Não pode acessar diretamente atributos como x,y, ângulo!
- Classe ancestral desconhece completamente a classe herdeira:
  - Nave não pode executar método habilitaRaios nem usar atributo temRaios.

Breve introdução à OO: Herança

- Herança deve ser planejada:
  - Classes preferencialmente no mesmo pacote.
  - Atributos devem ser protected.
  - *Protected*: classes no mesmo pacote podem acessar atributos diretamente!
- Considerar o uso de interfaces e classes abstratas.
  - Interfaces: somente contém declarações dos métodos que devem ser implementados: *contratos* para as classes.
  - Classes abstratas não podem ser instanciadas, podem conter declarações ou métodos completos.

Breve introdução à OO: Polimorfismo

- Associado à herança: se a classe B herda da classe A, ela contém os mesmos métodos de A (ou métodos sobrepostos).
  - *B é-um-tipo-de A.*
  - Podemos manipular instâncias de classes herdeiras da mesma forma que instâncias de classes ancestrais.
- Útil para algumas generalizações e estruturas de dados.
- Podemos manipular coleções de muitos objetos da mesma forma, executando métodos comuns.

Breve introdução à Java: Aplicações

- Já vimos exemplo: classes com método main.
- Deve ser public, static, void e receber um *array* de strings como parâmetro.
  - Estas strings podem ser passadas pela linha de comando.
- Outros métodos estáticos podem ser executados como funções, sem precisar criar instância da classe.
- Aplicações executáveis em Java não são arquivos .exe!
- Podemos empacotar aplicações em arquivos .jar, "executáveis" pelo *Java Runtime Environment*.

Breve introdução à Java: Estruturas de Dados

- Estruturas de dados servem para armazenar várias instâncias diferentes com alguma relação entre si.
- A mais simples: *arrays*.
  - Tamanho fixo, todos elementos iguais (respeitando herança).
  - Rápida e indexada diretamente.

```
public class ArrayNaves
  {
  public static void main(String[] args)
    {
    Nave[] frota = new Nave[50];
    for(int i=0;i<frota.length;i++)
      frota[i] = new Nave();
    }
  }
```

O *array* e as instâncias devem ser criadas com new!

- Ainda arrays (e polimorfismo):

```
package exemplos;

public class ArrayNaves2
  {
  public static void main(String[] args)
    {
    Nave[] frota = new Nave[50];
    for(int i=0;i<25;i++)
      frota[i] = new Nave();
    for(int i=25;i<frota.length;i++)
      frota[i] = new NaveEspecial();
    for(Nave n:frota)
      n.move();
    for(Nave n:frota)
      n.habilitaRaios();
    }
  }
```

NaveEspecial é-um-tipo-de Nave.

Nave não é um tipo de NaveEspecial.

- *Sets* (conjuntos): coleção de objetos sem duplicatas.
  - Podem ter melhor performance (HashSet) ou manter a ordem natural dos elementos (TreeSet).
- Métodos mais usados:
  - add: adiciona um objeto ao *set*.
  - remove: remove um objeto do *set*.
  - contains: retorna true se o *set* contém o objeto.
  - addAll: adiciona um *set* a outro.
  - retainAll: retém em um *set* tudo o que estiver em outro: interseção de *sets*.
  - containsAll: retorna true se o *set* conter todos os elementos de outro.

Breve introdução à Java: Estruturas de Dados

```
package exemplos;
import java.util.HashSet;

public class ExemploSet
  {
  // Implementação real deveria ser feita de outra forma!
  public static void main(String[] args)
    {
    HashSet<String> itens = new HashSet<String>();
    itens.add("espada"); // [espada]
    itens.add("lanterna"); // [lanterna, espada]
    itens.add("espada"); // [lanterna, espada]
    itens.add("chave"); // [lanterna, chave, espada]
    itens.add("livro"); // [livro, lanterna, chave, espada]
    System.out.println(itens);
    itens.remove("lanterna"); // [livro, chave, espada]
    System.out.println(itens);
    System.out.println(itens.contains("chave")); // true
    System.out.println(itens.contains("poção")); // false
    System.out.println(itens); // [livro, chave, espada]
    }
  }
```

Breve introdução à Java: Estruturas de Dados

- Listas: lista de objetos, aceita duplicatas.
  - Diferente de *arrays*: tamanho pode aumentar e diminuir.
  - Melhor performance geral (LinkedList), melhor quando não há redimensioamento (ArrayList), especial para pilhas (Stack).
- Métodos mais usados:
  - add: adiciona um objeto à lista.
  - remove: remove um objeto da lista.
  - contains: retorna true se a lista contém o objeto.
  - get: recupera um objeto da lista (por índice).
  - indexOf: retorna o índice do objeto ou -1 se não existir na lista.

Breve introdução à Java: Estruturas de Dados

```
package simon;

import java.util.ArrayList;

public class Simon
  {
  private String[] disponíveis =
    {"azul","amarelo","vermelho","verde","ciano","magenta","laranja"};
  private int dificuldade;
  private int máximo;
  private ArrayList<String> cores;

  public Simon(int dif,int m)
    {
    máximo = m;
    dificuldade = Math.max(dif,disponíveis.length);
    cores = new ArrayList<String>();
    }
```

Breve introdução à Java: Estruturas de Dados

```
  public void adiciona()
    {
    if (cores.size() >= máximo) return;
    int índice = (int)(dificuldade*Math.random());
    cores.add(disponíveis[índice]);
    }

  public void imprime()
    {
    System.out.println(cores);
    }
```

```
package simon;

public class DemoSimon
  {
  public static void main(String[] args)
    {
    Simon s = new Simon(6,10); // seis cores, máximo de 10 entradas.
    s.adiciona();  s.adiciona();  s.adiciona(); s.adiciona();  s.adiciona();
    s.adiciona();  s.adiciona();  s.adiciona(); s.adiciona();  s.adiciona();
    s.adiciona();  s.adiciona();  s.adiciona();
    s.imprime();
    // [verde, azul, amarelo, amarelo, vermelho, verde, magenta, azul, verde, ciano]
    }
  }
```

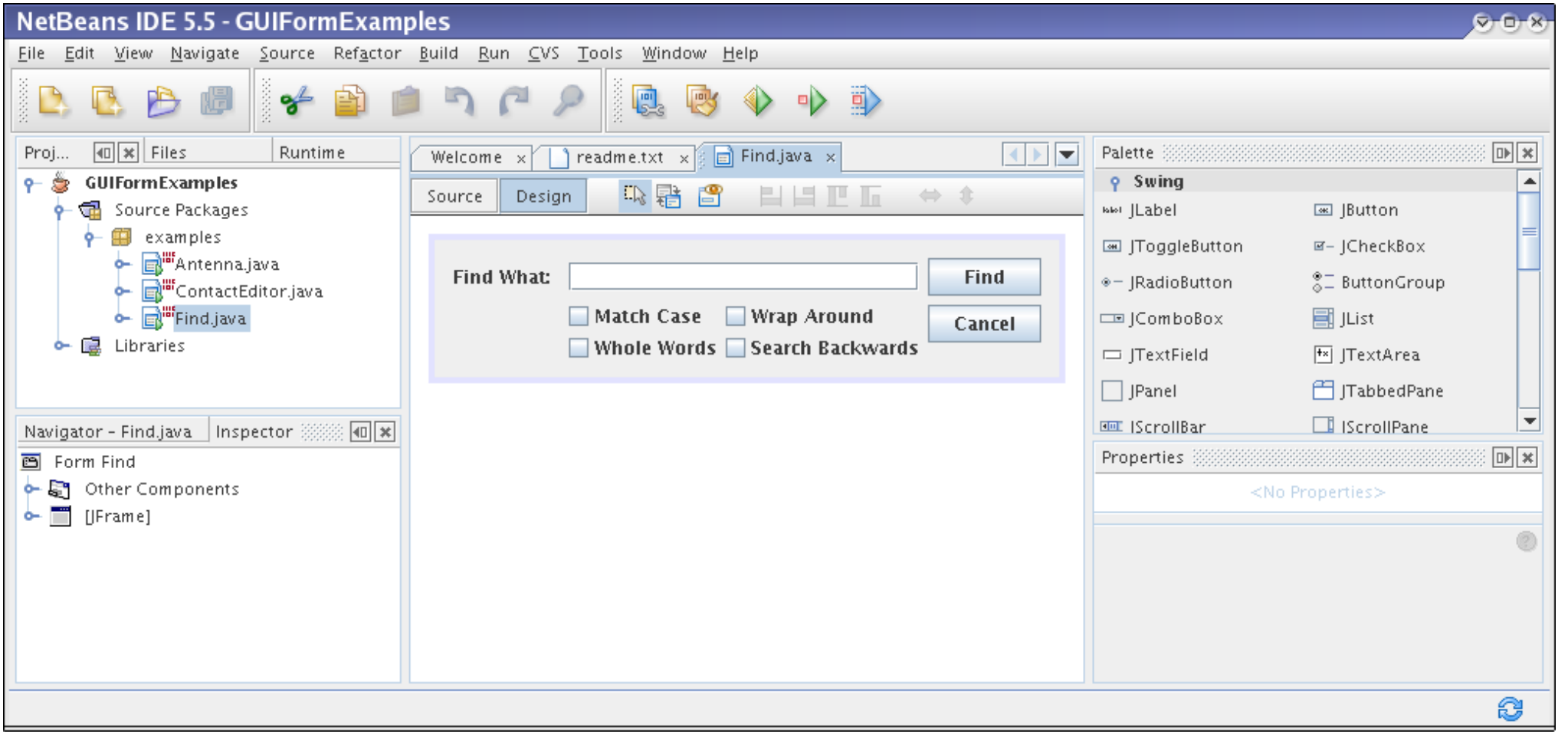
Breve introdução à Java: Estruturas de Dados

- Mapas: Listas com objetos como índices.
  - Mapas associativos: conjuntos de (chave, valor).
  - Interface Map, classes HashMap (boa performance) e TreeMap (chaves ordenadas).
- Métodos mais usados:
  - put: adiciona um objeto ao mapa.
  - remove: remove um objeto do mapa.
  - get: recupera um objeto do mapa.
  - keySet: retorna um set com todas as chaves.
  - values: retorna uma coleção com todos os valores.

2 Mais Java

Aplicações com Interfaces Gráficas

- IDEs permitem a criação visual de interfaces gráficas.
  - Útil para *layouts* complexos.
  - Código gerado automaticamente: bom e ruim.



Aplicações com Interfaces Gráficas

- Código gerado por IDEs pode ser complexo.
- Interfaces gráficas podem ser declaradas pelo programador.
- Simples! Criamos uma classe que herda de JFrame.
  - O construtor pode ser usado para montar a interface gráfica.
  - A própria classe pode ter um método main que cria uma instância dela mesma.
- Vantagens do uso do mecanismo de herança:
  - Vários métodos para JFrames podem ser executados pela nossa classe.
  - Podemos sobreescrever métodos com comportamento específico.

Aplicações com Interfaces Gráficas

- Passos para criar aplicações com interfaces gráficas:
  1. Herdar de JFrame.
  2. Escrever construtor que declara comportamento específico.
  3. Criar método main que instancia a classe.

Aplicações com Interfaces Gráficas

```
package gui;

import javax.swing.JFrame;

public class PrimeiraJanela extends JFrame
  {
  public PrimeiraJanela()
    {
    super("Primeira Janela");
    }

  public static void main(String[] args)
    {
    new PrimeiraJanela();
    }
  }
```

- Ao executar o código, nada aparece!
- Faltou executar métodos que definem aparência e comportamento básico da aplicação.

Aplicações com Interfaces Gráficas

```
package gui;

import javax.swing.JFrame;

public class SegundaJanela extends JFrame
  {
  public SegundaJanela()
    {
    super("Segunda Janela");
    setSize(400,200);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

  public static void main(String[] args)
    {
    new SegundaJanela();
    }
  }
```

Aplicações com Interfaces Gráficas

- Passos (revistos) para criar aplicações com interfaces gráficas:
  1. Herdar de JFrame.
  2. Escrever construtor que declara comportamento específico.
  3. **No construtor, instanciar componentes gráficos. Adicionar os componentes ao Container (getContentPane).**
  4. Criar método main que instancia a classe.

Aplicações com Interfaces Gráficas

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

public class Dado0 extends JFrame
  {
  private JButton botão;
  private JLabel label;

  public Dado1()
    {
    super("Dado");
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,2));
    botão = new JButton("Sorteia novo número");    c.add(botão);
    label = new Jlabel("");                        c.add(label);
    pack();
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

  public static void main(String[] args)
    {
    new Dado1();
    }
  }
```

Aplicações com Interfaces Gráficas: Eventos

- Eventos ocorrem quando (entre outros)...
  - Ativamos ou modificamos o conteúdo de um componente.
  - Movemos o mouse ou usamos o teclado em componentes com foco.
  - Modificamos de qualquer forma uma janela.
  - Causados por temporizadores.
- Passos:
  1. Criamos objetos.
  2. Criamos classes que contém *listeners* para determinados tipos de eventos.
  3. Registramos *listeners* para eventos nos objetos criados.

Aplicações com Interfaces Gráficas

- Passos (revisitados) para criar aplicações com interfaces gráficas:
  1. Herdar de JFrame e **implementar interfaces de eventos.**
  2. Escrever construtor que declara comportamento específico.
  3. No construtor, instanciar componentes gráficos. Adicionar os componentes ao Container (getContentPane).
  4. **No construtor, registrar eventos.**
  5. **Implementar métodos definidos pelas interfaces de eventos.**
  6. Criar método main que instancia a classe.

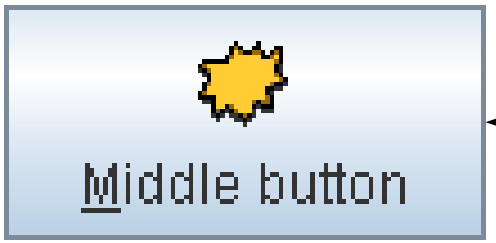
Aplicações com Interfaces Gráficas

```
package gui;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class Dado1 extends JFrame implements ActionListener
  {
  private JButton botão;
  private JLabel label;

  public Dado1()
    {
    super("Dado");
    Container c = getContentPane();
    c.setLayout(new GridLayout(1,2));
    botão = new JButton("Sorteia novo número");
    label = new JLabel("");
    c.add(botão);
    c.add(label);
    botão.addActionListener(this);
    pack();
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

Aplicações com Interfaces Gráficas

```
public void actionPerformed(ActionEvent e)
  {
  if (e.getSource() == botão)
    label.setText(""+(int)(1+Math.random()*6));
  }

public static void main(String[] args)
  {
  new Dado1();
  }
```

Componentes para GUIs



JButton

eventos tipo ActionEvent (classe deve implementar ActionListener)

JTextField

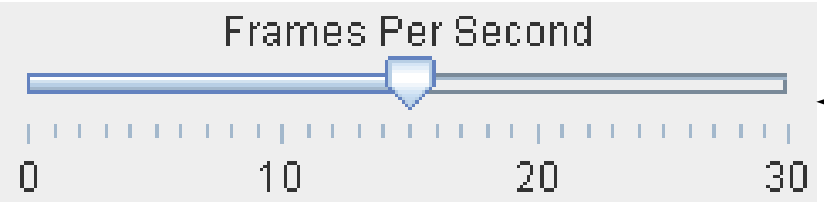
eventos tipo ActionEvent (classe deve implementar ActionListener)

JPasswordField

eventos tipo ActionEvent (classe deve implementar ActionListener)

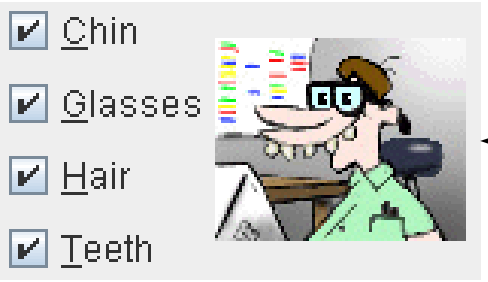
http://java.sun.com/docs/books/tutorial/ui/features/components.html

Componentes para GUIs



JSlider

eventos tipo ChangeEvent (classe deve implementar ChangeListener)



JCheckBox

eventos tipo ItemEvent (classe deve implementar ItemListener)

JRadioButton/ButtonGroup

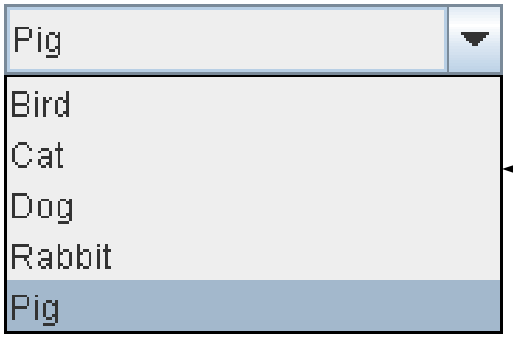
eventos tipo ActionEvent (classe deve implementar ActionListener)

http://java.sun.com/docs/books/tutorial/ui/features/components.html

Componentes para GUIs

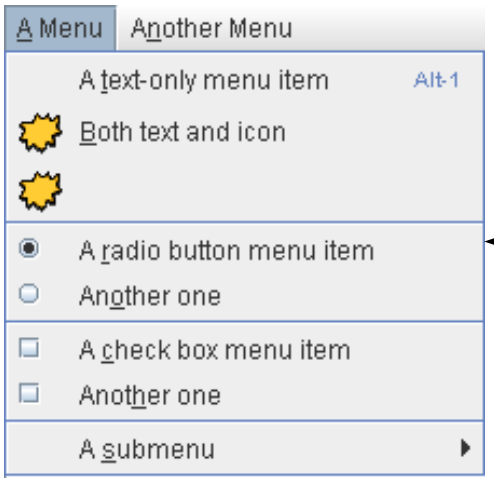
JList

eventos tipo ListSelectionEvent (classe deve implementar ListSelectionListener)



JComboBox

eventos tipo ActionEvent (classe deve implementar ActionListener)



JMenu

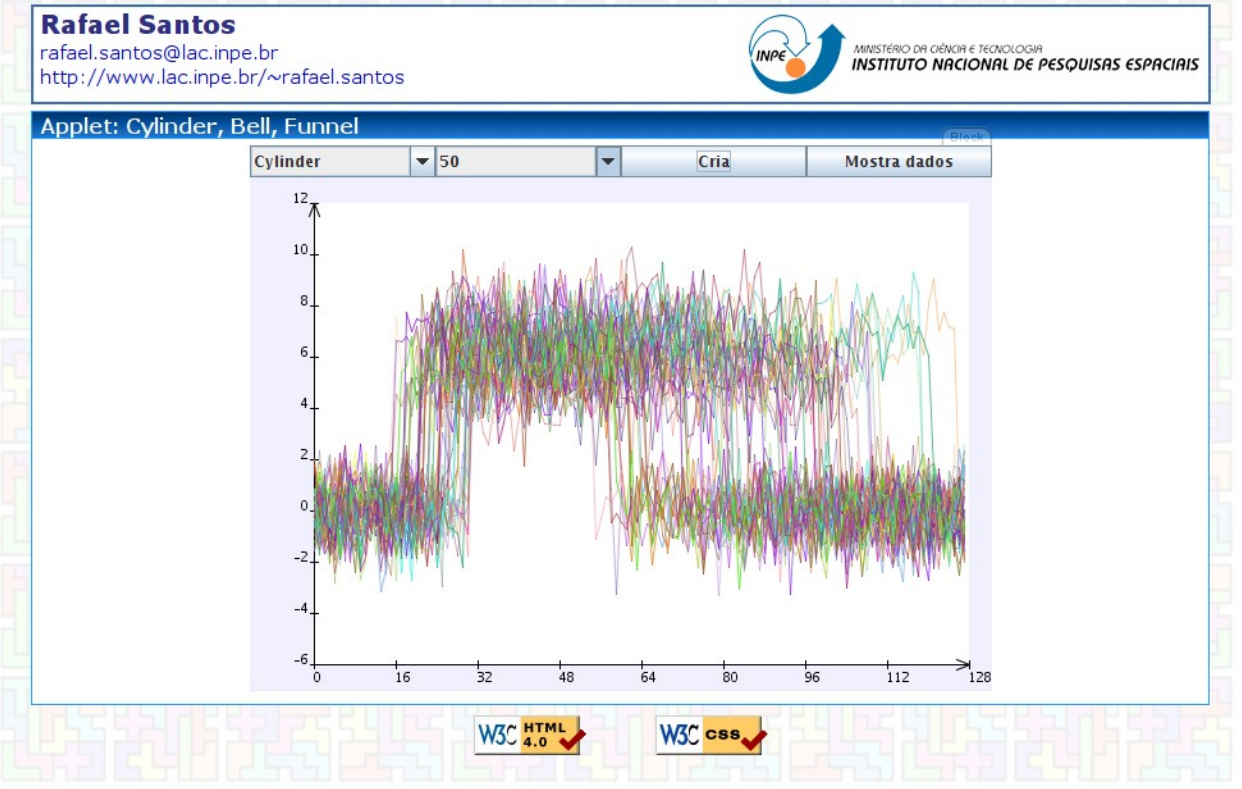
eventos tipo ActionEvent (classe deve implementar ActionListener)

http://java.sun.com/docs/books/tutorial/ui/features/components.html

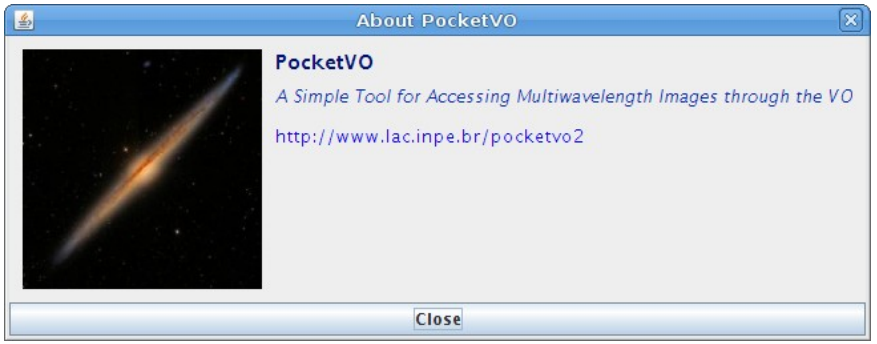
Componentes *Top-Level* para GUIs

JFrame

JApplet

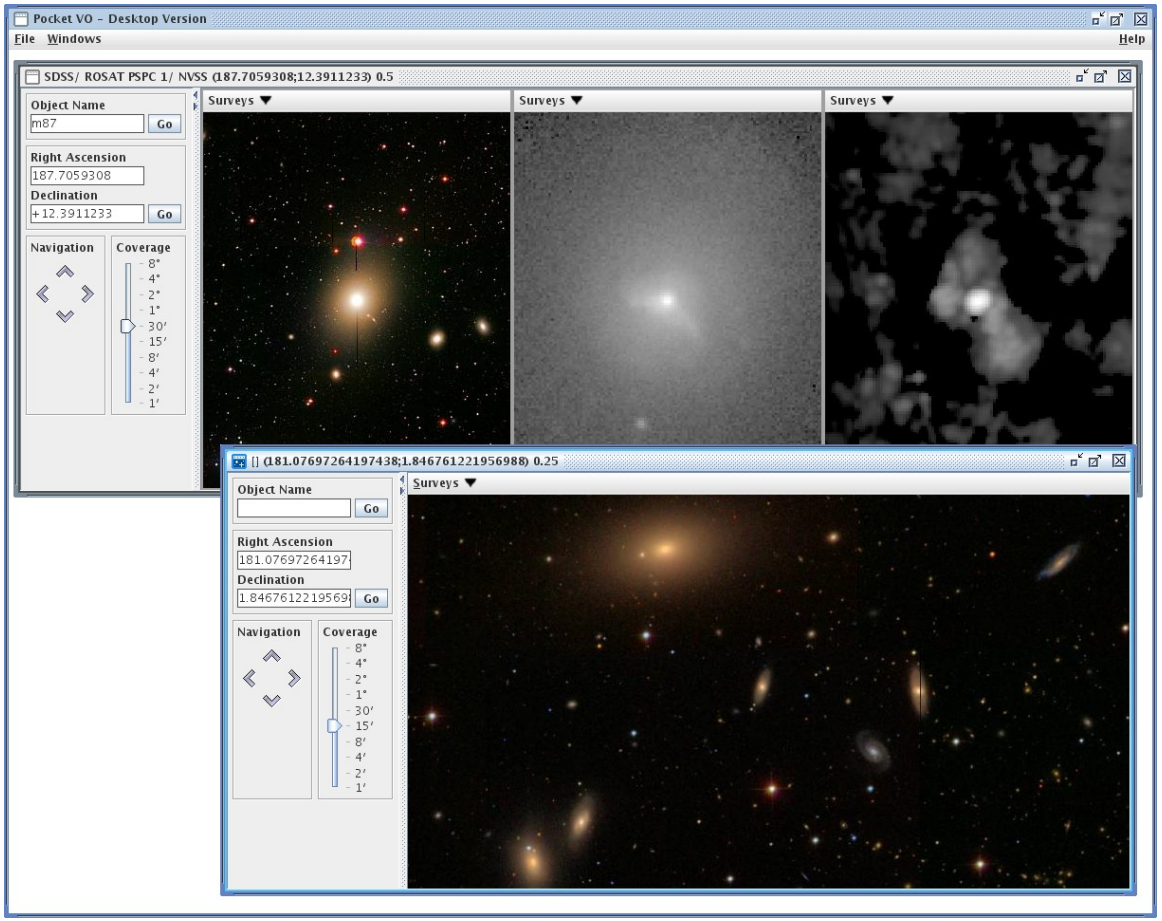


Componentes *Top-Level* para GUIs



JDialog

JDesktopFrame/
JInternalFrame



GUIs: Exemplos mais complexos

- Aplicação para exibir imagens em JFrames internas.
- Primeiro, a classe principal (com um JDesktopPane).

```
package exemplos;

import java.awt.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.*;
import javax.imageio.ImageIO;
import javax.swing.*;

public class MostraMultiplasImagens extends JFrame implements ActionListener
  {
  private JDesktopPane desktop;
  private JTextField url;
  private JComboBox escala;
  private String[] escalas = {"0.01","0.05","0.1","0.2","0.5","1","2","5","10","20"};
```

Pula?

GUIs: Exemplos mais complexos

```
public MostraMultiplasImagens()
  {
  desktop = new JDesktopPane();
  JPanel controle = new JPanel(new FlowLayout(FlowLayout.LEFT));
  controle.add(new JLabel("URL da Imagem:"));
  url = new JTextField(50);
  url.addActionListener(this);
  url.setText("http://www.lac.inpe.br/~rafael.santos/JIPCookbook/"+
              "MiscResources/Datasets/Landsat7_218076_742.png");
  controle.add(url);
  controle.add(new JLabel("Escala:"));
  escala = new JComboBox(escalas);
  escala.setSelectedIndex(5);
  controle.add(escala);
  getContentPane().add(controle,BorderLayout.NORTH);
  getContentPane().add(desktop,BorderLayout.CENTER);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(800,600);
  setVisible(true);
  }
```

GUIs: Exemplos mais complexos

```
public void actionPerformed(ActionEvent e)
  {
  BufferedImage imagem = null;       boolean carregada = true;
  try  {  imagem = ImageIO.read(new URL(url.getText()));  }
  catch (MalformedURLException e1)
    {
    JOptionPane.showMessageDialog(this,"Erro na URL: "+url.getText(),
                                       "Erro na URL",JOptionPane.ERROR_MESSAGE);
    carregada = false;
    }
  catch (IOException e1)
    {
    JOptionPane.showMessageDialog(this,"Erro de IO: "+e1.getMessage(),
                                       "Erro de IO",JOptionPane.ERROR_MESSAGE);
    carregada = false;
    }
  if (carregada)
    {
    if (imagem == null)
      {
      JOptionPane.showMessageDialog(this,"Não pode ler "+url.getText(),
                                         "Não pode ler",JOptionPane.ERROR_MESSAGE);
      }
    else
      {
      float usaEscala = Float.parseFloat(escalas[escala.getSelectedIndex()]);
      ImagemIF i = new ImagemIF(url.getText(),usaEscala,new ImageIcon(imagem));
      desktop.add(i);
      }
    }
  }
```

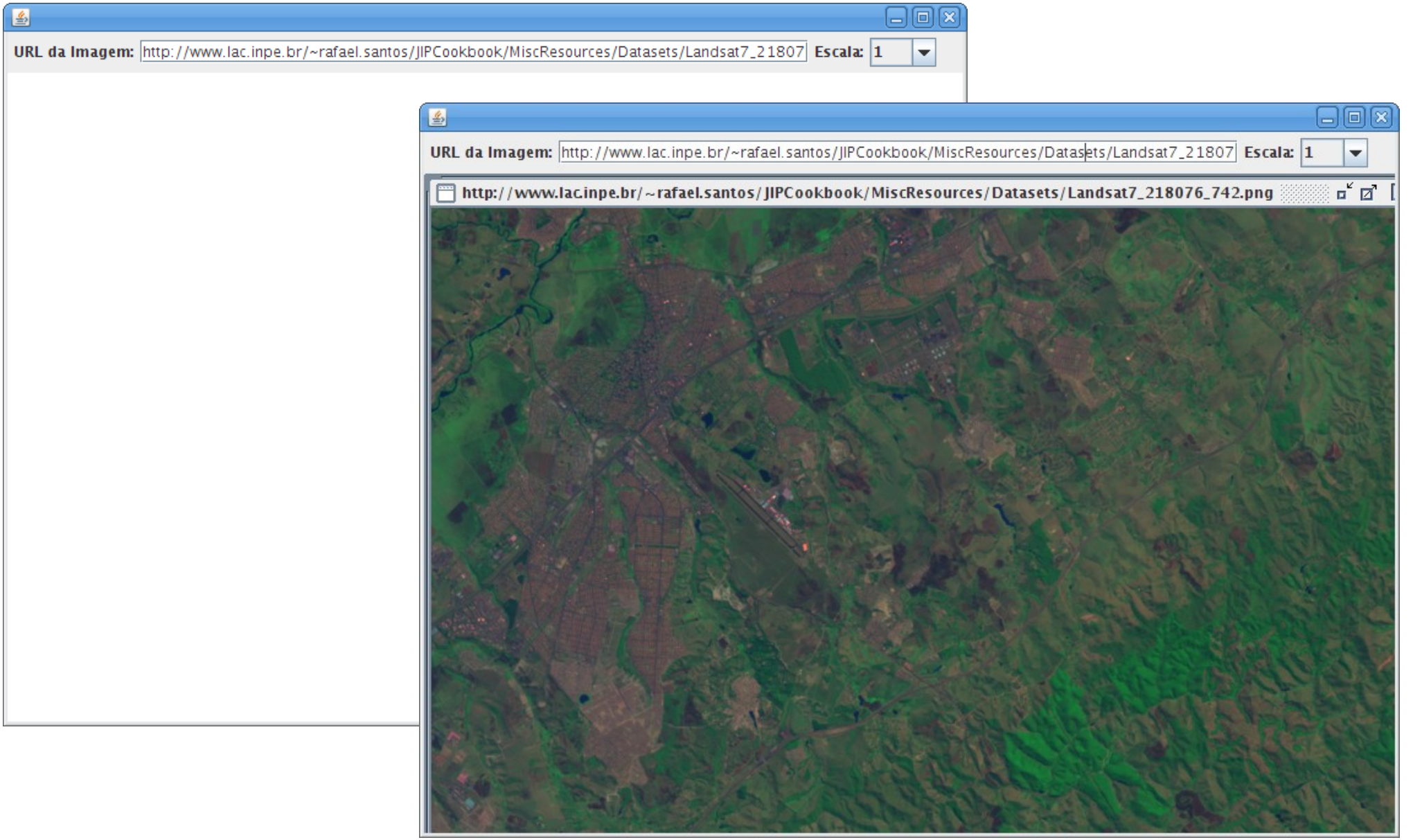
```
public static void main(String[] args)
  { new MostraMultiplasImagens(); }
```

GUIs: Exemplos mais complexos

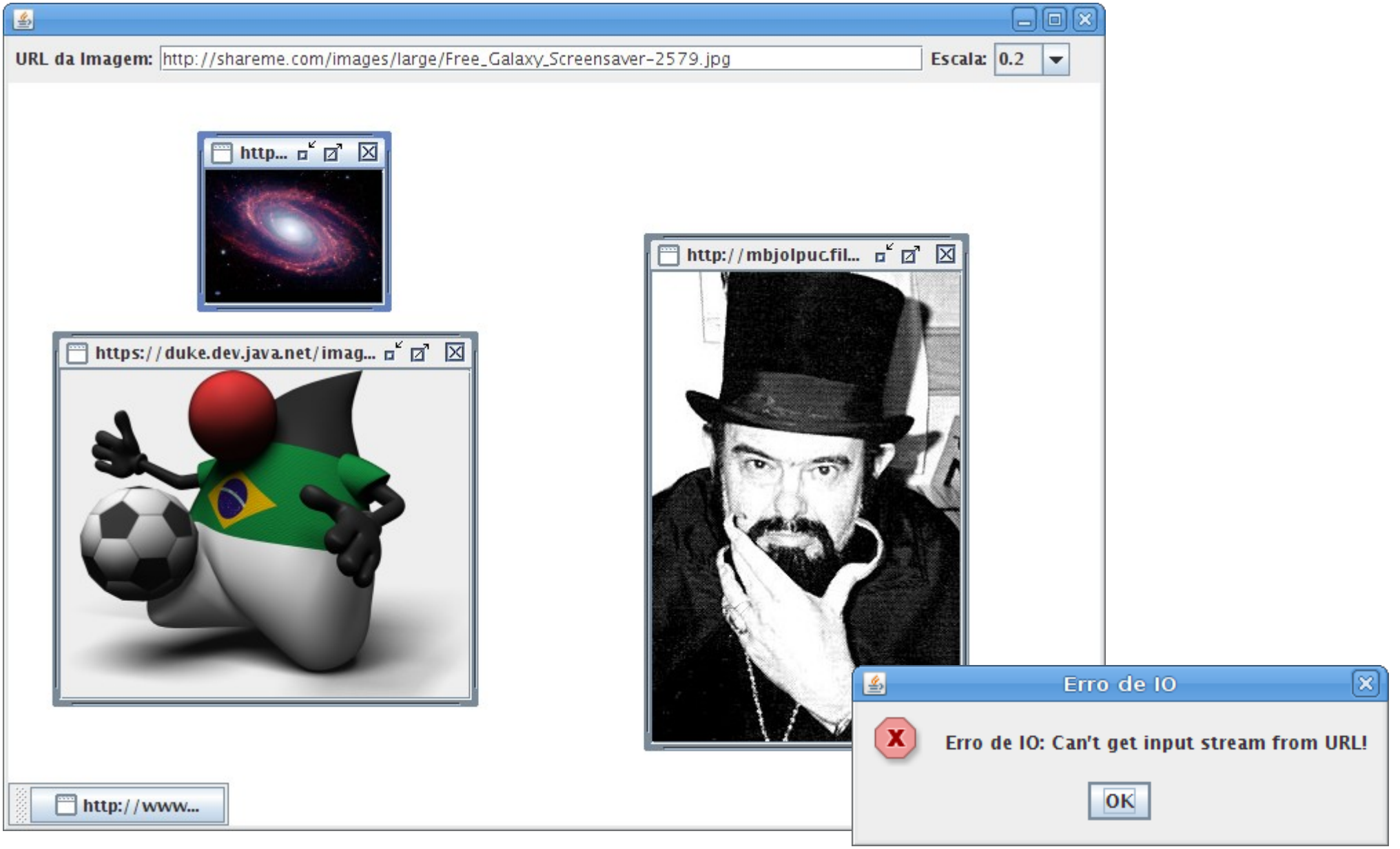
- Classe que herda de JInternalFrame:

```
package exemplos;

import java.awt.Image;
import javax.swing.*;

public class ImagemIF extends JInternalFrame
  {
  public ImagemIF(String name,float escala,ImageIcon ícone)
    {
    // Resizable, closable, maximizable e iconifiable.
    super(name,true,true,true,true);
    // Vamos mudar a escala da imagem?
    float width = ícone.getIconWidth();
    float height = ícone.getIconHeight();
    width *= escala; height *= escala;
    ícone =
      new ImageIcon(ícone.getImage().getScaledInstance((int)width,
                                                       (int)height,Image.SCALE_SMOOTH));
    // Mostra em um JLabel.
    getContentPane().add(new JScrollPane(new JLabel(ícone)));
    pack();
    setVisible(true);
    }
  }
```

GUIs: Exemplos mais complexos



GUIs: Exemplos mais complexos

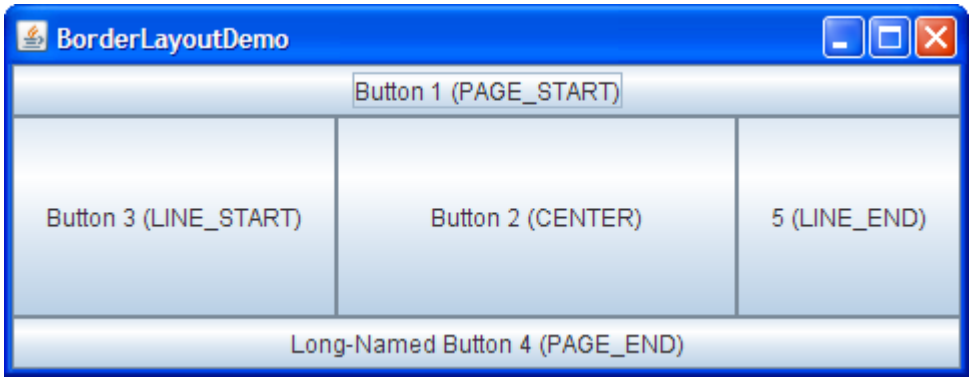


Layouts para GUIs

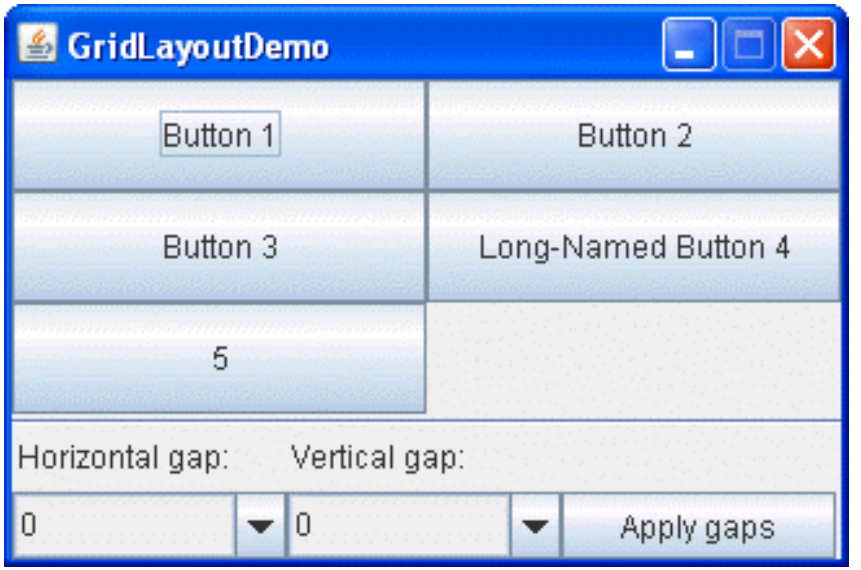
- Como os componentes serão arranjados e exibidos na interface gráfica?
  - ***Layouts***.
- Existem vários *layouts* simples, implementados como classes.
  - Podemos criar um *layout* diferente ou mesmo não usar nenhum!
- Para usar um *layout*, executamos um método para indicar o *layout* do painel de conteúdo da aplicação.
- O *layout* também indica como os componentes serão rearranjados se o tamanho da janela mudar.

Layouts para GUIs

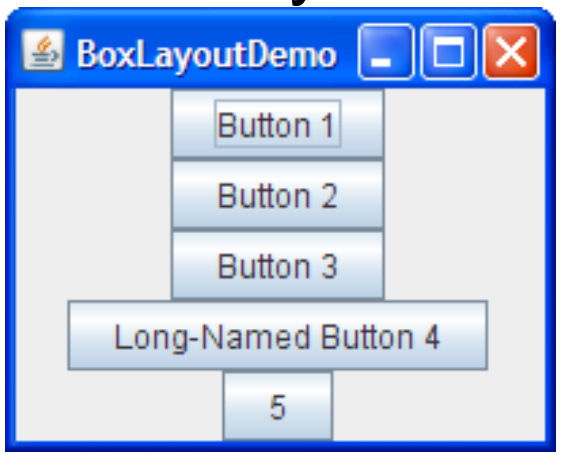
BorderLayout



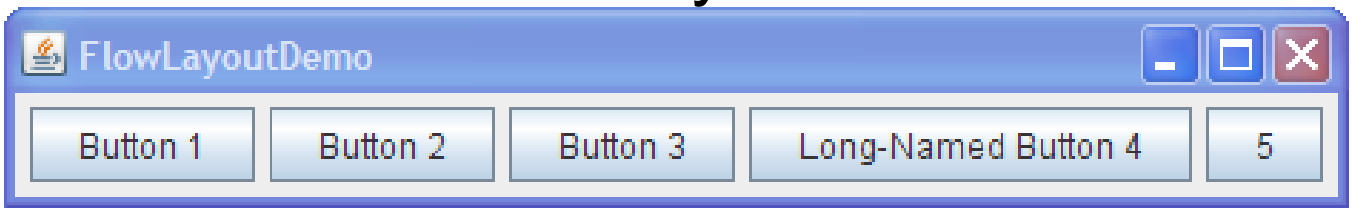
GridLayout



BoxLayout



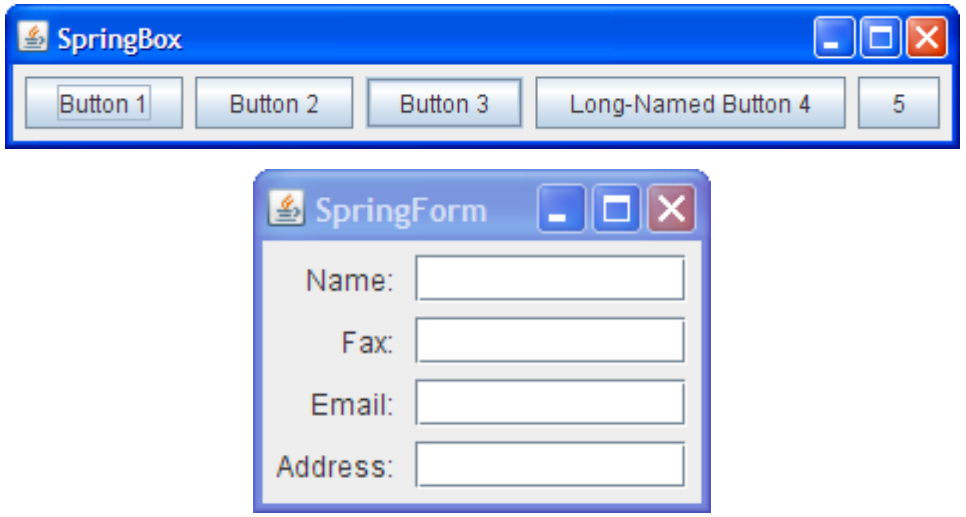
FlowLayout



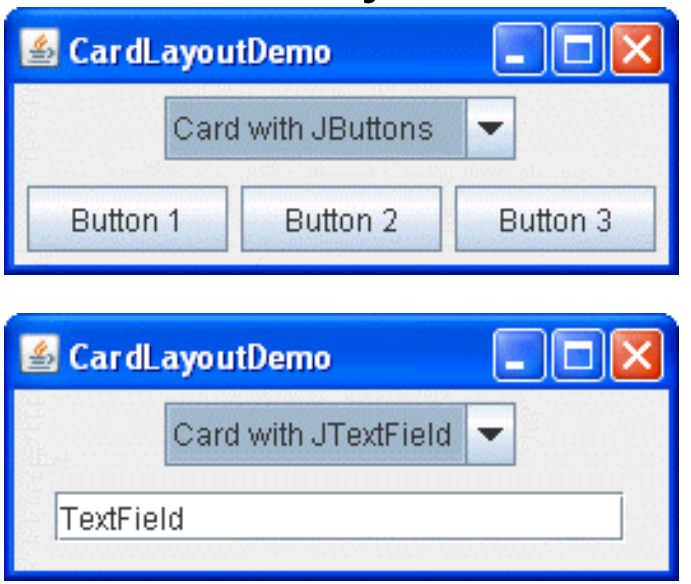
http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html

Layouts para GUIs

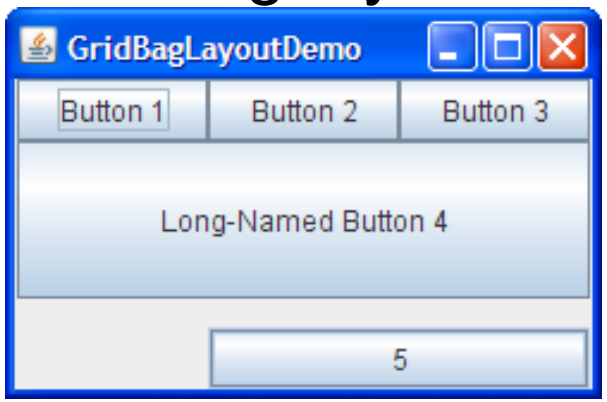
SpringLayout



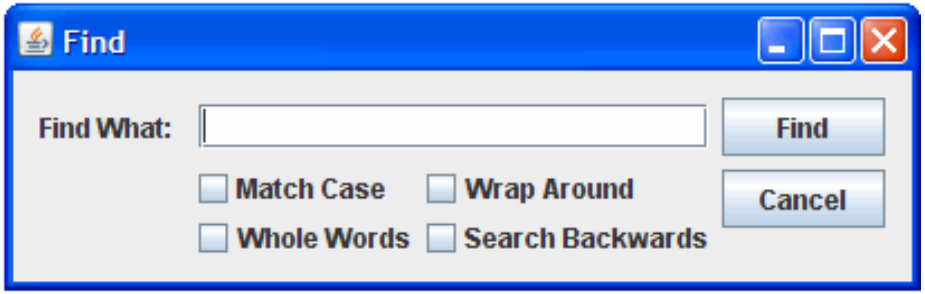
CardLayout



GridBagLayout



GroupLayout



http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html

Layouts para GUIs

- Podemos combinar *layouts* (exemplo de GridLayout):
  - Usando JPanels, cada um com seu *layout*.
- Podemos **não** usar um *layout*: setLayout(null).
  - Devemos posicionar cada componente manualmente, com coordenadas em pixels.
  - Métodos úteis:
    - Componentes: get/setPreferredSize() e setBounds()
    - Painel: getInsets()
  - Cuidado com pack!

Criando Novos Componentes Gráficos

- Pode ser necessário criar novos componentes para:
  - Exibição de dados (jogos) com formato específico;
  - Entrada de informações especializadas;
  - Exibir comportamento diferente dos componentes já existentes.
- Duas abordagens:
  - Criar componentes que herdam de outros, já existentes, com funções similares.
  - Criar novos componentes a partir de um componente genérico.

Criando Novos Componentes Gráficos

- Passos (nem todos são obrigatórios):
  - Herdar de classe que tem comportamento semelhante.
  - No construtor, chamar construtor ancestral, inicializar atributos relevantes e modificar comportamento através de métodos.
  - Sobreescrever métodos get{Maximum,Minimum,Preferred}Size().
  - Sobreescrever paintComponent().

Criando Novos Componentes Gráficos

- Exemplo: peça para Reversi.
  - Botão com aparência e comportamento diferente.
  - Existem várias maneiras de implementar...

```
package reversi;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JButton;

public class Peca extends JButton
  {
  private static int tamanho = 64;
  private Estado estado;

  public Peca()
    {
    super();
    estado = Estado.VAZIO;
    }
```

```
package reversi;

public enum Estado { VAZIO, PRETO, BRANCO }
```

Criando Novos Componentes Gráficos

```
public Dimension getMaximumSize() { return getPreferredSize(); }
public Dimension getMinimumSize() { return getPreferredSize(); }
public Dimension getPreferredSize() { return new Dimension(tamanho,tamanho); }

protected void paintComponent(Graphics g)
  {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  // Não preenchemos botões vazios.
  if (estado != Estado.VAZIO)
    {
    if (estado == Estado.BRANCO) g2d.setColor(Color.WHITE);
    else if (estado == Estado.PRETO) g2d.setColor(Color.BLACK);
    g2d.fillOval(6,6,getWidth()-12,getHeight()-12);
    }
  // Pintamos a borda da peça independente do estado.
  g2d.setColor(Color.GRAY);
  g2d.drawOval(6,6,getWidth()-12,getHeight()-12);
  }
}
```

Criando Novos Componentes Gráficos

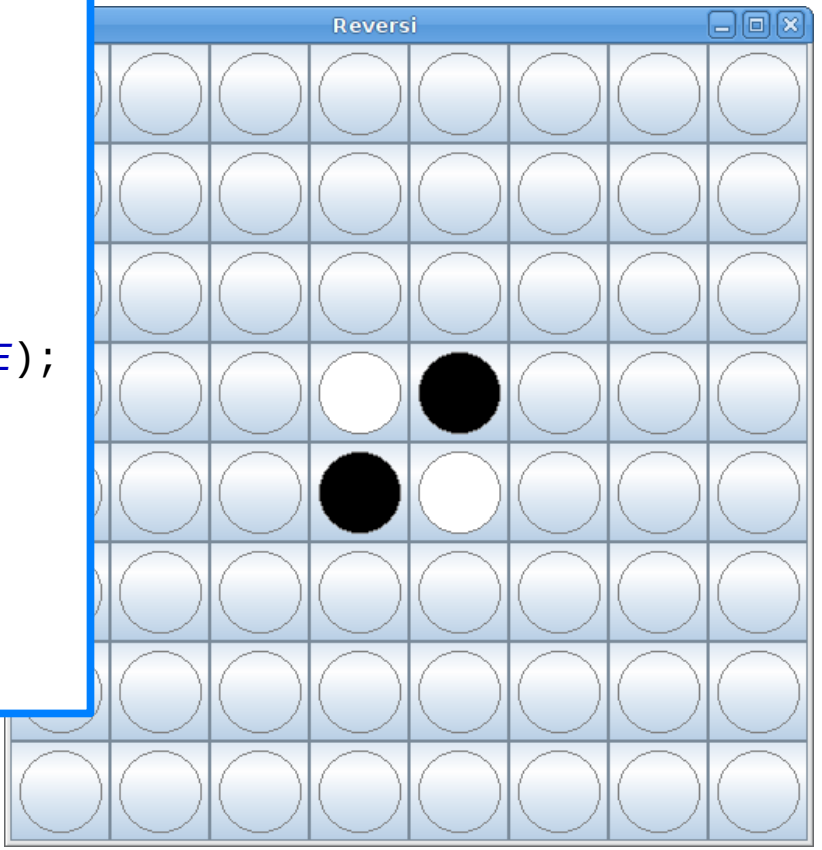
- Para mostrar a interface gráfica precisamos de:
  - Classe Peca, que representa botões para o jogo.
  - Classe Tabuleiro, que é um conjunto 8x8 de peças.
  - Classe Jogo, que é a aplicação que usa Tabuleiro.
- Um jogo funcional precisaria ainda de...
  - Regras do jogo, que possivelmente devem ser implementadas com código em todas as classes.
  - Mecanismo do adversário: rede (para adversário humano), inteligência artificial (para computador).

Criando Novos Componentes Gráficos

```
package reversi;

import java.awt.GridLayout;
import javax.swing.JPanel;

public class Tabuleiro extends JPanel
  {
  private Peca[][] tabuleiro;

  public Tabuleiro()
    {
    setLayout(new GridLayout(8,8));
    tabuleiro = new Peca[8][8];
    for(int l=0;l<8;l++)
      for(int c=0;c<8;c++)
        {
        tabuleiro[c][l] = new Peca();
        add(tabuleiro[c][l]);
        }
    tabuleiro[3][3].setEstado(Estado.BRANCO);
    tabuleiro[4][4].setEstado(Estado.BRANCO);
    tabuleiro[3][4].setEstado(Estado.PRETO);
    tabuleiro[4][3].setEstado(Estado.PRETO);
    }

  }
```

Criando Novos Componentes Gráficos

```
package reversi;

import javax.swing.JFrame;

public class Jogo extends JFrame
  {

  public Jogo()
    {
    super("Reversi");
    getContentPane().add(new Tabuleiro());
    pack();
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

  public static void main(String[] args)
    {
    new Jogo();
    }
  }
```



Dá pra melhorar a aparência?

Criando Novos Componentes Gráficos

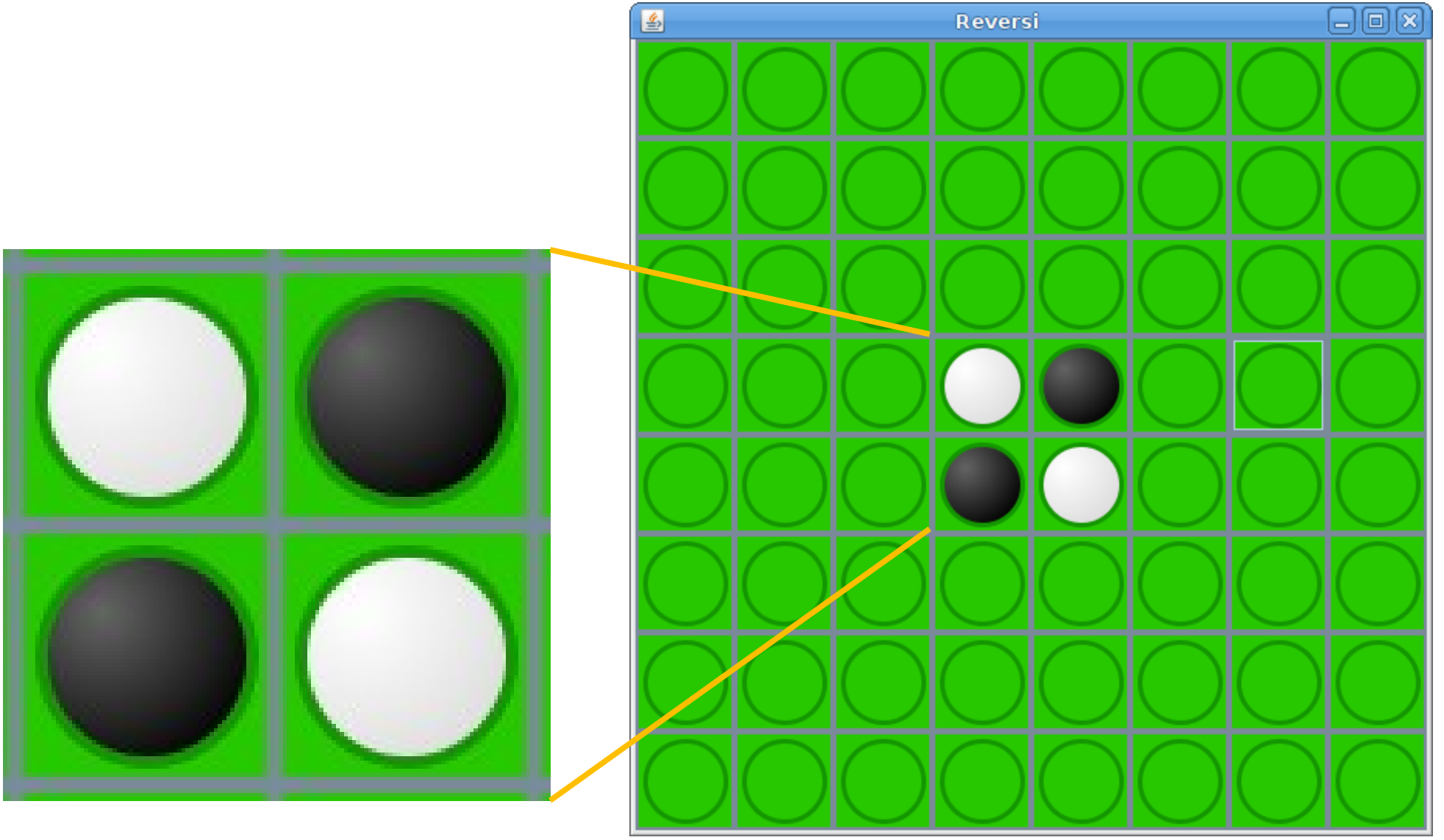
```
package reversi;

import java.awt.*;
import java.awt.geom.Point2D;

import javax.swing.JButton;

public class PecaMelhor extends JButton
  {
  private static int tamanho = 64;
  private Estado estado;

  public PecaMelhor()
    {
    super();
    setBackground(new Color(40,200,0));
    estado = Estado.VAZIO;
    }

  public void setEstado(Estado e) { estado = e; }

  public Dimension getMaximumSize() { return getPreferredSize(); }
  public Dimension getMinimumSize() { return getPreferredSize(); }
  public Dimension getPreferredSize() { return new Dimension(tamanho,tamanho); }
```

Criando Novos Componentes Gráficos

```
protected void paintComponent(Graphics g)
  {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);
  // Não preenchemos botões vazios.
  if (estado != Estado.VAZIO)
    {
    Color[] cores = new Color[2];
    if (estado == Estado.BRANCO)
      { cores[0] = Color.WHITE; cores[1] = new Color(220,220,220); }
    else if (estado == Estado.PRETO)
      { cores[0] = new Color(100,100,100); cores[1] = Color.BLACK; }
    RadialGradientPaint paint =
      new RadialGradientPaint(new Point2D.Double(tamanho/3,tamanho/3),
                              2*tamanho/3,new float[]{0f,1f},cores);
    g2d.setPaint(paint);
    g2d.fillOval(6,6,getWidth()-12,getHeight()-12);
    }
  // Pintamos a borda da peça independente do estado.
  g2d.setColor(new Color(20,150,0));
  g2d.setStroke(new BasicStroke(3f));
  g2d.drawOval(6,6,getWidth()-12,getHeight()-12);
  }
}
```

Criando Novos Componentes Gráficos

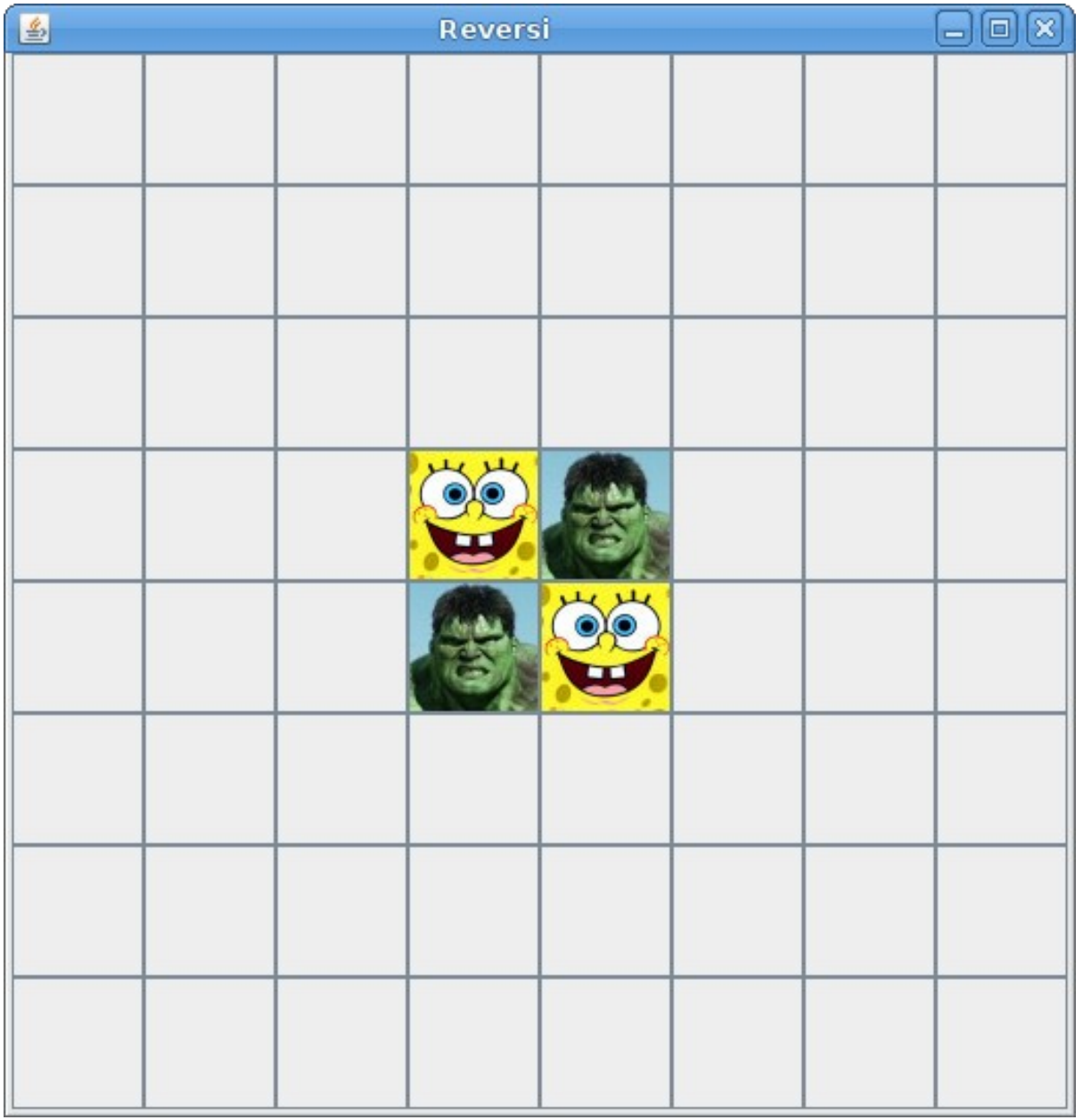
- Basta usar PecaMelhor no lugar de Peca em Tabuleiro.



Criando Novos Componentes Gráficos

```
...
public class PecaIcone extends JButton
  {
  private static int tamanho = 64;
  private Estado estado;
  private Image i1,i2;

  public PecaIcone()
    {
    super();
    setContentAreaFilled(false);
    estado = Estado.VAZIO;
    i1 = new ImageIcon(getClass().getResource("/Sprites/sbob.jpg")).getImage();
    i2 = new ImageIcon(getClass().getResource("/Sprites/hulk.jpg")).getImage();
    } ...
  protected void paintComponent(Graphics g)
    {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    // Não preenchemos botões vazios.
    if (estado != Estado.VAZIO)
      {
      if (estado == Estado.BRANCO) g2d.drawImage(i1,0,0,null);
      else if (estado == Estado.PRETO) g2d.drawImage(i2,0,0,null);
      }
    }...
```

Criando Novos Componentes Gráficos



Criando Novos Componentes Gráficos

- Outro exemplo: componente que processa seus próprios eventos.

```
package rabisco;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import javax.swing.JComponent;

public class ComponenteParaRabiscos extends JComponent
  implements MouseListener,MouseMotionListener,KeyListener
  {
  private ArrayList<Point> pontos;
  private int size = 8; private int halfsize = size/2;
  private Color cor;

  public ComponenteParaRabiscos(Color cor)
    {
    this.cor = cor;
    pontos = new ArrayList<Point>(1024);
    addMouseListener(this);
    addMouseMotionListener(this);
    addKeyListener(this);
    setFocusable(true);
    requestFocus();
    }
```

Pula?

Criando Novos Componentes Gráficos

```
protected void paintComponent(Graphics g)
   {
   Graphics2D g2d = (Graphics2D)g;
   g2d.setColor(Color.WHITE);
   g2d.fillRect(0,0,getWidth(),getHeight());
   g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
   g2d.setColor(cor);
   for(Point p:pontos)
     {
     g2d.fillOval(p.x-halfsize,p.y-halfsize,size,size);
     }
   }

 public void mousePressed(MouseEvent e)
   {
   pontos.add(e.getPoint());
   repaint();
   }

 public void mouseDragged(MouseEvent e)
   {
   pontos.add(e.getPoint());
   repaint();
   }
```

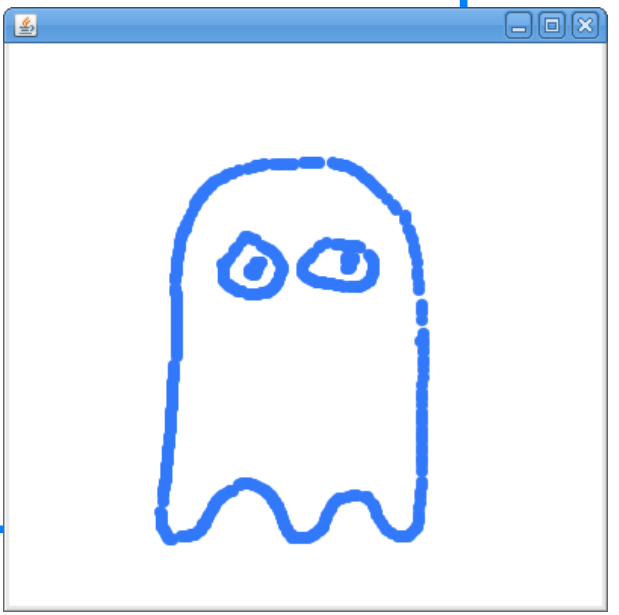
Criando Novos Componentes Gráficos

```
  public void mouseEntered(MouseEvent e)
    {
    requestFocus();
    }

  public void mouseReleased(MouseEvent e) { } // NOP
  public void mouseClicked(MouseEvent e) { } // NOP
  public void mouseExited(MouseEvent e) { } // NOP
  public void mouseMoved(MouseEvent e) { } // NOP

  public void keyPressed(KeyEvent e)
    {
    System.out.println(e.getKeyCode());
    if (e.getKeyCode() == KeyEvent.VK_C)
      {
      pontos.clear();
      repaint();
      }
    }

  public void keyReleased(KeyEvent e) { } // NOP
  public void keyTyped(KeyEvent e) { } // NOP
  }
```

Criando Novos Componentes Gráficos

- Aplicação que usa o componente

```
package rabisco;

import java.awt.Color;
import java.awt.Container;
import javax.swing.JFrame;

public class AppRabiscos extends JFrame
  {
  public AppRabiscos()
    {
    ComponenteParaRabiscos c = new ComponenteParaRabiscos(new Color(50,120,250));
    Container cp = getContentPane();
    cp.add(c);
    setSize(400,400);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

  public static void main(String[] args)
    {
    new AppRabiscos();
    }

  }
```



Criando Novos Componentes Gráficos

- Outro exemplo: componente que produz ***e*** processa seus próprios eventos.

```
package exemplos;

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

public class ComponenteLuzVermelha extends JComponent implements ActionListener
  {
  private int nível,passo;
  private Timer timer;

  public ComponenteLuzVermelha(int passo)
    {
    this.passo = passo;
    nível = 0;
    setPreferredSize(new Dimension(200,200));
    timer = new Timer(50,this);
    timer.setCoalesce(true);
    timer.start();
    }
```

Pula?

Criando Novos Componentes Gráficos

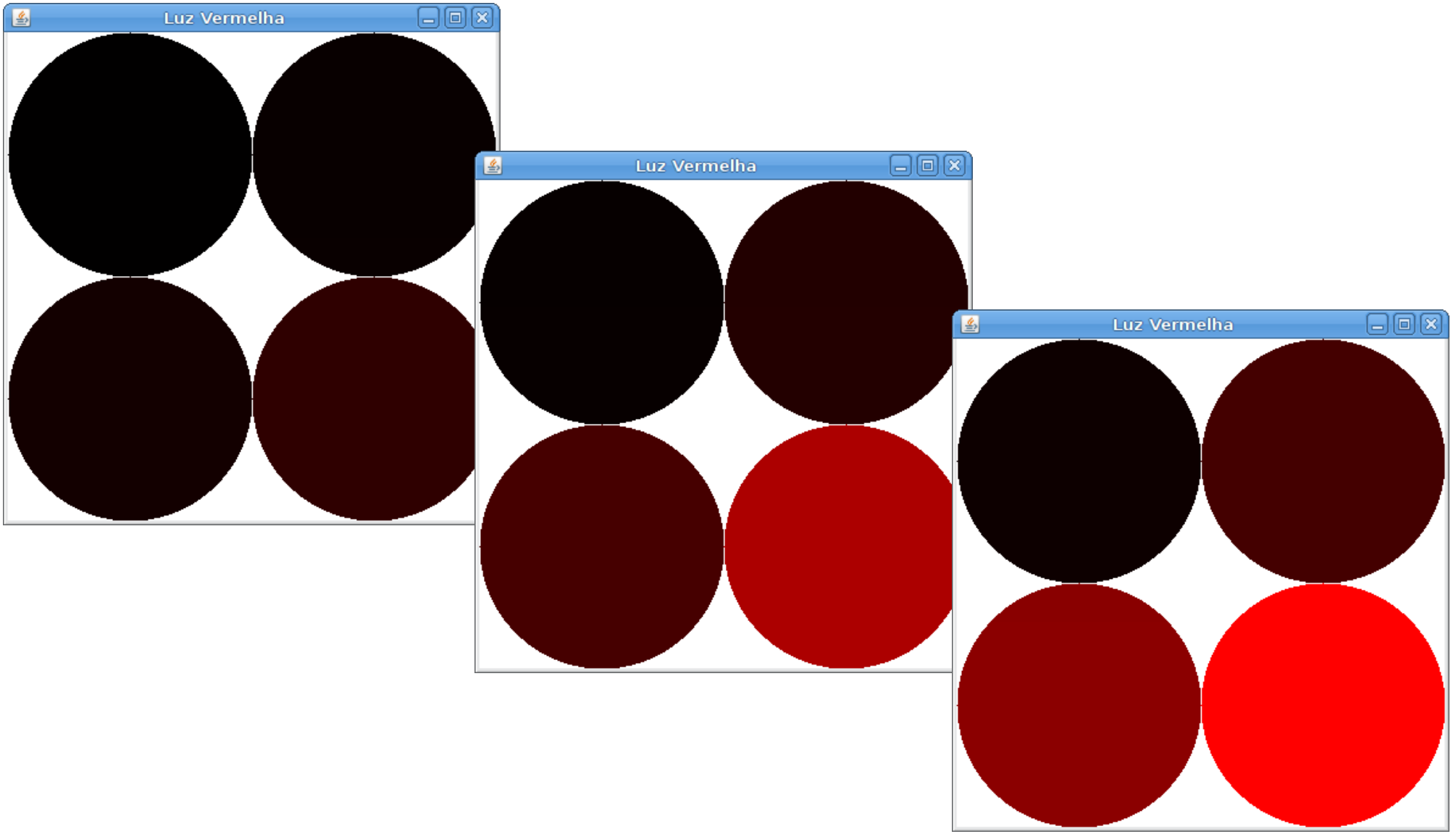
```
protected void paintComponent(Graphics g)
  {
  g.setColor(Color.WHITE);
  g.fillRect(0,0,getWidth(),getHeight());
  // Calculamos a cor de acordo com o passo.
  g.setColor(new Color(nível/100,0,0));
  g.fillArc(0,0,getWidth(),getHeight(),0,360);
  }

public void actionPerformed(ActionEvent e)
  {
  if (nível < 25500) nível += passo;
  repaint();
  }
```

Criando Novos Componentes Gráficos

- Aplicação que usa o componente

```
package exemplos;

import java.awt.*;
import javax.swing.*;

public class LuzVermelhaApp extends JFrame
  {
  public LuzVermelhaApp()
    {
    super("Luz Vermelha");
    getContentPane().setLayout(new GridLayout(2,2));
    ComponenteLuzVermelha c1,c2,c3,c4;
    c1 = new ComponenteLuzVermelha(10);    c2 = new ComponenteLuzVermelha(50);
    c3 = new ComponenteLuzVermelha(100);   c4 = new ComponenteLuzVermelha(250);
    getContentPane().add(c1); getContentPane().add(c2);
    getContentPane().add(c3); getContentPane().add(c4);
    pack();
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

  public static void main(String[] args)
    { new LuzVermelhaApp(); }
  }
```

Criando Novos Componentes Gráficos



Applets

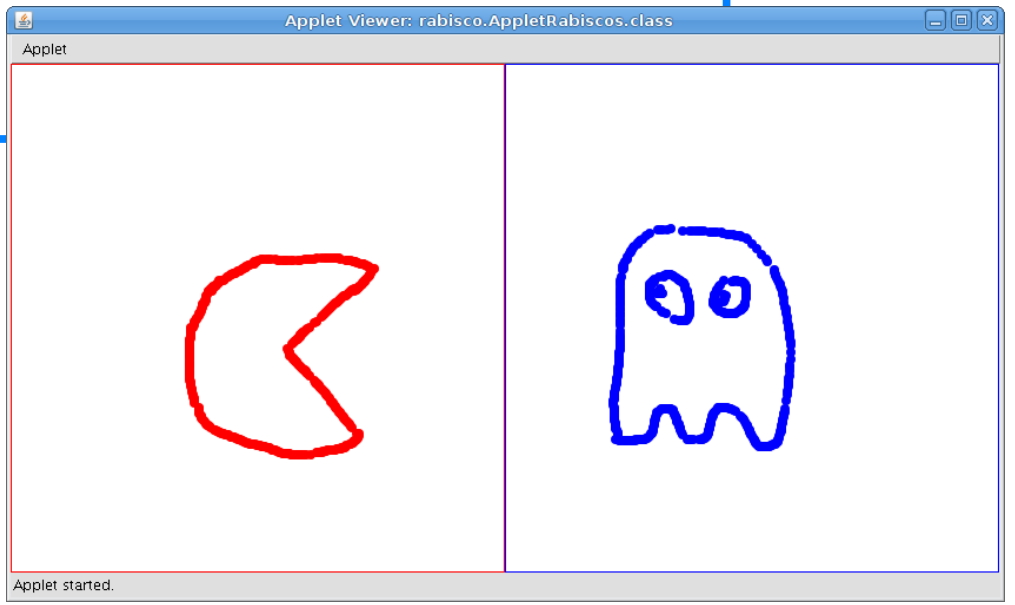
- Aplicações com interface gráfica que são executadas em um navegador.
- Mais seguras (para o cliente) do que aplicações.
- Menos flexíveis do que aplicações (*sandbox*).
- Herdam da classe JApplet.
- Têm métodos que podem ser sobreescritos, em particular:
  - init(): inicializa a *applet*.
  - paint(Graphics g): faz com que a *applet* seja pintada/desenhada.
- Tamanho depende da declaração no navegador.

Applets

```
package exemplos;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JApplet;

public class Applet1 extends JApplet
  {
  public void paint(Graphics g)
    {
    g.setColor(Color.BLUE);
    g.drawString("Olá, Mundo",5,15);
    }
  }
```

Applets

```
package rabisco;

import java.awt.*;
import javax.swing.*;

public class AppletRabiscos extends JApplet
  {
  public void init()
    {
    ComponenteParaRabiscos c1 = new ComponenteParaRabiscos(Color.RED);
    c1.setBorder(BorderFactory.createLineBorder(Color.RED));
    ComponenteParaRabiscos c2 = new ComponenteParaRabiscos(Color.BLUE);
    c2.setBorder(BorderFactory.createLineBorder(Color.BLUE));
    getContentPane().setLayout(new GridLayout(1,2));
    getContentPane().add(c1);
    getContentPane().add(c2);
    }
  }
```



Graphics/Graphics2D

- Vimos os métodos paintComponent (JComponents) e paint (JApplets), que recebem como argumento uma instância de Graphics.
- Graphics: contexto gráfico, sabe onde e como desenhar objetos gráficos.
  - Graphics2D: herda de Graphics, capacidades adicionais.
- Modificamos o comportamento através da a instância de Graphics/Graphics2D, usamos a instância para desenhar.

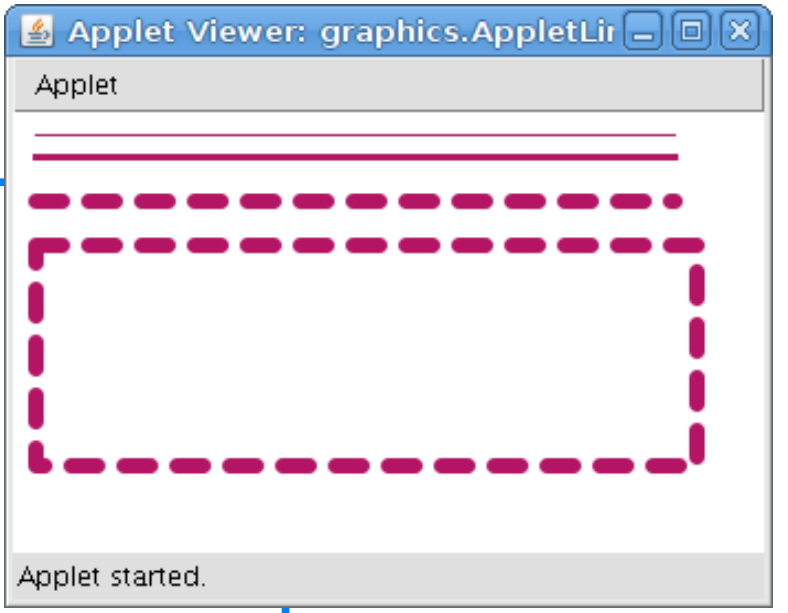
Graphics/Graphics2D

```
package graphics;

import java.awt.*;
import javax.swing.JApplet;

public class AppletTexto extends JApplet
  {
  public void paint(Graphics g)
    {
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setColor(Color.BLUE);
    g2d.setFont(new Font("SansSerif",Font.ITALIC,24));
    g2d.drawString("SansSerif",5,25);
    g2d.setFont(new Font("Serif",Font.ITALIC|Font.BOLD,36));
    g2d.drawString("Serif",5,65);
    g2d.setFont(new Font("Dialog",Font.PLAIN,48));
    g2d.drawString("Dialog",5,115);
    g2d.setFont(new Font("DialogInput",Font.PLAIN,48));
    g2d.drawString("DialogInput",5,175);
    }
  }
```

SansSerif SansSerif

Serif Serif

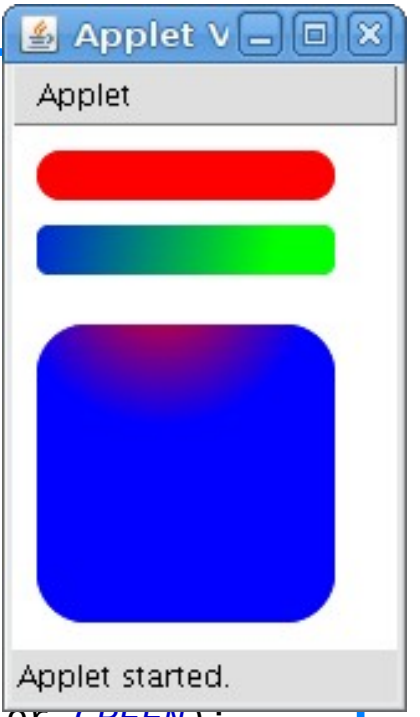
Graphics/Graphics2D




```
package graphics;

import java.awt.*;
import javax.swing.JApplet;

public class AppletLinhas extends JApplet
  {
  public void paint(Graphics g)
    {
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setColor(new Color(180,20,100));
    g2d.drawLine(10,10,300,10);
    g2d.setStroke(new BasicStroke(3f));
    g2d.drawLine(10,20,300,20);
    float[] dash = new float[]{12f};
    BasicStroke s = new BasicStroke(7f,BasicStroke.CAP_ROUND,
        BasicStroke.JOIN_ROUND,0f, dash, 0.0f);
    g2d.setStroke(s);
    g2d.drawLine(10,40,300,40);
    g2d.drawRect(10,60,300,100);
    }
  }
```

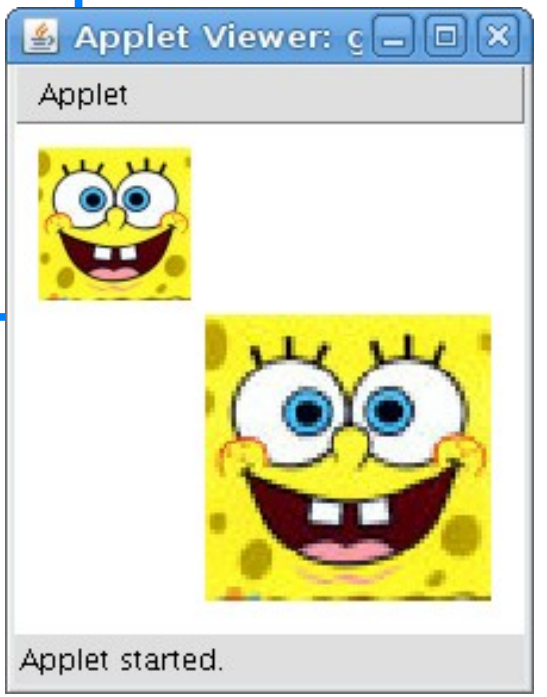
Graphics/Graphics2D

```
package graphics;

import java.awt.*;
import java.awt.geom.Point2D;
import javax.swing.JApplet;

public class AppletFills extends JApplet
  {
  public void paint(Graphics g)
    {
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setColor(Color.RED);
    g2d.fillRoundRect(10,10,120,20,20,20);
    GradientPaint degrade = new GradientPaint(0,0,Color.BLUE,110,20,Color.GREEN);
    g2d.setPaint(degrade);
    g2d.fillRoundRect(10,40,120,20,10,10);
    Point2D centro = new Point2D.Float(60,60);
    float raio = 60;
    float[] frações = {0f, 1f};
    Color[] cores = {Color.RED, Color.BLUE};
    RadialGradientPaint radial = new RadialGradientPaint(centro,raio,frações,cores);
    g2d.setPaint(radial);
    g2d.fillRoundRect(10,80,120,120,40,40);
    }
  }
```



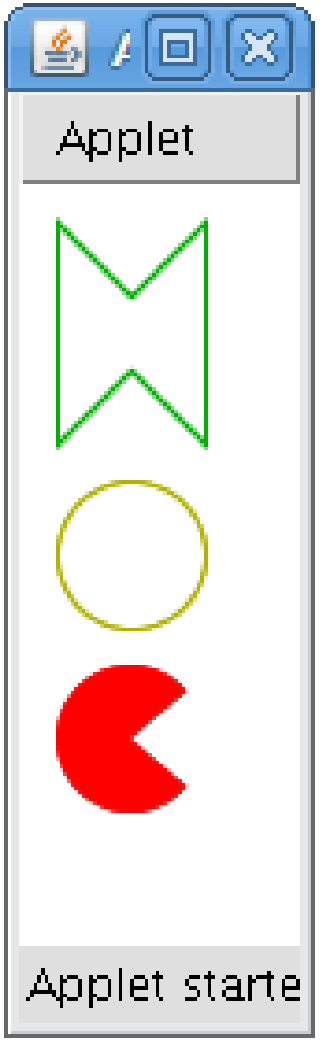
Graphics/Graphics2D

```
package graphics;

import java.awt.*;
import javax.swing.*;

public class AppletImage extends JApplet
  {
  private Image imagem;

  public void init()
    {
    imagem = getImage(getDocumentBase(),"Sprites/spongebob.jpg");
    }

  public void paint(Graphics g)
    {
    g.drawImage(imagem,10,10,this);
    g.drawImage(imagem,80,80,120,120,this);
    }
  }
```



Graphics/Graphics2D

```
package graphics;

import java.awt.*;
import javax.swing.JApplet;

public class AppletPolys extends JApplet
  {
  public void paint(Graphics g)
    {
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setColor(Color.GREEN.darker());
    int[] x = new int[]{10,30,50,50,30,10};
    int[] y = new int[]{10,30,10,70,50,70};
    Polygon poly = new Polygon(x,y,x.length);
    g2d.drawPolygon(poly);

    g2d.setColor(Color.YELLOW.darker());
    g2d.drawArc(10,80,40,40,0,360);

    g2d.setColor(Color.RED);
    g2d.fillArc(10,130,40,40,40,280);

    }
  }
```



Linhas de Execução (*Threads*)

- Em que ordem os métodos são executados?

```
package exemplos;

public class CarroDeCorrida
  {
  private String nome;
  private int distância;
  private int velocidade;
  public CarroDeCorrida(String n, int vel)
    {
    nome = n;
    distância = 0;
    velocidade = vel;
    }
  public void executa()
    {
    while (distância <= 1200)
      {
      System.out.println(nome + " rodou " + distância + " km.");
      distância += velocidade;
      // Causa um delay artificial.
      for (int sleep = 0; sleep < 1000000; sleep++)
        {  double x = Math.sqrt(Math.sqrt(Math.sqrt(sleep)));   }
      }
    }
  }
```

Linhas de Execução (*Threads*)

- Em que ordem os métodos são executados?

```
package exemplos;

public class SimulacaoSemThreads
  {
  public static void main(String[] args)
    {
    // Criamos instâncias da classe CarroDeCorrida.
    CarroDeCorrida penélope =
      new CarroDeCorrida("Penélope Charmosa",60);
    CarroDeCorrida dick =
      new CarroDeCorrida("Dick Vigarista",100);
    CarroDeCorrida quadrilha =
      new CarroDeCorrida("Quadrilha da Morte",120);
    // Criados os carros, vamos executar as simulações.
    penélope.executa();
    dick.executa();
    quadrilha.executa();
    }
  }
```

```
Penélope Charmosa rodou 0 km.
Penélope Charmosa rodou 60 km.
Penélope Charmosa rodou 120 km.
...
Penélope Charmosa rodou 1140 km.
Penélope Charmosa rodou 1200 km.
Dick Vigarista rodou 0 km.
Dick Vigarista rodou 100 km.
Dick Vigarista rodou 200 km.
...
Dick Vigarista rodou 1100 km.
Dick Vigarista rodou 1200 km.
Quadrilha da Morte rodou 0 km.
Quadrilha da Morte rodou 120 km.
...
Quadrilha da Morte rodou 1080 km.
Quadrilha da Morte rodou 1200 km.
```

Linhas de Execução (*Threads*)

- Código é executado na ordem explicitada dentro dos métodos.
- Métodos podem ser executados concorrentemente.
  - Ou com a aparência de concorrência em computadores com 1 CPU.
- Métodos podem ser executados concorrentemente em classes que herdam da classe Thread.
  - Estas classes devem implementar o método run (o método que será executado concorrentemente).

Linhas de Execução (*Threads*)

- Método mais simples de concorrência: classe herda de Thread e implementa o método run.
  - Criamos instâncias desta classe e executamos o método start das mesmas.
- Podemos agrupar *threads* (instâncias de herdeiras de Thread) em um ThreadGroup.
  - Fácil manipular grupos e verificar quantas *threads* estão ativas.

Linhas de Execução (*Threads*)

```
package exemplos;

public class CarroDeCorridaComThreads extends Thread
  {
  private String nome;
  private int distância;
  private int velocidade;
  public CarroDeCorridaComThreads(String n, int vel)
    {
    nome = n;
    distância = 0;
    velocidade = vel;
    }
  public void run()
    {
    while (distância <= 1200)
      {
      System.out.println(nome + " rodou " + distância + " km.");
      distância += velocidade;
      // Causa um delay artificial.
      try { Thread.sleep(4800/velocidade); }
      catch (InterruptedException e) { e.printStackTrace(); }
      }
    }
  }
```

Linhas de Execução (*Threads*)

```
package exemplos;

public class SimulacaoComThreads
  {
  public static void main(String[] args)
    {
    // Criamos instâncias da classe CarroDeCorridaComThreads.
    CarroDeCorridaComThreads penélope =
      new CarroDeCorridaComThreads("Penélope Charmosa",60);
    CarroDeCorridaComThreads dick =
      new CarroDeCorridaComThreads("Dick Vigarista",100);
    CarroDeCorridaComThreads quadrilha =
      new CarroDeCorridaComThreads("Quadrilha da Morte",120);
    // Criados os carros, vamos executar as simulações.
    penélope.start();
    dick.start();
    quadrilha.start();
    }
  }
```

```
Penélope Charmosa rodou 0 km.
Dick Vigarista rodou 0 km.
Quadrilha da Morte rodou 0 km.
Quadrilha da Morte rodou 120 km.
Dick Vigarista rodou 100 km.
...
Quadrilha da Morte rodou 1200 km.
Dick Vigarista rodou 900 km.
Penélope Charmosa rodou 360 km.
Dick Vigarista rodou 1000 km.
Dick Vigarista rodou 1100 km.
Penélope Charmosa rodou 420 km.
Dick Vigarista rodou 1200 km.
Penélope Charmosa rodou 480 km.
Penélope Charmosa rodou 540 km.
...
Penélope Charmosa rodou 1200 km.
```

Linhas de Execução (*Threads*)

- Herdar de Thread pode não ser viável!
  - Solução: implementar interface Runnable.
  - Usar uma instância local de Thread.

```
package exemplos;

public class CarroDeCorridaRunnable implements Runnable
  {
  private String nome;
  private int distância;
  private int velocidade;
  private Thread thread;

  public CarroDeCorridaRunnable(String n, int vel)
    {
    nome = n;
    distância = 0;
    velocidade = vel;
    }
```

Linhas de Execução (*Threads*)

- Precisamos de um método que crie a instância local de Thread e que execute start.

```
public void inicia()
  {
  if (thread == null)
    {
    thread = new Thread(this);
    thread.start();
    }
  }

public void run()
  {
  while (distância <= 1200)
    {
    System.out.println(nome + " rodou " + distância + " km.");
    distância += velocidade;
    // Causa um delay artificial.
    Try   { Thread.sleep(4800/velocidade); }
    catch (InterruptedException e)   { e.printStackTrace(); }
    }
  }
}
```

Linhas de Execução (*Threads*)

```
package exemplos;

public class SimulacaoComRunnable
  {
  // Este método permite a execução da classe.
  public static void main(String[] args)
    {
    // Criamos instâncias da classe CarroDeCorridaRunnable.
    CarroDeCorridaRunnable penÃ©lope =
      new CarroDeCorridaRunnable("Penélope Charmosa",60);
    CarroDeCorridaRunnable dick =
      new CarroDeCorridaRunnable("Dick Vigarista",100);
    CarroDeCorridaRunnable quadrilha =
      new CarroDeCorridaRunnable("Quadrilha da Morte",120);
    // Criados os carros, vamos executar as simulações.
    penélope.inicia();
    dick.inicia();
    quadrilha.inicia();
    }
  }
```

```
Penélope Charmosa rodou 0 km.
Dick Vigarista rodou 0 km.
Quadrilha da Morte rodou 0 km.
Quadrilha da Morte rodou 120 km.
...
Dick Vigarista rodou 1100 km.
Penélope Charmosa rodou 420 km.
Dick Vigarista rodou 1200 km.
Penélope Charmosa rodou 480 km.
Penélope Charmosa rodou 540 km.
...Penélope Charmosa rodou 1140 km.
Penélope Charmosa rodou 1200 km.
```

3 Jogos em Java

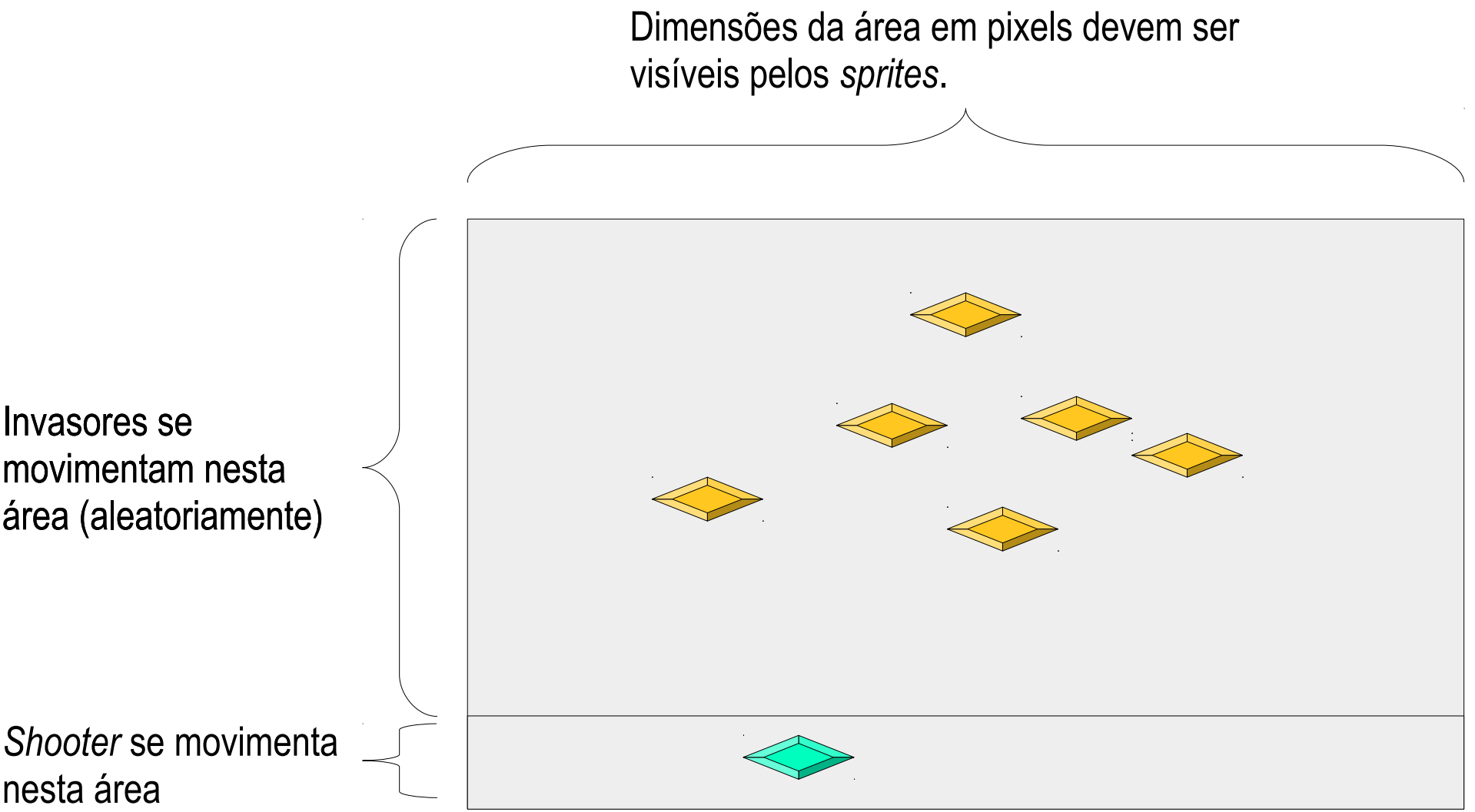
Jogo: *Invaders*

- *Shoot-em-up* simples: invasores voando, defensor atirando.
- **Exemplo incompleto**: o objetivo é demonstrar técnicas e dar idéias!
  - Nenhuma ênfase em arte gráfica!
- Código apresentado em iterações de desenvolvimento.

- Usaremos classes especializadas para objetos móveis (*sprites*).
- Usaremos um JPanel para controle do jogo e animação (veja o livro *Killer Game Programming in Java*).
- Uma aplicação conterá a instância do JPanel.

Jogo: *Invaders*

- *Layout* gráfico inicial:



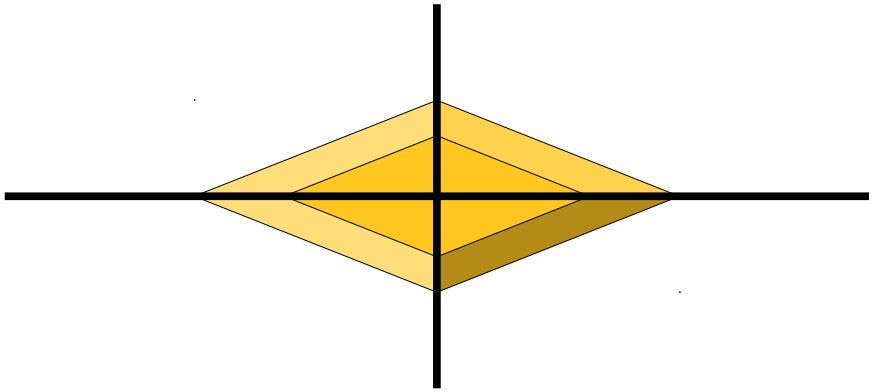
Jogo: *Invaders* (classe **Invader**)

- A classe Invader deve conter (entre outros):
  - Posição e velocidades;
  - Ícone para exibição;
  - Métodos para desenhar e mover.

```
package simpleinvaders0;

import java.awt.*;

import javax.swing.ImageIcon;

// Esta classe representa um invasor (UFO) no jogo.
public class Invader
  {
  // Posição e “velocidade” do UFO em pixels.
  private int x,y;
  private int dx,dy;
  // Tamanho do UFO em pixels.
  private int iw,ih;
  // Imagem do UFO.
  private Image icon;
  // área do painel do jogo (para controlar movimento).
  private Dimension area;
```

Jogo: *Invaders* (classe **Invader**)

```
// Construtor, inicializa atributos e posiciona o UFO.
public Invader(Dimension a)
  {
  area = a;
  icon = new ImageIcon(getClass().getResource("/Sprites/ufoa.png")).getImage();
  iw = icon.getWidth(null);
  ih = icon.getHeight(null);
  // x e y calculados usando a área do jogo.
  x = (int)(iw/2+Math.random()*(a.width-iw));
  y = (int)(ih/2+Math.random()*(a.height-100-ih));
  // dx e dy aleatórios.
  while(dx == 0 || dy == 0)
    {
    dx = 3-(int)(Math.random()*6);
    dy = 2-(int)(Math.random()*4);
    }
  }
```



Jogo: *Invaders* (classe **Invader**)

```
// Método que movimenta o UFO, verificando se está na área válida.
public void move()
  {
  x += dx;
  y += dy;
  if (x < iw/2) { dx = -dx; x += dx; }
  if (y < ih/2) { dy = -dy; y += dy; }
  if (x > area.width-iw/2) { dx = -dx; x += dx; }
  if (y > area.height-100-ih/2) { dy = -dy; y += dy; }
  }

// Método que desenha o UFO em um contexto gráfico.
public void draw(Graphics g)
  {
  g.drawImage(icon,x-iw/2,y-ih/2,null);
  }
```

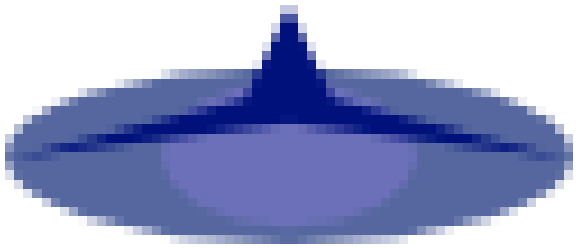
Jogo: *Invaders* (classe **Shooter**)

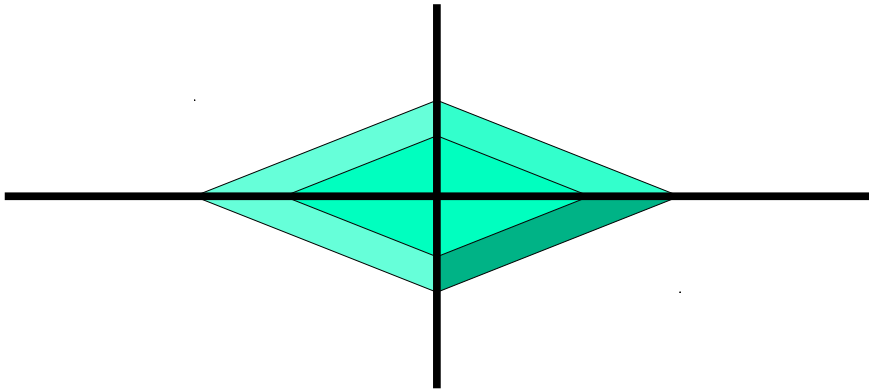
- A classe Shooter deve conter (entre outros):
  - Posição e direção;
  - Ícone para exibição;
  - Métodos para desenhar e mover.

```
package simpleinvaders0;

import java.awt.*;

import javax.swing.ImageIcon;

// Esta classe representa um defensor no jogo.
public class Shooter
  {
  // Posição do defensor em pixels.
  private int x,y;
  // Tamanho do defensor em pixels.
  private int iw,ih;
  // Imagem do defensor.
  private Image icon;
  // área do painel do jogo (para controlar movimento).
  private Dimension area;
```

Jogo: *Invaders* (classe **Shooter**)

```
// Construtor, inicializa atributos e posiciona o shooter.
public Shooter(Dimension a)
  {
  area = a;
  icon =
     new ImageIcon(getClass().getResource("/Sprites/shooter.png")).getImage();
  iw = icon.getWidth(null);
  ih = icon.getHeight(null);
  // x e y iniciais centrados na área de movimentação.
  x = (int)(iw/2+(a.width-iw)/2);
  y = (int)(a.height-100+ih/2);
  }
```





Jogo: *Invaders* (classe **Shooter**)

```
// Método que movimenta o shooter, verificando se está na área válida.
public void move(Direcao dir)
  {
  if (dir == null) return;
  switch(dir)
    {
    case LEFT:
      { x--; if (x < iw/2) x = iw/2; break; }
    case RIGHT:
      { x++; if (x > area.width-iw/2) x = area.width-iw/2; break; }
    case UP:
      { y--; if (y < area.height-100+ih/2) y = area.height-100+ih/2; break; }
    case DOWN:
      { y++; if (y > area.height-ih/2) y = area.height-ih/2; break; }
    }
  }

// Método que desenha o shooter em um contexto gráfico.
public void draw(Graphics g)
  {
  g.drawImage(icon,x-iw/2,y-ih/2,null);
  }
```

```
package simpleinvaders0;

public enum Direcao { UP,DOWN,LEFT,RIGHT }
```

Jogo: *Invaders* (classe **InvadersPanel**)

- A classe InvadersPanel deve conter (entre outros):
  - Lista de *sprites*;
  - *Thread* para execução;
  - Métodos para desenhar e mover.
  - Métodos para controle (interface) de jogo.

```
package simpleinvaders0;

import java.awt.*;
import java.awt.event.*;

import javax.swing.JPanel;

// Classe que implementa o painel do jogo. O painel controlará os elementos
// do jogo e a renderização.
public class InvadersPanel extends JPanel implements Runnable,KeyListener
  {
  // Dimensões da área do jogo.
  private static final int largura = 800;
  private static final int altura = 600;
```

Jogo: *Invaders* (classe **InvadersPanel**)

```
// Uma thread para controlar a animação.
private Thread animator;
private boolean isPaused = false;

// Teremos alguns UFOs passeando na tela.
private Invader[] invasores;
// O shooter e sua direção de movimento.
private Shooter shooter;
private Direcao dir;

// Construtor, inicializa estruturas, registra interfaces, etc.
public InvadersPanel()
  {
  setBackground(Color.WHITE);
  setPreferredSize(new Dimension(largura,altura));
  setFocusable(true);
  requestFocus();
  addKeyListener(this);
  invasores = new Invader[20];
  for(int i=0;i<invasores.length;i++)
    invasores[i] = new Invader(this.getPreferredSize());
  shooter = new Shooter(this.getPreferredSize());
  }
```

Jogo: *Invaders* (classe **InvadersPanel**)

```
// Avisa que agora temos a interface em um container parente.
public void addNotify()
  {
  super.addNotify();
  startGame();
  }
// Iniciamos o jogo (e a thread de controle)
private void startGame()
  {
  if (animator == null)
    {
    animator = new Thread(this);
    animator.start();
    }
  }
// Laço principal do jogo.
public void run()
  {
  while(true)
    {
    gameUpdate();
    repaint();
    try { Thread.sleep(10); }
    catch (InterruptedException e) { e.printStackTrace(); }
    }
  }
```

Jogo: *Invaders* (classe **InvadersPanel**)

```
// Atualizamos os elementos do jogo.
private void gameUpdate()
  {
  if (!isPaused)
    {
    for(Invader i:invasores) i.move();
    shooter.move(dir);
    }
  }

// Desenhamos o componente (e seus elementos)
protected void paintComponent(Graphics g)
  {
  super.paintComponent(g);
  g.setColor(Color.BLACK);
  g.drawRect(0,0,getWidth(),getHeight()-100);
  for(Invader i:invasores) i.draw(g);
  shooter.draw(g);
  }
```

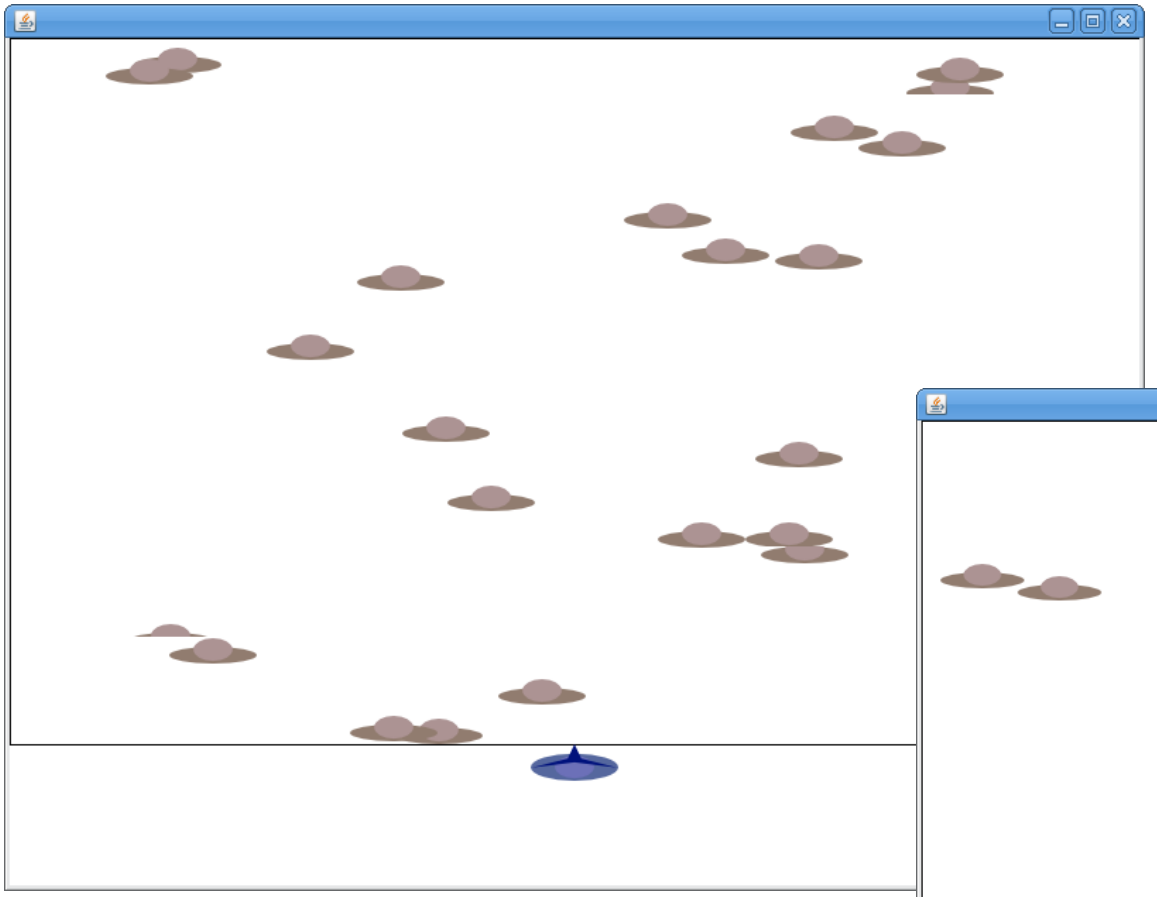
Jogo: *Invaders* (classe **InvadersPanel**)

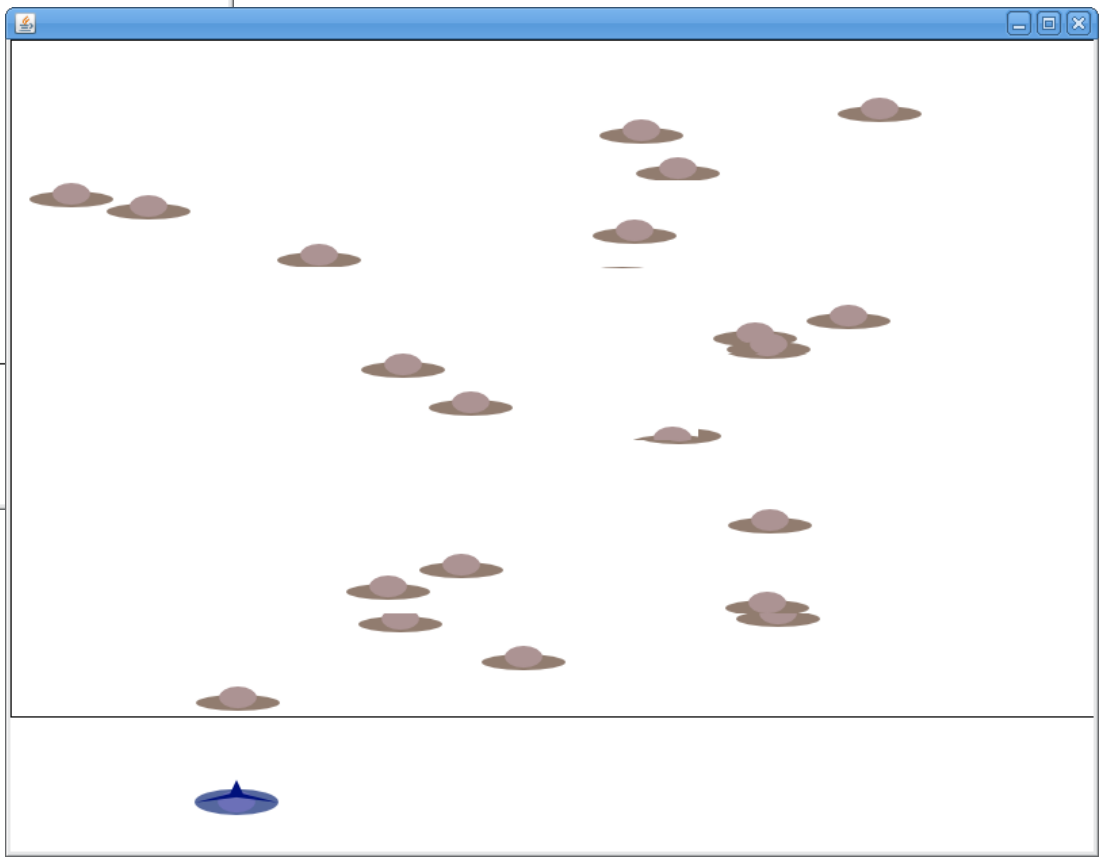
```
  // Processamos teclas pressionadas durante o jogo.
  public void keyPressed(KeyEvent e)
    {
    int keyCode = e.getKeyCode();
    if (keyCode == KeyEvent.VK_P) isPaused = !isPaused;
    if (isPaused) return;
    if (keyCode == KeyEvent.VK_LEFT) dir = Direcao.LEFT;
    else if (keyCode == KeyEvent.VK_RIGHT) dir = Direcao.RIGHT;
    else if (keyCode == KeyEvent.VK_UP) dir = Direcao.UP;
    else if (keyCode == KeyEvent.VK_DOWN) dir = Direcao.DOWN;
    }

  // Estes mÃ©todos servem para satisfazer a interface KeyListener
  public void keyReleased(KeyEvent e) { }

  public void keyTyped(KeyEvent e) { }
```

Jogo: *Invaders* (classe **InvadersApp**)

```
package simpleinvaders0;

import javax.swing.JFrame;

public class InvadersApp
  {
  public static void main(String[] args)
    {
    InvadersPanel sip = new InvadersPanel();
    JFrame frame = new JFrame();
    frame.getContentPane().add(sip);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
    }
  }
```

Jogo: *Invaders*





Jogo: *Invaders* (iteração 2)

- Nave atira, número de tiros limitado, elimina invasores.
- Interação entre algumas classes (*sprites*).
- Lista de *sprites* é modificada com o tempo!
  - Problemas de sincronização.
- Vamos ter que reescrever praticamente todas as classes...
  - ...mas são poucas modificações.

Jogo: *Invaders 2* (classe **Invader**)

- A classe Invader deve conter atributo estáAtivo (entre outros).

```
package simpleinvaders1;

import java.awt.*;

import javax.swing.ImageIcon;

// Esta classe representa um invasor (UFO) no jogo.
public class Invader
  {
  // Posição e "velocidade" do UFO em pixels.
  private int x,y;
  private int dx,dy;
  private boolean estáAtivo;
  // Tamanho do UFO em pixels.
  private int iw,ih;
  // Imagem do UFO.
  private Image icon;
  // área do painel do jogo (para controlar movimento).
  private Dimension area;
```

Jogo: *Invaders 2* (classe **Invader**)

```
// Construtor, inicializa atributos e posiciona o UFO.
public Invader(Dimension a)
  {
  area = a;
  icon = new ImageIcon(getClass().getResource("/Sprites/ufoa.png")).getImage();
  iw = icon.getWidth(null);
  ih = icon.getHeight(null);
  // x e y calculados usando a área do jogo.
  x = (int)(iw/2+Math.random()*(a.width-iw));
  y = (int)(ih/2+Math.random()*(a.height-100-ih));
  // dx e dy aleatórios.
  while(dx == 0 || dy == 0)
    { dx = 3-(int)(Math.random()*6);  dy = 2-(int)(Math.random()*4); }
  estáAtivo = true;
  }

// Método que movimenta o UFO, verificando se está na área válida.
public void move()
  {
  if (estáAtivo)
    {
    x += dx;         y += dy;
    if (x < iw/2) { dx = -dx; x += dx; }
    if (y < ih/2) { dy = -dy; y += dy; }
    if (x > area.width-iw/2) { dx = -dx; x += dx; }
    if (y > area.height-100-ih/2) { dy = -dy; y += dy; }
    }
  }
```

Jogo: *Invaders 2* (classe **Invader**)

```
// Método que desenha o UFO em um contexto gráfico.
public void draw(Graphics g)
  {
  if (estáAtivo) g.drawImage(icon,x-iw/2,y-ih/2,null);
  }

public void desativa()
  {
  estáAtivo = false;
  }

public boolean estáAtivo() { return estáAtivo; }

public int getX() { return x; }
public int getY() { return y; }
```

Jogo: *Invaders 2* (classe **Shooter**)

- A classe Shooter deve métodos para recuperar posição e tamanho (entre outros).

```
package simpleinvaders1;

import java.awt.*;
import javax.swing.ImageIcon;

// Esta classe representa um defensor no jogo.
public class Shooter
  {
  private int x,y;    // Posição do defensor em pixels.
  private int iw,ih;  // Tamanho do defensor em pixels.
  private Image icon; // Imagem do defensor.
  // área do painel do jogo (para controlar movimento).
  private Dimension area;

  // Construtor, inicializa atributos e posiciona o shooter.
  public Shooter(Dimension a)
    {
    area = a;
    icon = new ImageIcon(getClass().getResource("/Sprites/shooter.png")).getImage();
    iw = icon.getWidth(null);       ih = icon.getHeight(null);
    // x e y iniciais centrados na área de movimentação.
    x = (int)(iw/2+(a.width-iw)/2);       y = (int)(a.height-100+ih/2);
    }
```

Jogo: *Invaders 2* (classe **Shooter**)

```
// Método que movimenta o shooter, verificando se está na área válida.
public void move(Direcao dir)
  {
  if (dir == null) return;
  switch(dir)
    {
    case LEFT:  { x--; if (x < iw/2) x = iw/2; break; }
    case RIGHT: { x++; if (x > area.width-iw/2) x = area.width-iw/2; break; }
    case UP:  { y--; if (y < area.height-100+ih/2) y = area.height-100+ih/2; break; }
    case DOWN: { y++; if (y > area.height-ih/2) y = area.height-ih/2; break; }
    }
  }

// Método que desenha o shooter em um contexto gráfico.
public void draw(Graphics g)
  {
  g.drawImage(icon,x-iw/2,y-ih/2,null);
  }

public int getX() { return x; }
public int getY() { return y; }
public int getWidth() { return iw; }
public int getHeight() { return ih; }
```

```
package simpleinvaders1;

public enum Direcao { UP,DOWN,LEFT,RIGHT }
```

Jogo: *Invaders 2* (classe **Bullet**)

- A classe Bullet deve métodos para criar e desativar balas, movimentá-las e desenhá-las, recuperar posição e tamanho (entre outros).

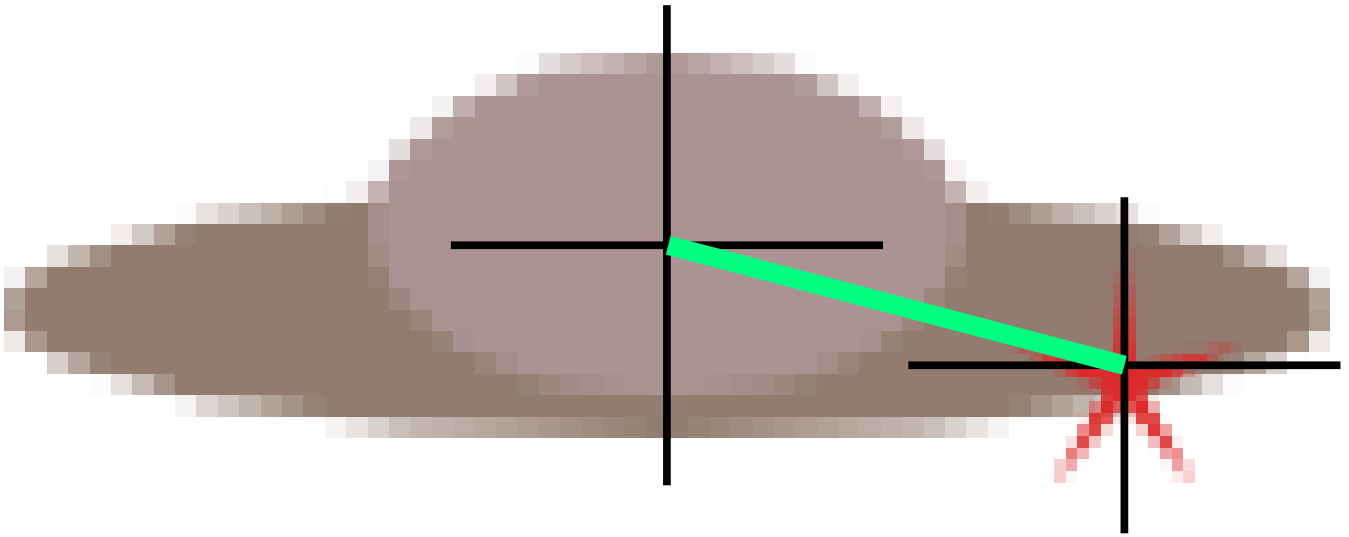
```
package simpleinvaders1;

import java.awt.*;

import javax.swing.ImageIcon;

// Esta classe representa um tiro no jogo.
public class Bullet
  {
  // Posição do tiro em pixels.
  private int x,y;
  // Este tiro está ativo?
  private boolean estáAtivo;
  // Tamanho do tiro em pixels.
  private int iw,ih;
  // Imagem do tiro.
  private Image icon;
  // área do painel do jogo (para controlar movimento).
  private Dimension area;
```

Jogo: *Invaders 2* (classe **Bullet**)

```
// Construtor, inicializa atributos, cria a bala.
public Bullet(Dimension a,Shooter s)
  {
  area = a;
  icon = new ImageIcon(getClass().getResource("/Sprites/bullet.png")).getImage();
  iw = icon.getWidth(null);
  ih = icon.getHeight(null);
  // x e y dependem do shooter
  x = s.getX();
  y = s.getY()-s.getHeight()/2;
  estáAtivo = true;
  }

// Método que movimenta a bala.
public void move()
  {
  if (!estáAtivo) return;
  y = y-3;
  if (y <= 0) estáAtivo = false;
  }

// Método que desenha a bala em um contexto gráfico.
public void draw(Graphics g)
  {
  if (estáAtivo) g.drawImage(icon,x-iw/2,y-ih/2,null);
  }
```

Jogo: *Invaders 2* (classe **Bullet**)

```
// Precisamos saber se esta bala está ativa!
public boolean estáAtivo() { return estáAtivo; }

// Verificamos se a bala está perto de um Invader
public boolean acertouEm(Invader i)
  {
  int ox = i.getX(); int oy = i.getY();
  if (Math.sqrt((x-ox)*(x-ox)+(y-oy)*(y-oy)) < 10)
    {
    estáAtivo = false;
    return true;
    }
  else return false;
  }
```



Jogo: *Invaders 2* (classe **InvadersPanel**)

- A classe InvadersPanel deve conter também (entre outros):
  - Gerenciamento de objetos que aparecem e somem.
  - Controle para tiro.

```
package simpleinvaders1;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.JPanel;

// Classe que implementa o painel do jogo. O painel controlará os elementos
// do jogo e a renderização.
public class InvadersPanel extends JPanel implements Runnable,KeyListener
  {
  // Dimensões da área do jogo.
  private static final int largura = 800;
  private static final int altura = 600;

  // Uma thread para controlar a animação.
  private Thread animator;
  private boolean isPaused = false;
```

Jogo: *Invaders 2* (classe **InvadersPanel**)

```
// Teremos alguns UFOs passeando na tela.
private ArrayList<Invader> invasores;
private static final int NUM_INVASORES = 20;
// O shooter e sua direção de movimento.
private Shooter shooter;
private Direcao dir;
// Um conjunto de tiros.
private ArrayList<Bullet> tiros;
private static final int MAX_TIROS = 5;

// Construtor, inicializa estruturas, registra interfaces, etc.
public InvadersPanel()
  {
  setBackground(Color.WHITE);
  setPreferredSize(new Dimension(largura,altura));
  setFocusable(true);
  requestFocus();
  addKeyListener(this);
  invasores = new ArrayList<Invader>(NUM_INVASORES);
  for(int i=0;i<NUM_INVASORES;i++)
    invasores.add(new Invader(this.getPreferredSize()));
  shooter = new Shooter(this.getPreferredSize());
  tiros = new ArrayList<Bullet>(MAX_TIROS);
  }
```

Jogo: *Invaders 2* (classe **InvadersPanel**)

```
// Avisa que agora temos a interface em um container parente.
public void addNotify()
  {
  super.addNotify();
  startGame();
  }
// Iniciamos o jogo (e a thread de controle)
private void startGame()
  {
  if (animator == null)
    {
    animator = new Thread(this);
    animator.start();
    }
  }
// Laço principal do jogo.
public void run()
  {
  while(true)
    {
    gameUpdate();
    repaint();
    try  {  Thread.sleep(10);  }
    catch (InterruptedException e)  { e.printStackTrace(); }
    }
  }
```

Jogo: *Invaders 2* (classe **InvadersPanel**)

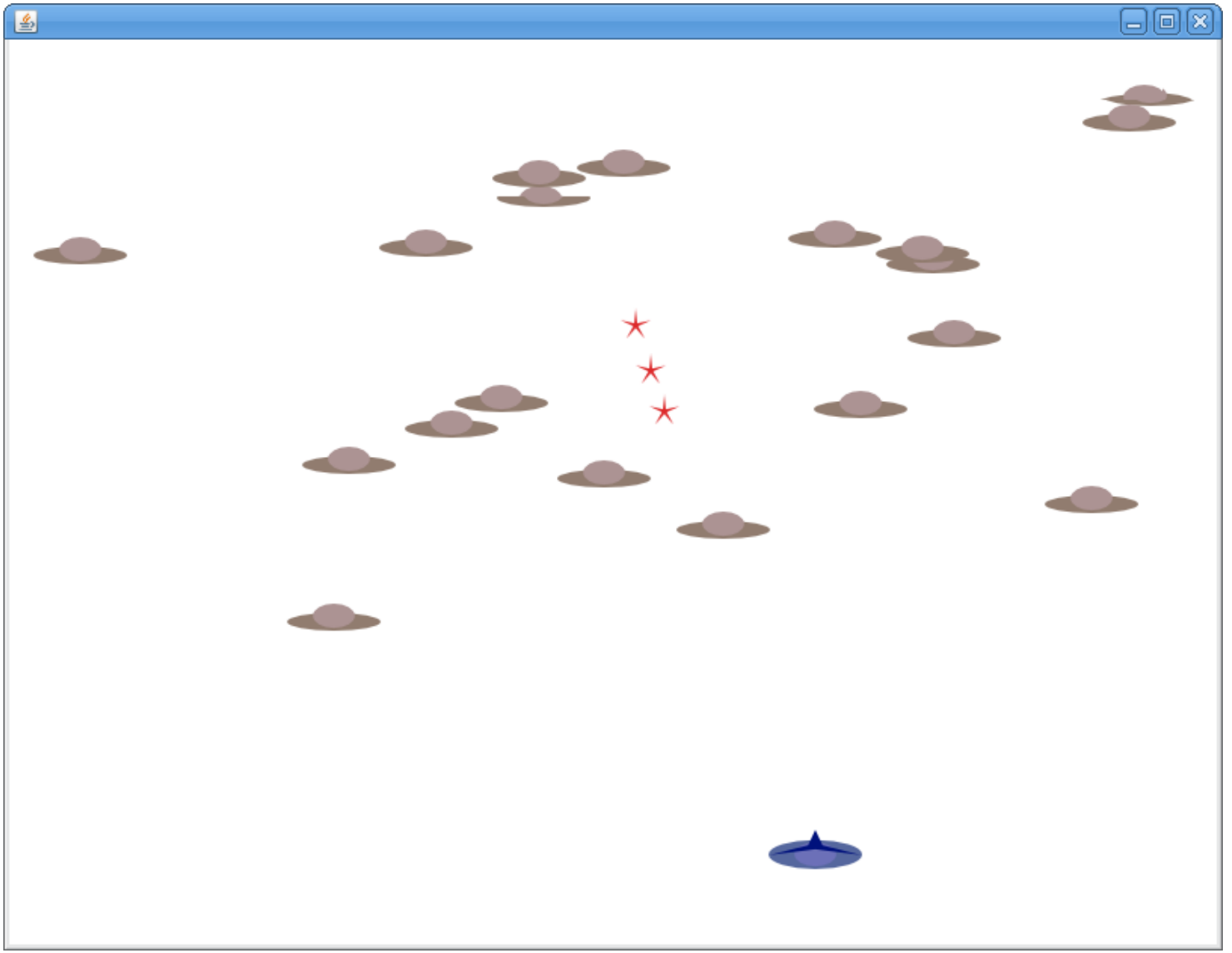
```
// Atualizamos os elementos do jogo.
private synchronized void gameUpdate()
  {
  if (!isPaused)
    {
    // Movemos os sprites.
    for(Invader i:invasores) i.move();
    shooter.move(dir);
    for(Bullet b:tiros) b.move();
    // Temos balas desativadas?
    // Não usar [for(Bullet b:tiros) { if (!b.estáAtivo()) tiros.remove(b); }]!
    Iterator<Bullet> it = tiros.iterator();
    while (it.hasNext()) { if (!(it.next()).estáAtivo()) it.remove(); }
    // Temos invasores desativados?
    Iterator<Invader> ii = invasores.iterator();
    while (ii.hasNext()) { if (!(ii.next()).estáAtivo()) ii.remove(); }
    // Temos colisões?
    for(Bullet b:tiros)
      for(Invader i:invasores)
        if (b.acertouEm(i)) i.desativa();
    }
  }
```

Jogo: *Invaders 2* (classe **InvadersPanel**)

```
// Desenhamos o componente (e seus elementos)
protected synchronized void paintComponent(Graphics g)
  {
  super.paintComponent(g);
  for(Invader i:invasores) i.draw(g);
  shooter.draw(g);
  for(Bullet b:tiros) b.draw(g);
  }

// Processamos teclas pressionadas durante o jogo.
public void keyPressed(KeyEvent e)
  {
  int keyCode = e.getKeyCode();
  if (keyCode == KeyEvent.VK_P) isPaused = !isPaused;
  if (isPaused) return;
  if (keyCode == KeyEvent.VK_LEFT) dir = Direcao.LEFT;
  else if (keyCode == KeyEvent.VK_RIGHT) dir = Direcao.RIGHT;
  else if (keyCode == KeyEvent.VK_UP) dir = Direcao.UP;
  else if (keyCode == KeyEvent.VK_DOWN) dir = Direcao.DOWN;
  else if (keyCode == KeyEvent.VK_SPACE)
    {
    if (tiros.size() < MAX_TIROS) // podemos adicionar mais tiros
      {    tiros.add(new Bullet(getPreferredSize(),shooter));   }
    }
  }
// Estes métodos servem para satisfazer a interface KeyListener
public void keyReleased(KeyEvent e) { }
public void keyTyped(KeyEvent e) { }
```

Jogo: *Invaders 2* (classe **InvadersApp**)

- Sem modificações.



Jogo: *Invaders* (iteração 3)

- Nave atira bombas, uma de cada vez, número limitado.
- Pequenas (muito pequenas!) melhorias na interface gráfica.
- Vamos ter que reescrever algumas classes...
  - ...mas são poucas modificações.
- Classes/Interfaces/Enums que não mudam:
  - Invader
  - Shooter
  - Direcao
  - InvadersApp

Jogo: *Invaders 3* (classe **Bullet**)

- Classe Bullet: agora tem direção, mudamos o construtor.

```
  // Posição do tiro em pixels.
  private int x,y;
  // Direção do tiro.
  private Direcao direção;
...

  // Construtor, inicializa atributos, cria a bala.
  public Bullet(Dimension a,int x,int y,Direcao dir)
    {
    area = a;
    icon = new ImageIcon(getClass().getResource("/Sprites/bullet.png")).getImage();
    iw = icon.getWidth(null);
    ih = icon.getHeight(null);
    // x e y passados direto como argumentos
    this.x = x;
    this.y = y;
    direção = dir;
    estáAtivo = true;
    }
```

Jogo: *Invaders 3* (classe **Bullet**)

- Classe Bullet: mudamos o método move().

```
// Método que movimenta a bala (em quatro possíveis direções).
public void move()
  {
  if (!estáAtivo) return;
  switch(direção)
    {
    case LEFT:
      {
      x -= 3; if (x < 0) estáAtivo = false; break;
      }
    case RIGHT:
      {
      x += 3; if (x > area.width) estáAtivo = false; break;
      }
    case UP:
      {
      y -= 3; if (y < 0) estáAtivo = false; break;
      }
    case DOWN:
      {
      y += 3; if (y > area.height-100) estáAtivo = false; break;
      }
    }
  }
```

Jogo: *Invaders 3* (classe **Bomb**)

- Classe Bomb: Baseada em Bullet.

```
package simpleinvaders2;

import java.awt.*;
import java.util.ArrayList;

import javax.swing.ImageIcon;

// Esta classe representa uma bomba no jogo.
public class Bomb
  {
  // Posição da bomba em pixels.
  private int x,y;
  // Esta bomba está ativa?
  private boolean estáAtivo;
  // Tamanho da bomba em pixels.
  private int iw,ih;
  // Imagem da bomba.
  private Image icon;
  // área do painel do jogo (para controlar movimento).
  private Dimension area;
```

Jogo: *Invaders 3* (classe **Bomb**)

```
// Construtor, inicializa atributos, cria a bomba.
public Bomb(Dimension a,int x,int y)
  {
  area = a;
  icon = new ImageIcon(getClass().getResource("/Sprites/bomb.png")).getImage();
  iw = icon.getWidth(null);
  ih = icon.getHeight(null);
  // x e y passadas diretamente como parâmetros
  this.x = x;
  this.y = y;
  estáAtivo = true;
  }
// Método que movimenta a bomba.
public void move()
  {
  if (!estáAtivo) return;
  y = y-3;
  if (y <= 0) estáAtivo = false;
  }
// Método que desenha a bomba em um contexto gráfico.
public void draw(Graphics g)
  {
  if (estáAtivo) g.drawImage(icon,x-iw/2,y-ih/2,null);
  }
// Precisamos saber se esta bomba está ativa!
public boolean estáAtivo() { return estáAtivo; }
```

Jogo: *Invaders 3* (classe **Bomb**)

```
// Verificamos se a bomba está perto de um Invader
public boolean acertouEm(Invader i)
  {
  int ox = i.getX(); int oy = i.getY();
  return (Math.sqrt((x-ox)*(x-ox)+(y-oy)*(y-oy)) < 25);
  }

// Explodimos a bomba (retornando bullets).
public ArrayList<Bullet> explode()
  {
  ArrayList<Bullet> novasBalas = new ArrayList<Bullet>(4);
  novasBalas.add(new Bullet(area,x,y,Direcao.LEFT));
  novasBalas.add(new Bullet(area,x,y,Direcao.RIGHT));
  novasBalas.add(new Bullet(area,x,y,Direcao.UP));
  novasBalas.add(new Bullet(area,x,y,Direcao.DOWN));
  estáAtivo = false;
  return novasBalas;
  }
```

Jogo: *Invaders 3* (classe **InvadersPanel**)

- Classe InvadersPanel ampliada.

```
package simpleinvaders2;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.JPanel;

public class InvadersPanel extends JPanel implements Runnable, KeyListener
  {
  private static final int largura = 800;
  private static final int altura = 600;
  // Uma thread para controlar a animação.
  private Thread animator;
  private boolean isPaused = false;
  // Teremos alguns UFOs passeando na tela.
  private ArrayList<Invader> invasores;
  private static final int NUM_INVASORES = 20;
  // O shooter e sua direção de movimento.
  private Shooter shooter;
  private Direcao dir;
  // Um conjunto de tiros.
  private ArrayList<Bullet> tiros;
  private static final int MAX_TIROS = 5;
  // Uma única bomba.
  private Bomb bomba;
  private int numBombas;
  private static final int NUM_BOMBAS_DISPONIVEL = 10;
```

Jogo: *Invaders 3* (classe **InvadersPanel**)

```
// Construtor, inicializa estruturas, registra interfaces, etc.
public InvadersPanel()
  {
  setBackground(Color.WHITE);
  setPreferredSize(new Dimension(largura,altura));
  setFocusable(true);    requestFocus();
  addKeyListener(this);
  invasores = new ArrayList<Invader>(NUM_INVASORES);
  for(int i=0;i<NUM_INVASORES;i++)
     invasores.add(new Invader(this.getPreferredSize()));
  shooter = new Shooter(this.getPreferredSize());
  tiros = new ArrayList<Bullet>(MAX_TIROS);
  bomba = null;
  numBombas = NUM_BOMBAS_DISPONIVEL;
  }

// Avisa que agora temos a interface em um container parente.
public void addNotify()
  {
  super.addNotify();
  startGame();
  }

// Iniciamos o jogo (e a thread de controle)
private void startGame()
  {
  if (animator == null) { animator = new Thread(this); animator.start(); }
  }
```

Jogo: *Invaders 3* (classe **InvadersPanel**)

```
// Atualizamos os elementos do jogo.
private synchronized void gameUpdate()
  {
  if (!isPaused)
    {
    // Movemos os sprites.
    for(Invader i:invasores) i.move();
    shooter.move(dir);
    for(Bullet b:tiros) b.move();
    if (bomba != null) bomba.move();
    // Temos balas desativadas?
    Iterator<Bullet> it = tiros.iterator();
    while (it.hasNext()) { if (!(it.next()).estáAtivo()) it.remove(); }
    // Temos invasores desativados?
    Iterator<Invader> ii = invasores.iterator();
    while (ii.hasNext()) { if (!(ii.next()).estáAtivo()) ii.remove(); }
    // A bomba está desativada?
    if (bomba != null) if (!bomba.estáAtivo()) bomba = null;
    // Temos colisões com balas?
    for(Bullet b:tiros)
      for(Invader i:invasores)
        if (b.acertouEm(i)) i.desativa();
    // Temos colisões com bombas?
    if (bomba != null)
      for(Invader i:invasores)
        if (bomba.acertouEm(i)) { tiros.addAll(bomba.explode()); i.desativa(); }
    }
  }
```

Jogo: *Invaders 3* (classe **InvadersPanel**)

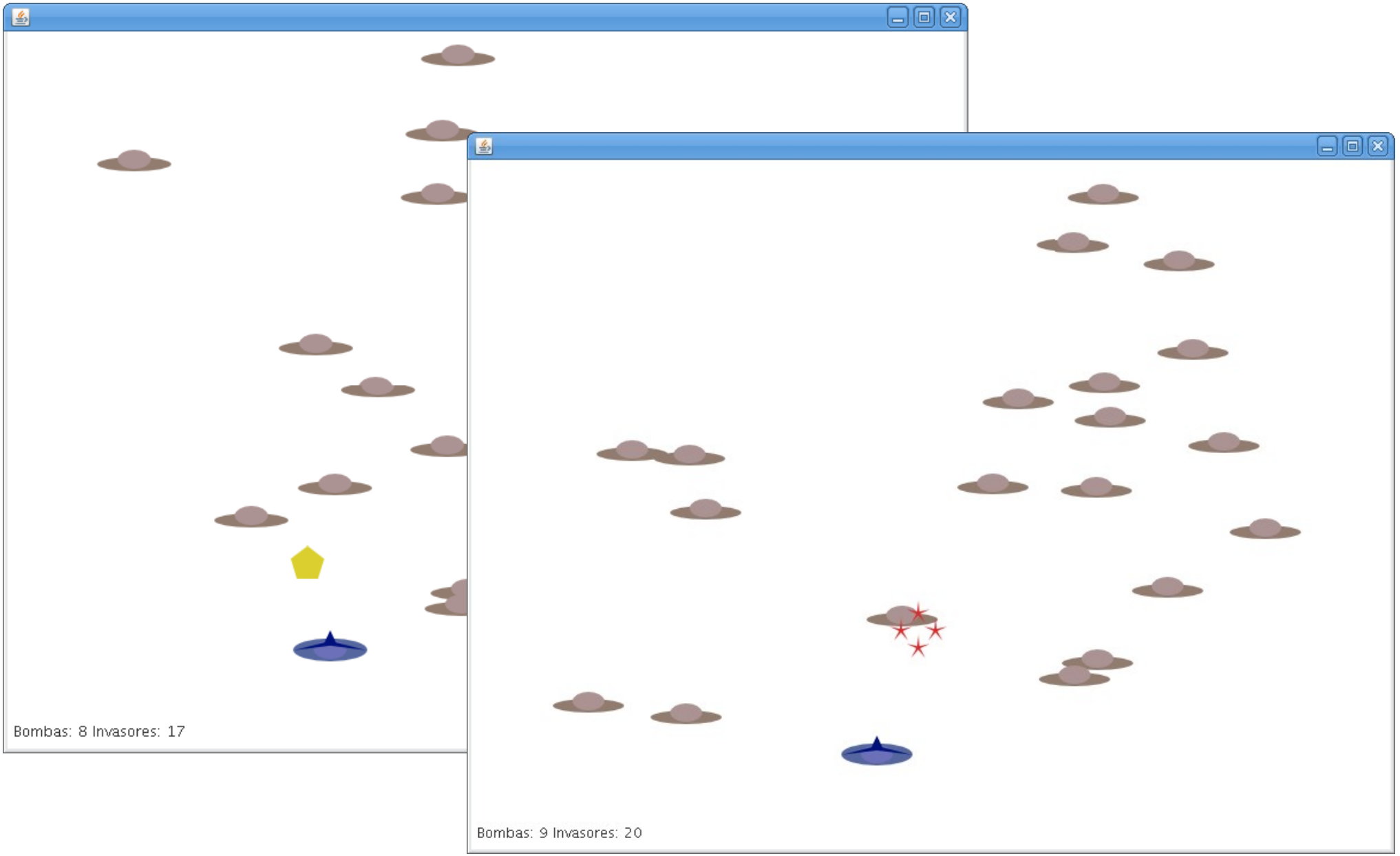
```
// Laço principal do jogo.
public void run()
  {
  while(true)
    {
    gameUpdate();
    repaint();
    try  { Thread.sleep(10); }
    catch (InterruptedException e) { e.printStackTrace(); }
    }
  }

// Desenhamos o componente (e seus elementos)
protected synchronized void paintComponent(Graphics g)
  {
  super.paintComponent(g);
  for(Invader i:invasores) i.draw(g);
  shooter.draw(g);
  for(Bullet b:tiros) b.draw(g);
  if (bomba != null) bomba.draw(g);
  // Pintamos estatísticas
  String s = "Bombas: "+numBombas+" Invasores: "+invasores.size();
  g.drawString(s,5,getHeight()-10);
  }
```

Jogo: *Invaders 3* (classe **InvadersPanel**)

```
  public void keyPressed(KeyEvent e)
    {
    int keyCode = e.getKeyCode();
    if (keyCode == KeyEvent.VK_P) isPaused = !isPaused;
    if (isPaused) return;
    if (keyCode == KeyEvent.VK_LEFT) dir = Direcao.LEFT;
    else if (keyCode == KeyEvent.VK_RIGHT) dir = Direcao.RIGHT;
    else if (keyCode == KeyEvent.VK_UP) dir = Direcao.UP;
    else if (keyCode == KeyEvent.VK_DOWN) dir = Direcao.DOWN;
    else if (keyCode == KeyEvent.VK_SPACE)
      {
      if (tiros.size() < MAX_TIROS) // podemos adicionar mais tiros
        {
        tiros.add(new Bullet(getPreferredSize(),
                  shooter.getX(),shooter.getY()-shooter.getHeight()/2,Direcao.Up));
        }
      }
    else if (keyCode == KeyEvent.VK_B)
      {
      if (numBombas > 0) // atiramos bombas
        {
        if (bomba == null) // só se não houver bomba ativa!
          {
          bomba = new Bomb(getPreferredSize(),
                           shooter.getX(),shooter.getY()-shooter.getHeight()/2);
          numBombas--;
          }
        }
      }
    }
```

```
  public void keyReleased(KeyEvent e) { }
  public void keyTyped(KeyEvent e) { }
```

Jogo: *Invaders 3*



Jogo: Exercícios sobre *Invaders*

- Como implementar...
  - Algum esquema de pontuação?
  - Inimigos atirando?
  - *Game Over*?
  - Teleporte da nave (com número de usos)?
  - Fases (com invasores diferentes)?
  - Dificuldades em fases?
  - Poderes adicionais (capturados pelo *shooter*)?
  - *Bosses*?
- Sugestão: fazer uma crítica do jogo visando melhorias.
  - Comparar com clássicos, pensar em novidades.
  - Evitar comparação com jogos profissionais (times de desenvolvedores, meses para desenvolver).

Jogo: Exercícios sobre *Invaders*

- Aprendendo OO: Vale a pena reorganizar classes que representam *sprites*? Como fazer isto?
- Criar um JPanel específico para este tipo de jogo?
- Como resolver problemas potenciais com Thread.sleep()?
  - Número de objetos pode influenciar na velocidade do jogo.
  - Solução (complexa): *Killer Game Programming in Java*.
- Como armazenar o estado do jogo?
  - Serialização!
- Como permitir jogo entre dois jogadores?
  - Controle via rede!

Jogo: Exercícios em geral

- Visite um dos inúmeros sites com jogos em Flash. Veja um com lógica e matemática/física simples. Implemente uma versão feia em Java.
- Vasculhe http://www.java4k.com/ para idéias, tente melhorar os jogos (ignorando o limite de 4K).

4 Extras

Showcase

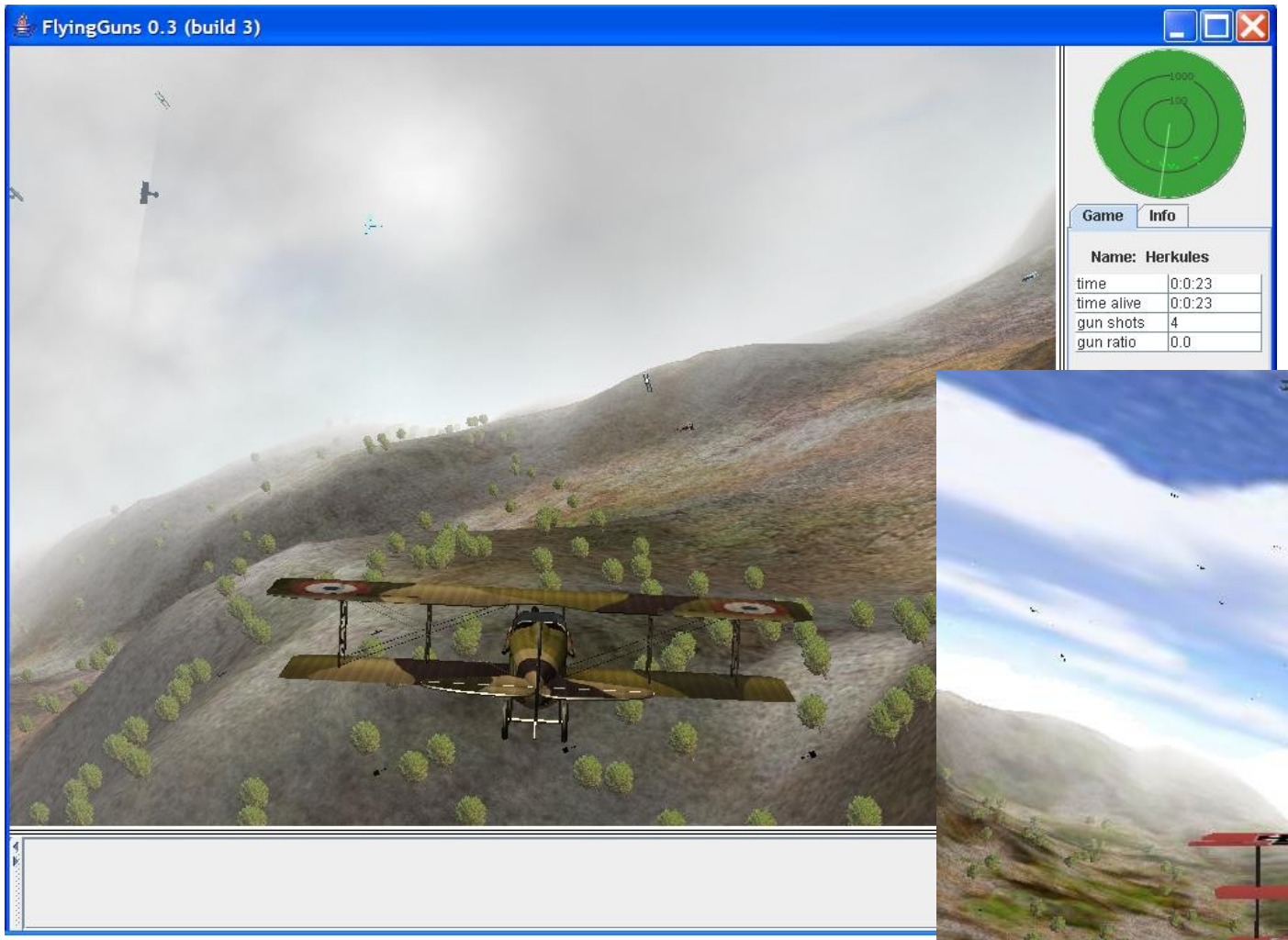
Puzzle Pirates, http://www.puzzlepirates.com

Showcase

Kingdoms of War, http://www.abandonedcastle.com

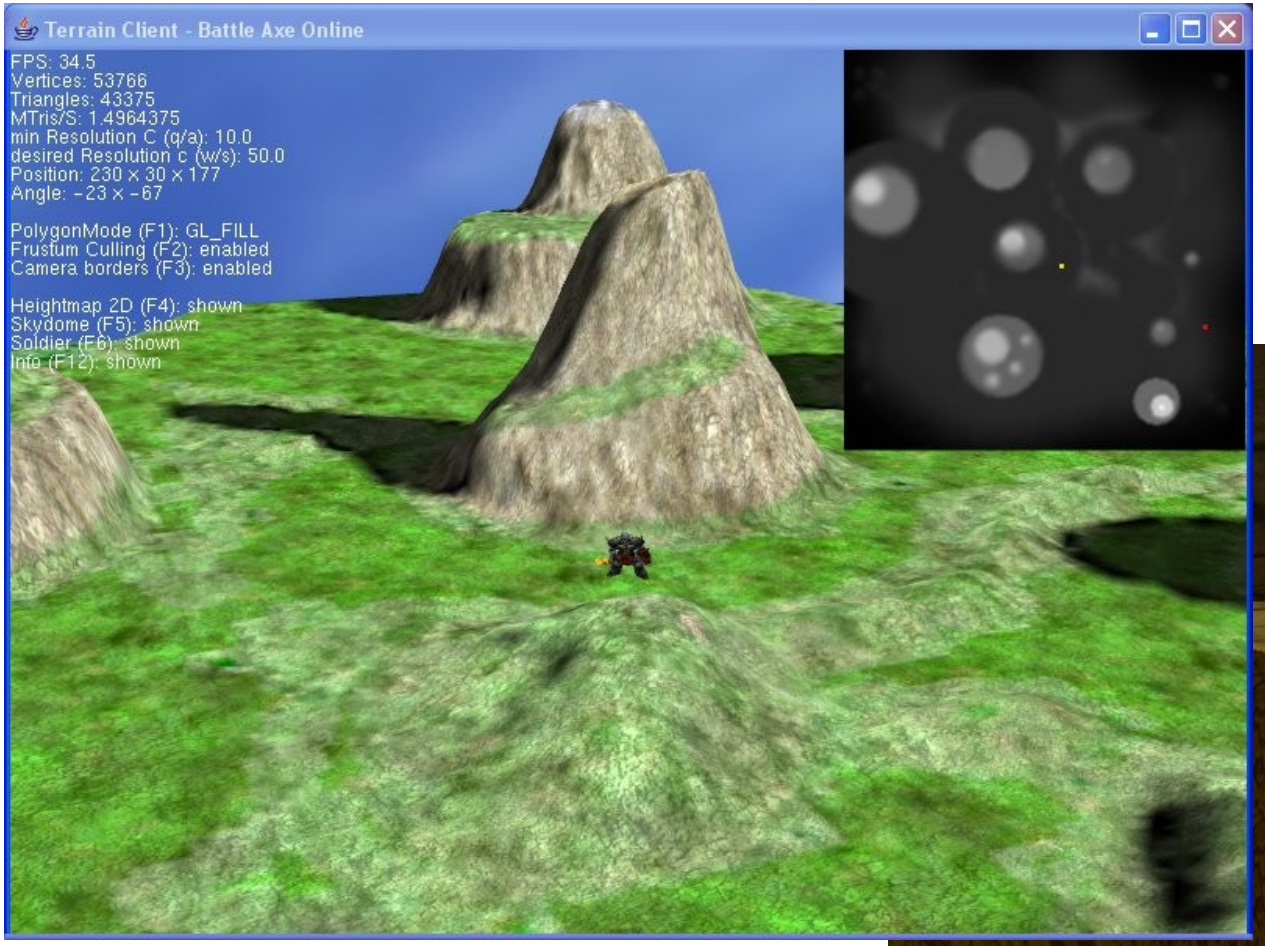
Showcase

FlyingGuns, http://www.flyingguns.com



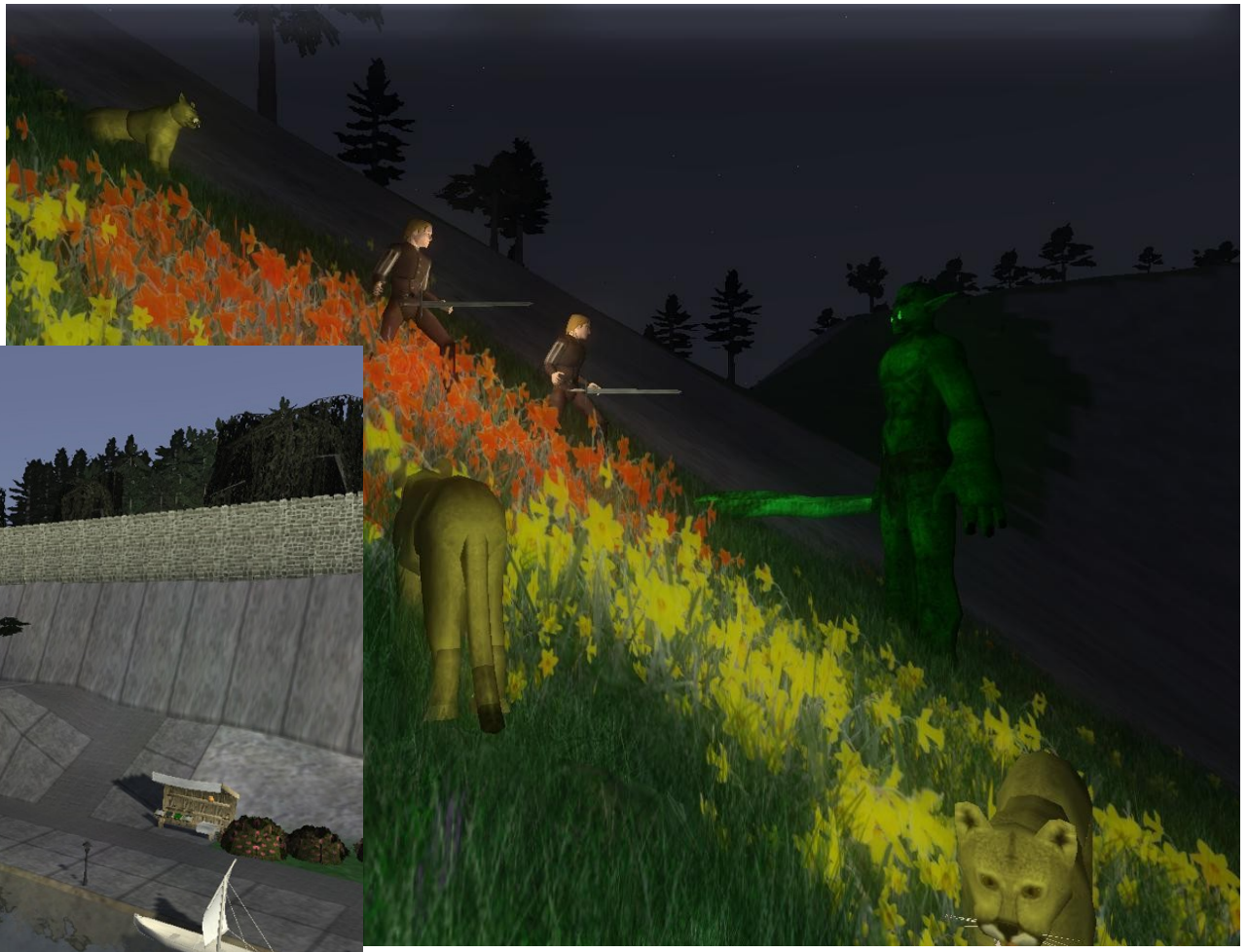
Showcase

Escape, http://javaisdoomed.sourceforge.net



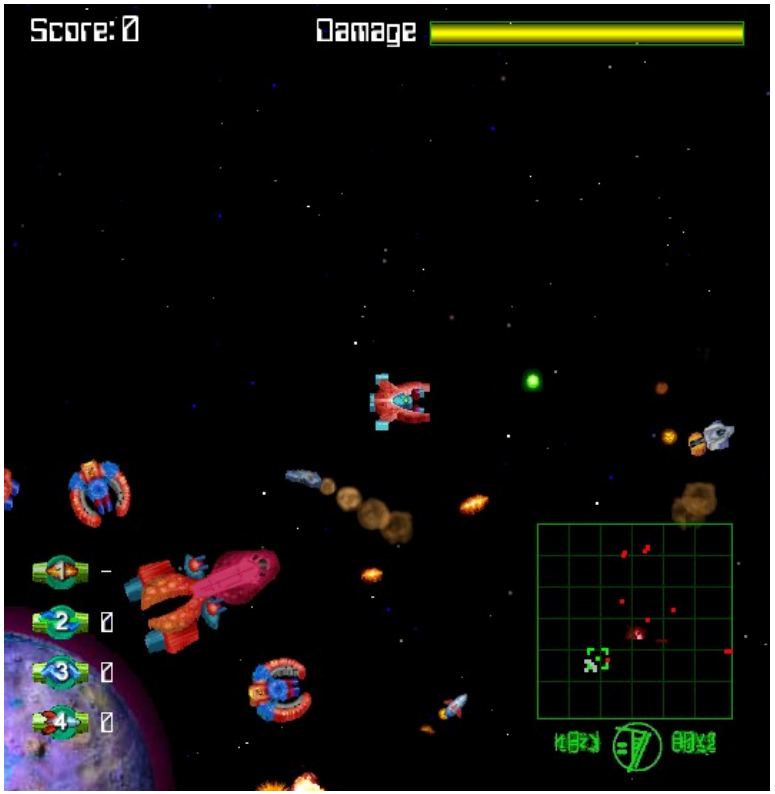
Showcase

Wurm Online, http://wurmonline.com/



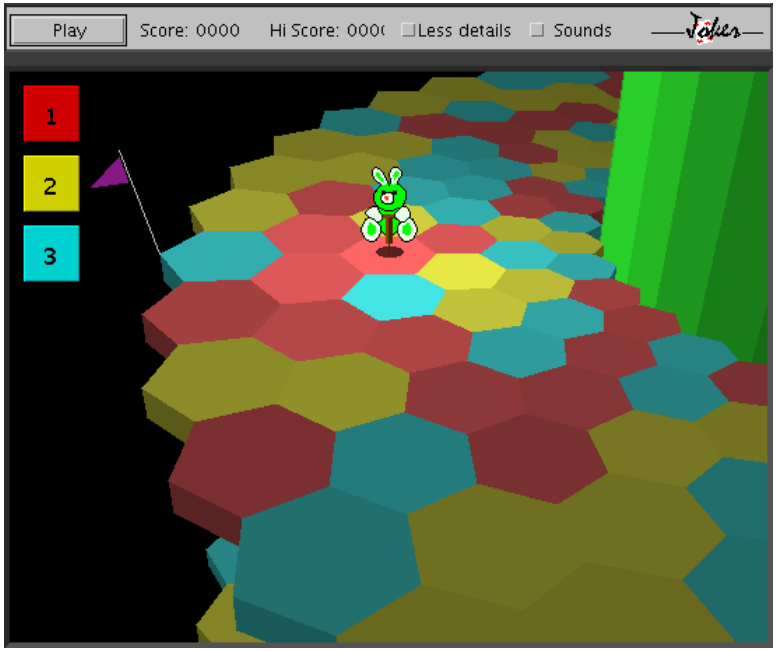
Showcase

http://www.arcadepod.com/java



Real Space 2

Solitaire

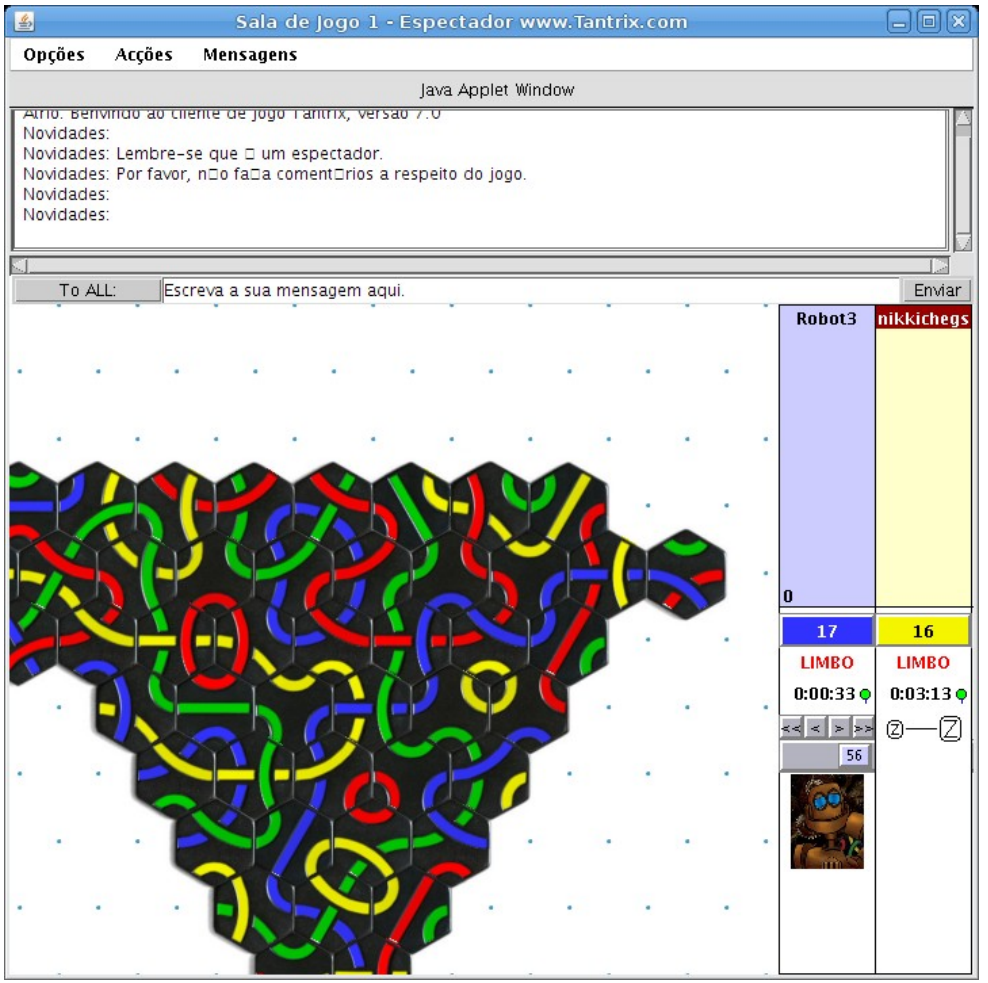


Think Tint

Showcase

http://www.arcadepod.com/java

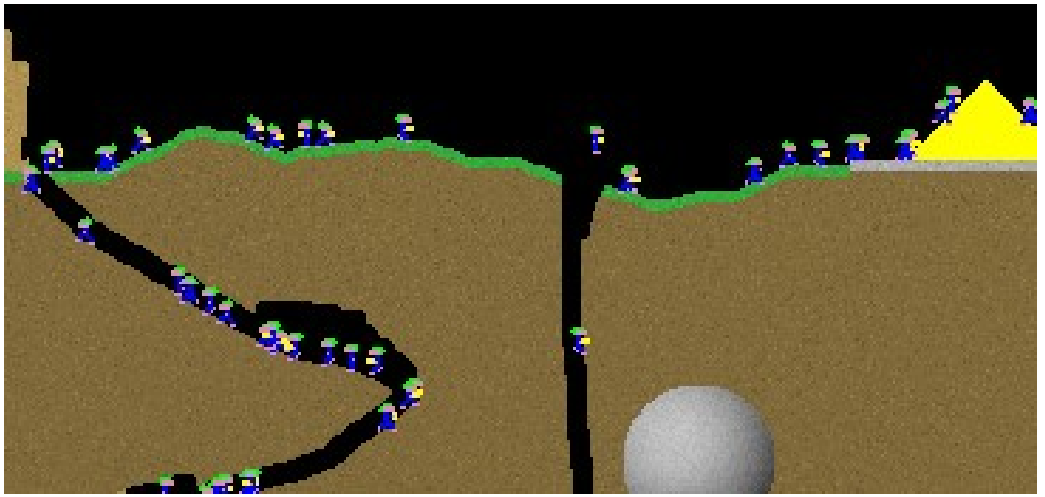
Realms of Rivalry



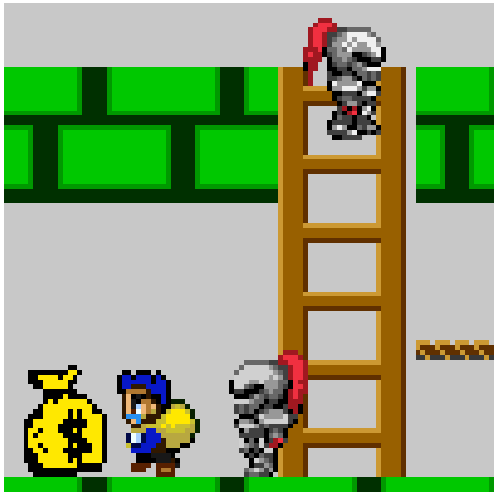
Tantrix

Showcase

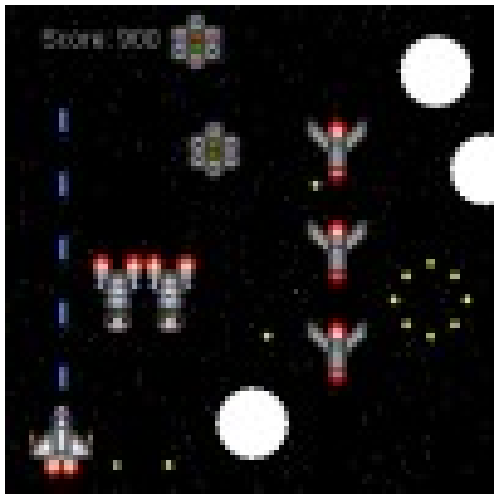
http://www.java4k.com



Miners4K

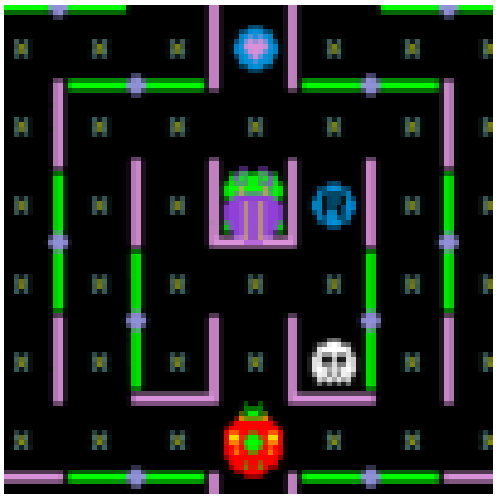


Lode Runner 4k



Xero

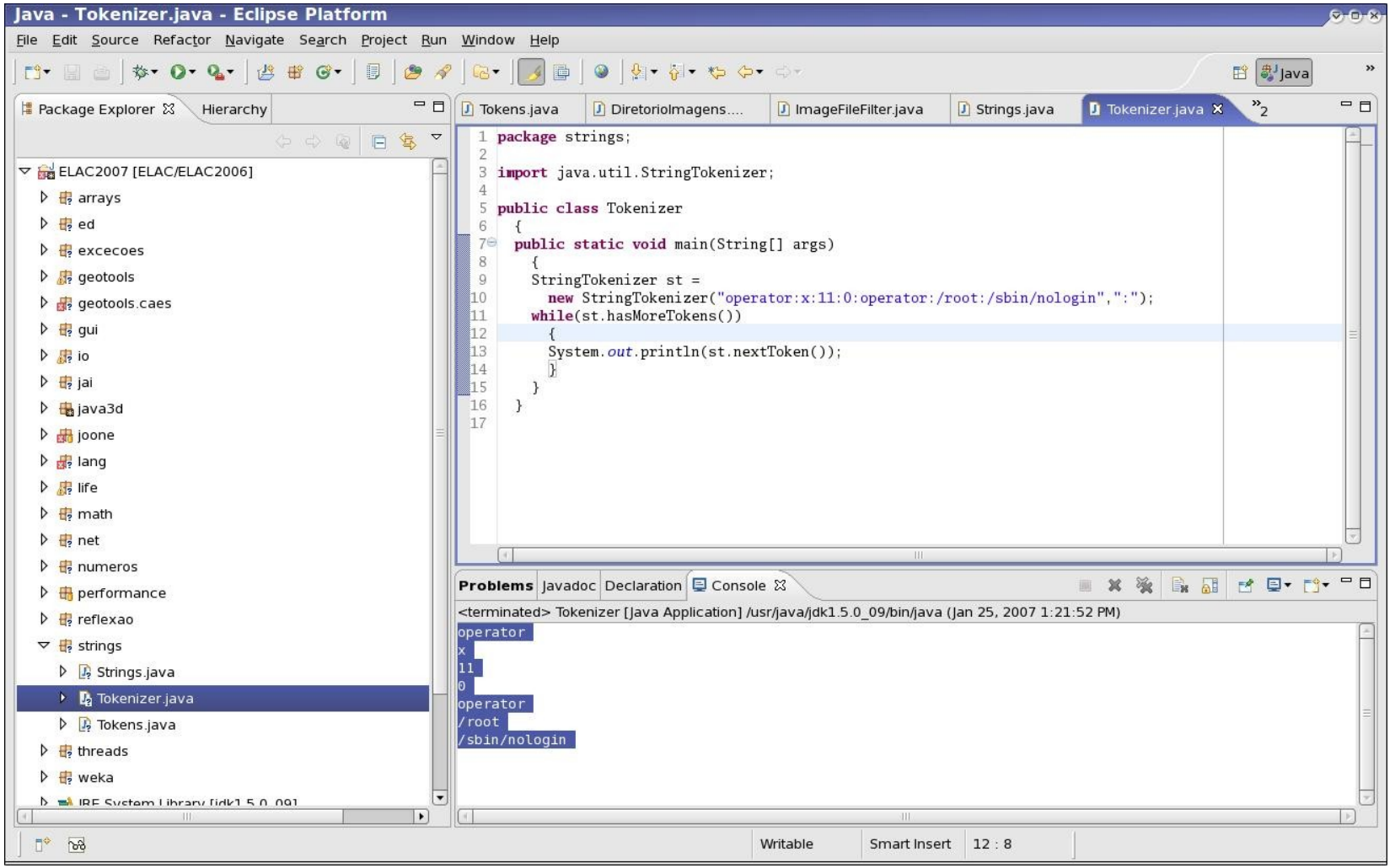
Cubis



Lady Bug 4k

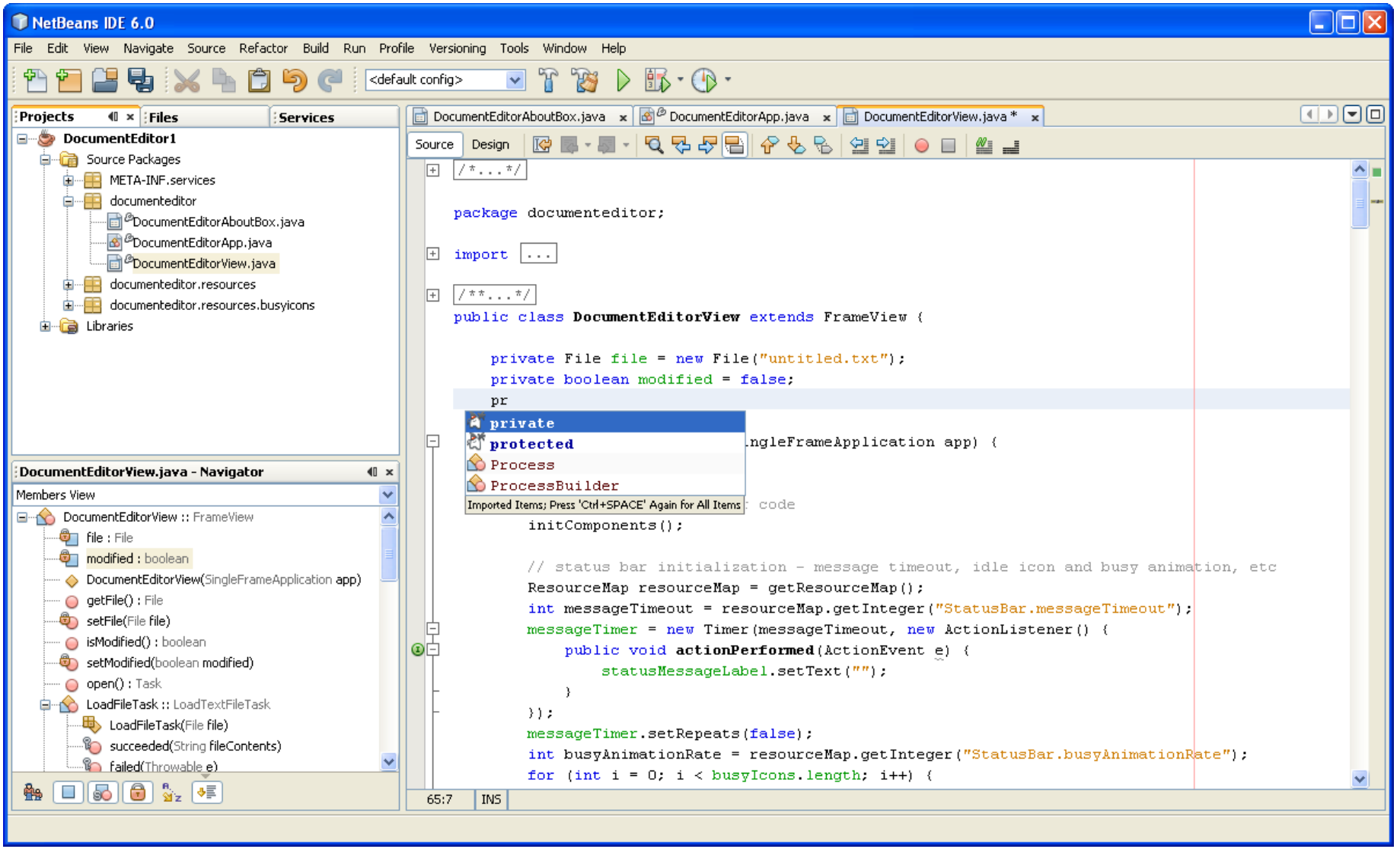
Ferramentas

- *Java Development Kit*!
- Eclipse (IDE gratuita): http://www.eclipse.org/downloads/.



Ferramentas

- Netbeans (IDE gratuita): http://www.netbeans.org

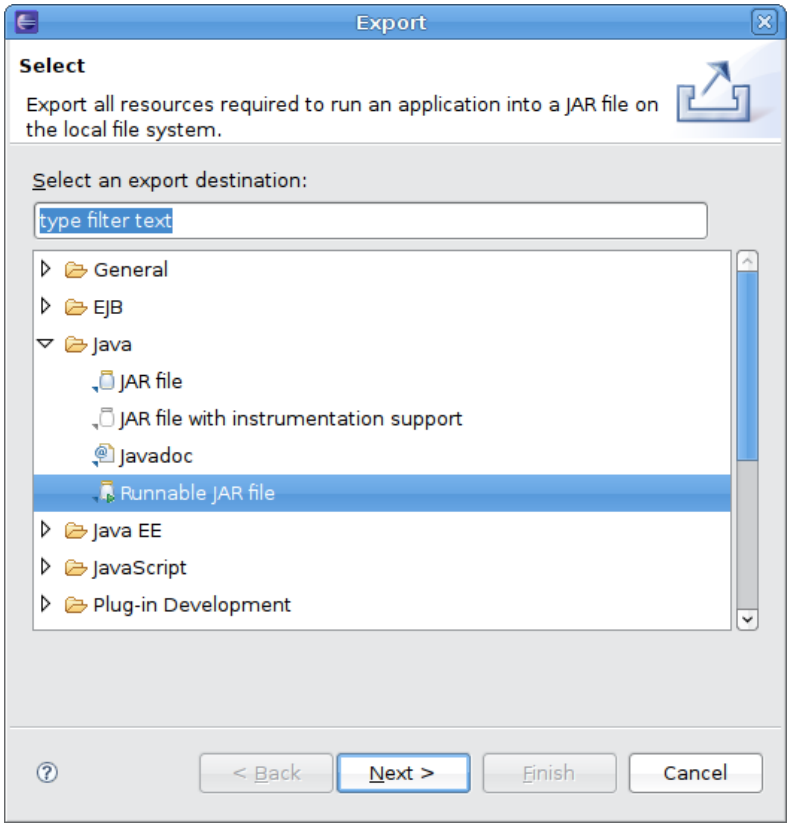


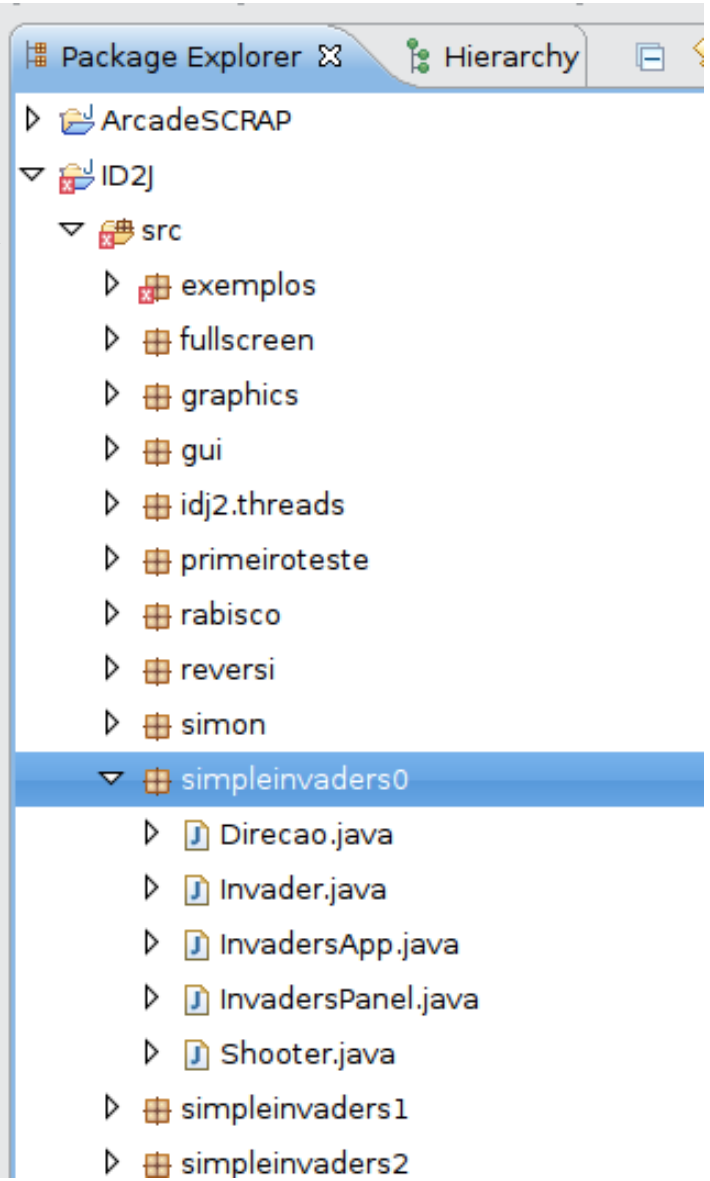
Empacotando

- Classes e arquivos necessários podem ser empacotados em um arquivo .jar, que pode ser executado por uma máquina virtual.
- Arquivos .jar são como arquivos .zip, com informações adicionais.

Empacotando (simples, Eclipse)

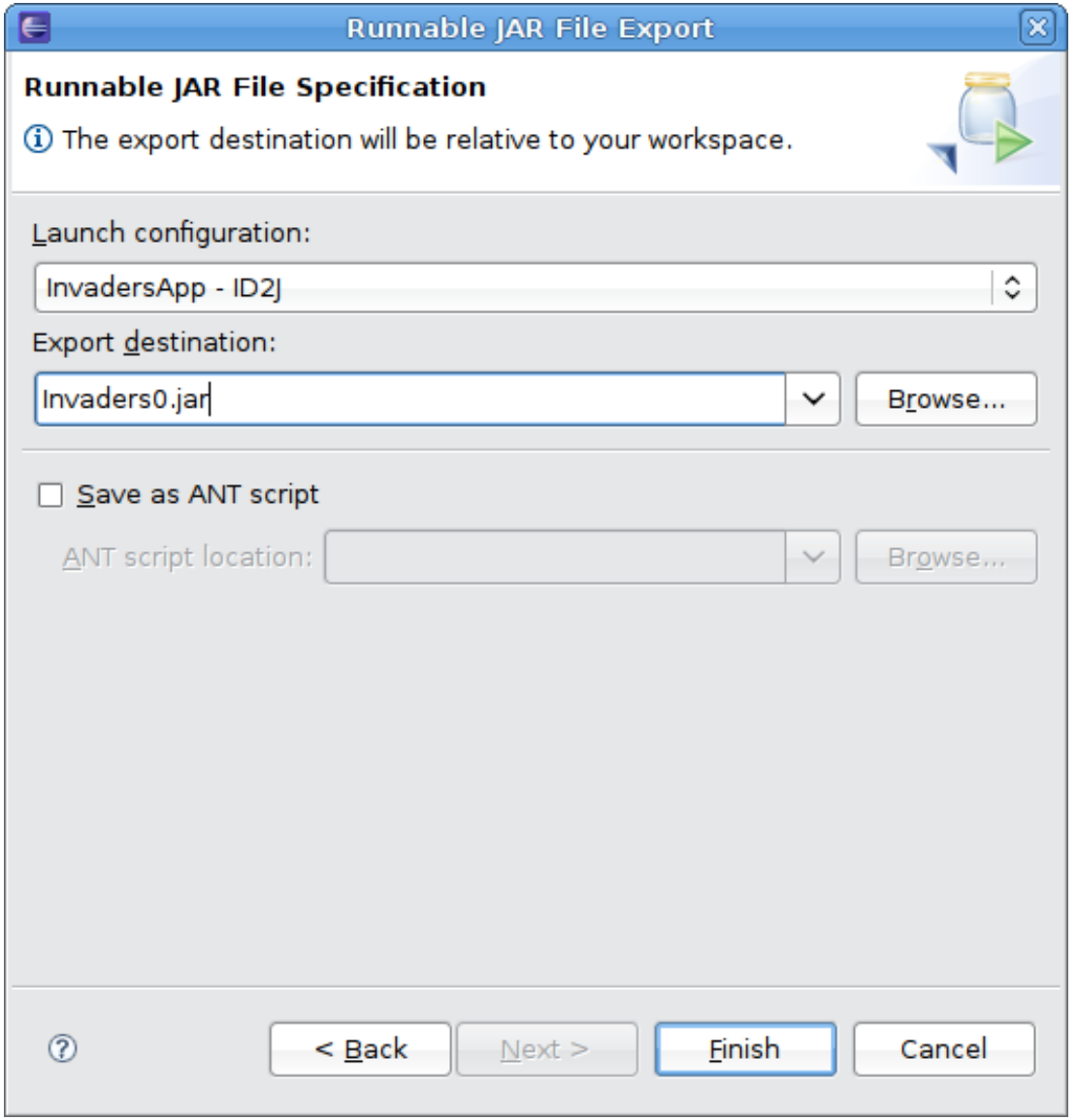
- Modo simples: Eclipse.
  1. Selecione o pacote no *Package Explorer*.
  2. Selecione *Runnable Jar File*.





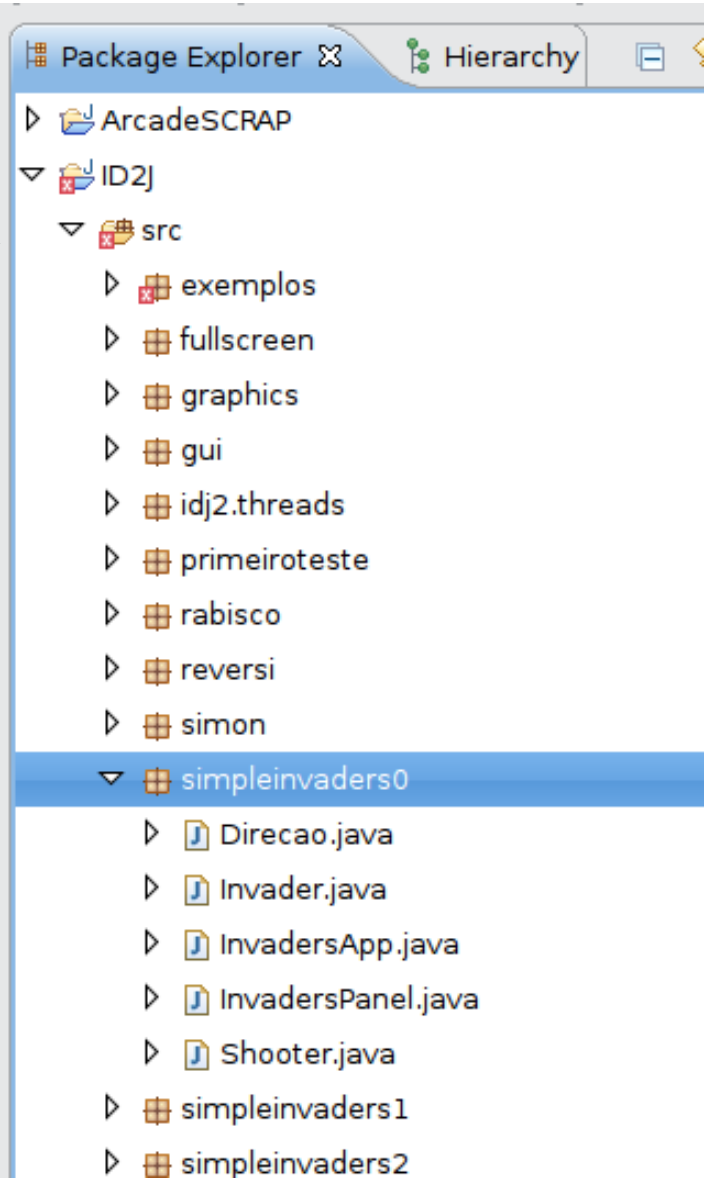
Empacotando (simples, Eclipse)

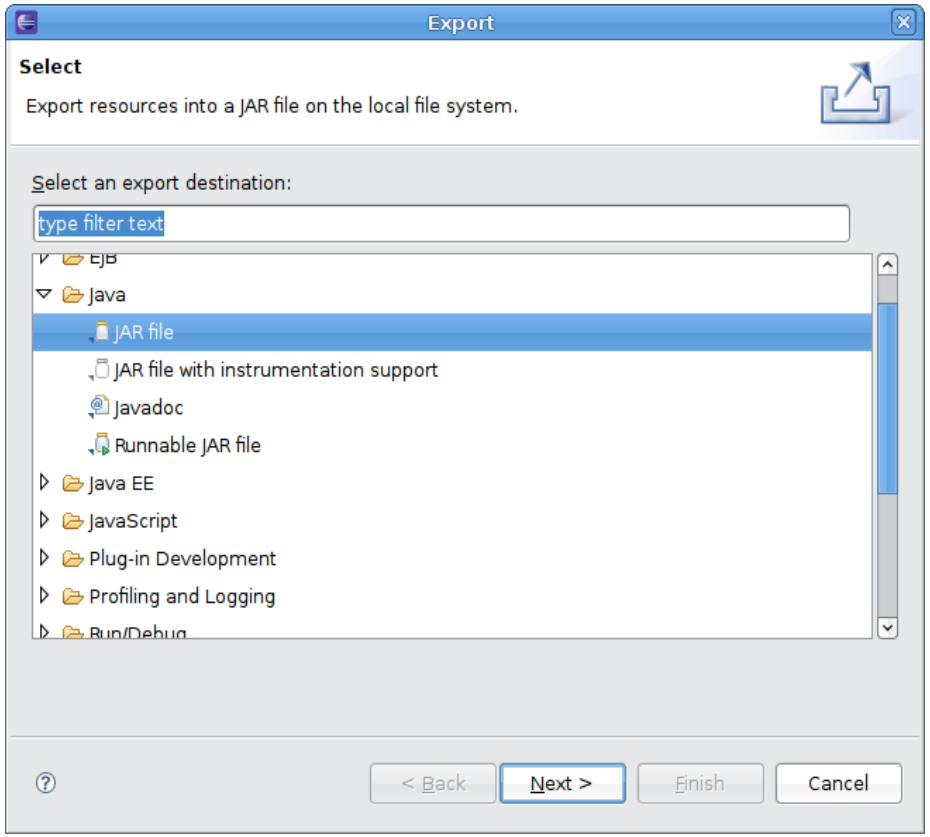
- Modo simples: Eclipse.
  3. Selecione um *Launch Configuration* e um nome para o .jar.
  4. Clique em *Finish*.
- O arquivo .jar conterá **todos** os arquivos do projeto, mas executará somente a classe do *Launch Configuration*.



Empacotando (manual, Eclipse)

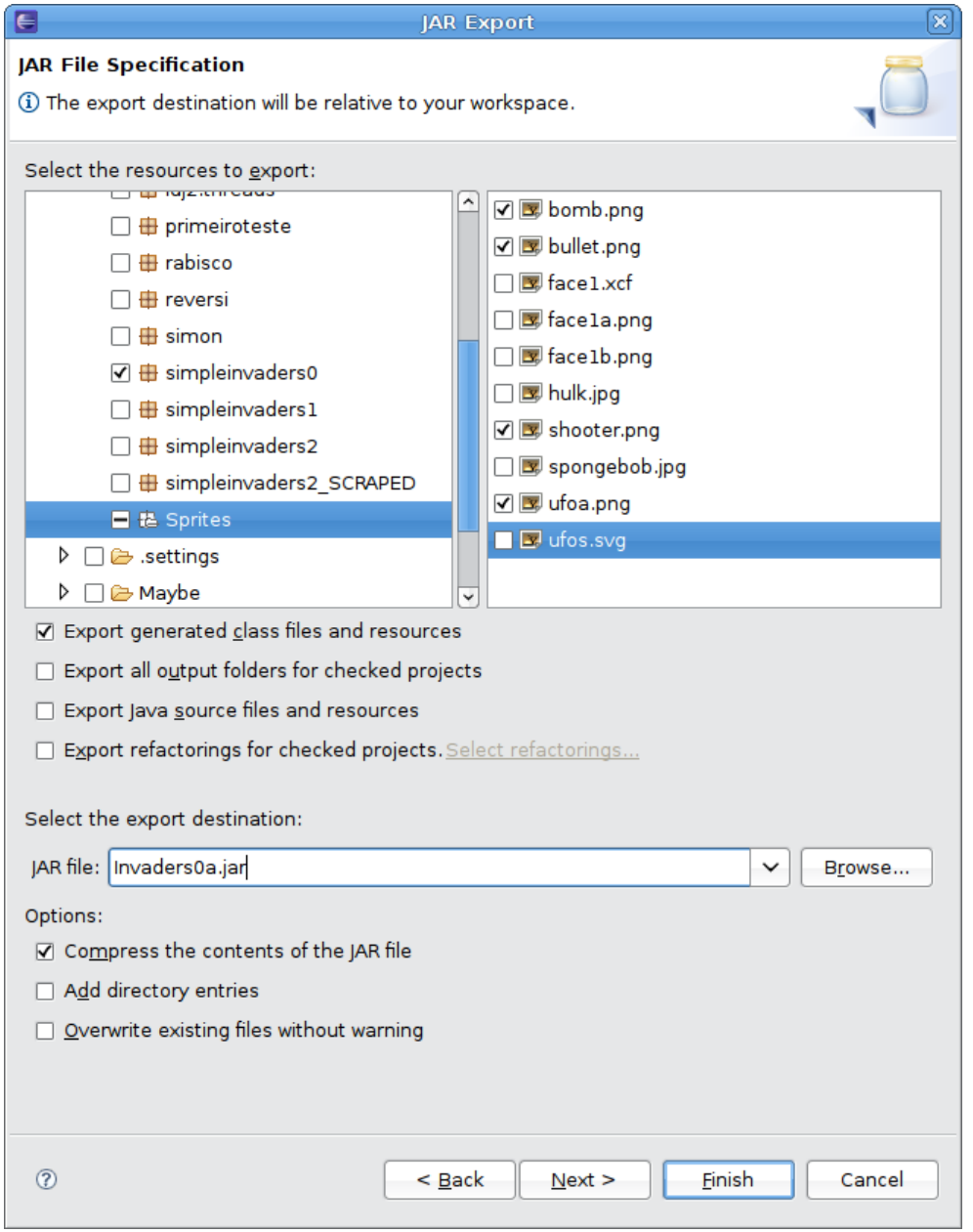
- Modo pouco menos simples: Eclipse.
  1. Selecione o pacote no *Package Explorer*.
  2. Selecione *Jar File*.





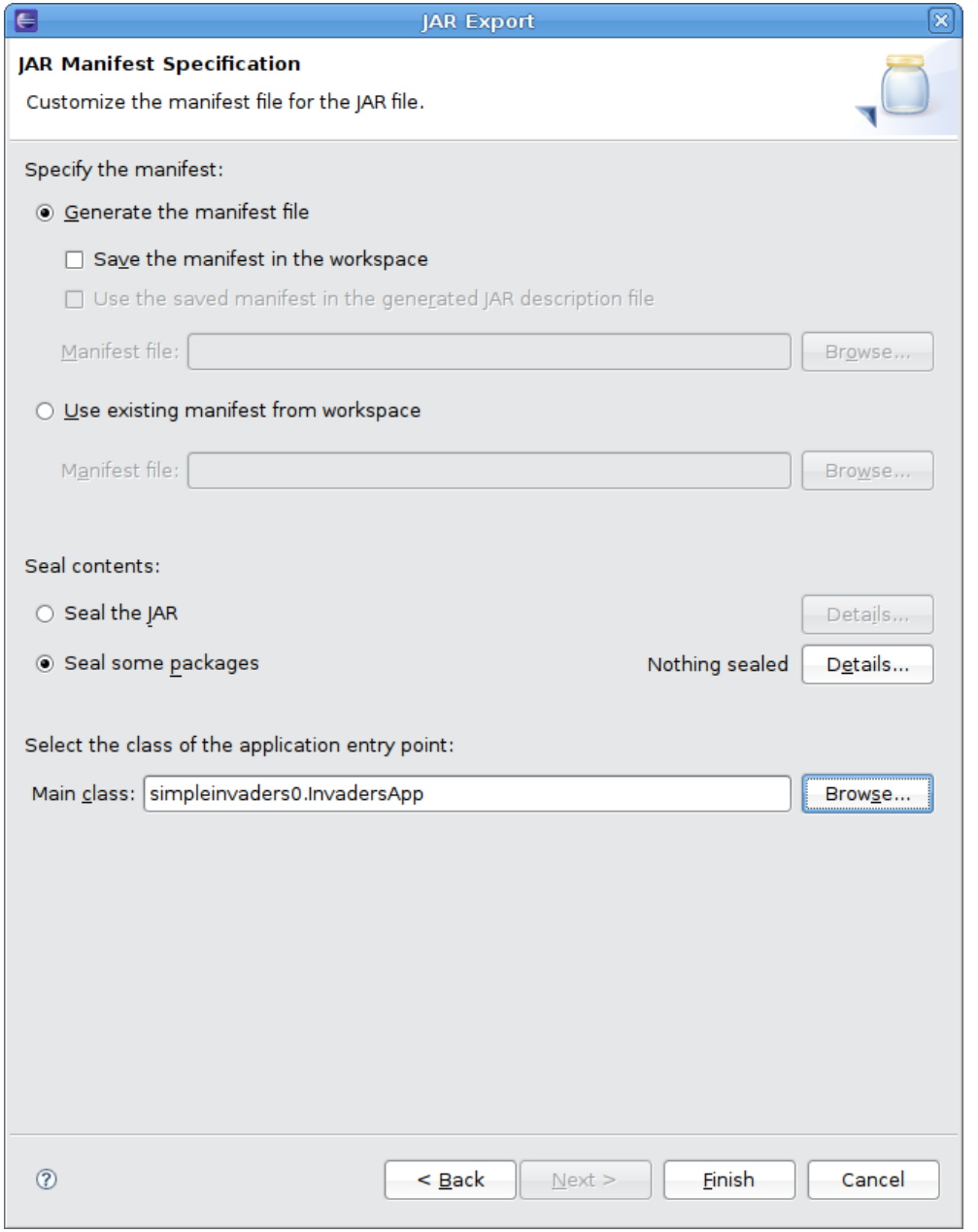
Empacotando (simples, Eclipse)

- Modo pouco menos simples: Eclipse.
  3. Selecione **todos** os arquivos relevantes (inclusive imagens e sons) e um nome para o .jar e clique em *Next*.
  4. No próximo diálogo clique também em *Next*.



Empacotando (simples, Eclipse)

- Modo pouco menos simples: Eclipse.
  5. Indique a classe que contém o método main e clique em *Finish*.



Empacotando

- Modo simples: 104198 bytes
- Modo pouco menos simples: 9927 bytes

- Empacotamento para *applets* é semelhante, mas modo simples no Eclipse não pode ser usado.

Omissões

- Modo *Full Screen*: ainda existem problemas de compatibilidade entre sistemas operacionais.
- Áudio.
- Performance de renderização.
  - Renderização ativa.
- *Deployment.*
- Inteligência artificial.

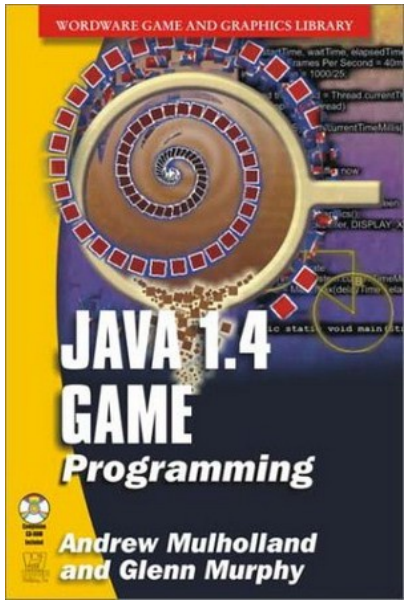
Full-Screen Exclusive Mode em Linux

- Série de pesadelos envolvendo Ubuntu, Fedora, Xinerama, FSEM.
- Varia de versão para versão!
- Solução 1: *Faux* FSEM?
- Solução 2 **(arriscada!):**

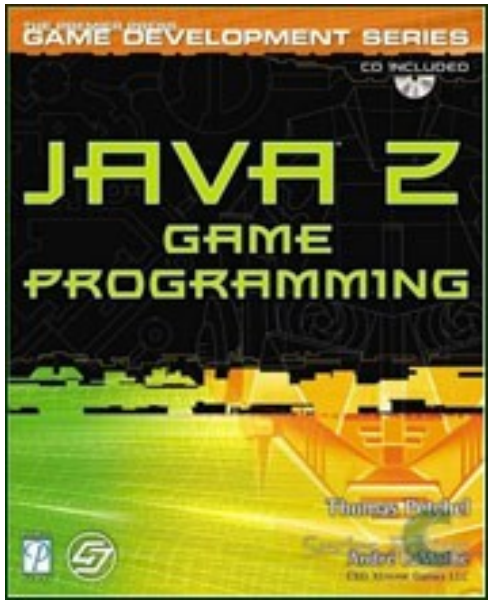
```
sed -i s/XINERAMA/ZINERAMA/g
/usr/bin/X11/Xorg
```

5 Referências

Referências

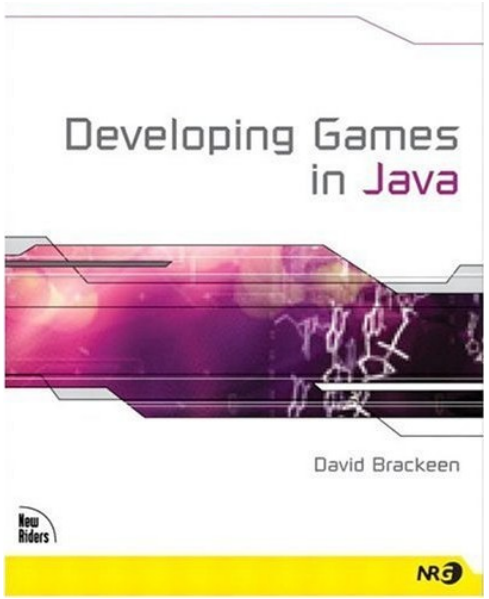


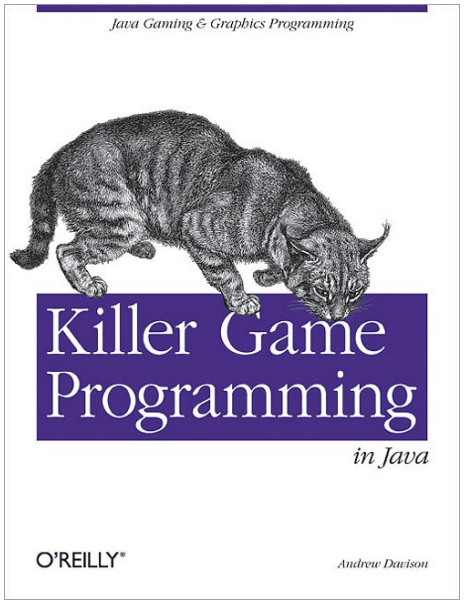
- *Java 1.4 Game Programming*; Andrew Mulholland e Glenn Murphy; Wordware, 2003, 647pp.
  - Introdução à Java, OO, I/O, threads, controles, animação, JDBC, MySQL, rede, etc.



- *Java 2 Game Programming*; Thomas Petchel; Premier Press, 2001, 784pp.
  - Introdução a Java, OO, AWT, Java2D, animações, controles, gerenciamento de cenas.
  - Exemplo: *Nodez!* (puzzle).

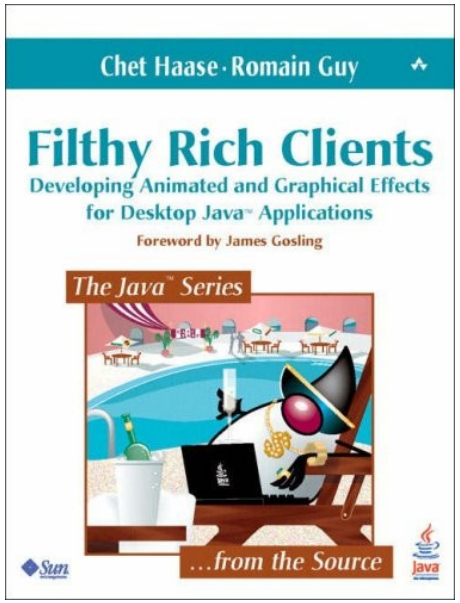
Referências



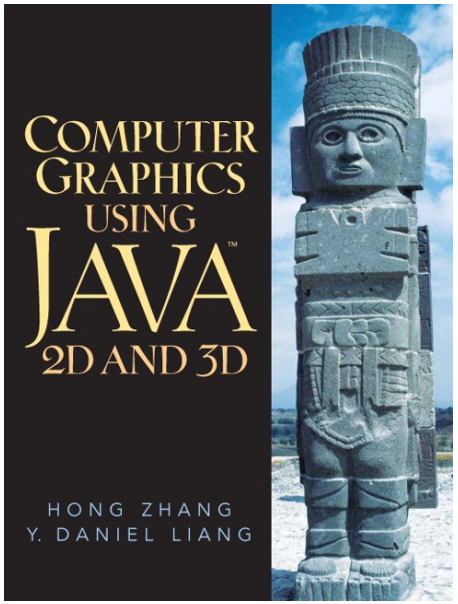


- *Developing Games in Java*; David Brackeen; New Riders, 2004, 996pp.
  - *Threads*, 2D e animação, interatividade, áudio, 3D, IA, otimização.
  - Exemplos: *scroller*.
- *Killer Game Programming in Java*; Andrew Davison; O'Reilly, 2005, 970pp.
  - Animação, imagens, sons, sprites, Java3D, sprites 3D, sistemas de partículas, etc.
  - Exemplos: worms, side-scroller, isométrico, labirinto 3D, FPS.
  - fivedots.coe.psu.ac.th/~ad/jg/

Referências

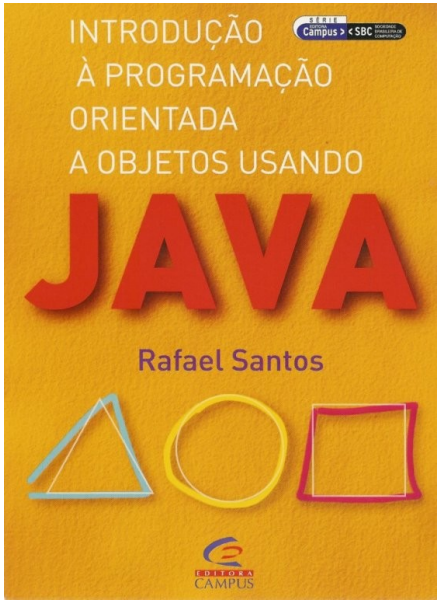


- *Filthy Rich Clients*; Chet Haase, Romain Guy; Prentice-Hall/Sun, 2007, 608pp.
  - Interfaces ricas em Swing, *threads*, animação de interfaces, temporização, processamento de imagens (para tela).



- *Computer Graphics Using Java 2D and 3D*; Hong Zhang, Y. Daniel Liang; Prentice-Hall, 2007, 632pp.
  - Gráficos, Java2D, renderização, Java3D.

Referências



- *Introdução à Programação Orientada a Objetos Usando Java*; Rafael Santos; Campus/SBC, 2003, 352pp.
  - Conceitos básicos: programação, orientação a objetos, modelagem, lógica.

- *Java: Como Programar*; H. M. Deitel, P. J. Deitel; Prentice-Hall, 2005, 1152pp.
  - Muita informação sobre Java e sobre APIs principais.