



# *Applets*

- Programas que podem ser executados por programas de navegação na Web ( “*browsers*” )
- possivelmente trazidos de computador remoto
- Iniciação por navegador a partir de uma página da Web
- em vez de pelo sistema operacional

# *E/S Gráfica*

- E/S em applets feita por (chamados) “componentes de interface gráfica”
- disponíveis na biblioteca *Swing*
- importação: `import javax.swing.*;`



# Páginas da Internet

- Descritas usando linguagem de descrição de hipertextos
- por meio de “marcas”.
- Chamadas de “*linguagens de marcação*”
- Linguagem *HTML* (*HyperText Markup Language*) é a linguagem mais usada para descrição de páginas na Web



# Linguagens de Marcação

- Contêm (além do texto propriamente dito) *marcas* que determinam estrutura e outras características que definem o aspecto do documento
- *Páginas dinâmicas* incluem também recursos para interação com usuários

# HTML

- Marca HTML começa com caractere <
- Segue nome da marca, e possivelmente parâmetros
- e finalmente o caractere >
- Texto seguinte modificado pela marca
- sendo fim do efeito especificado por marca com mesmo nome, só que precedido por /
- Ex: efeito da marca <strong> indica que texto seguinte deve ser escrito em negrito; terminado por </strong>

# Exemplo de página HTML

```
<html>
  <head> <title>MiniCalc</title> </head>
  <body> <h1>MiniCalc</h1> <hr>
    <applet code="MiniCalc.class" width=300 height=100>
      <param name=ops value="mult exp fat">
    </applet> <hr>
    <a href="http://www.dcc.ufmg.br/~camarao/ipcj/java/MiniCalc.java">
      Programa fonte</a>
  </body>
</html>
```

# Marcas HTML

- Conteúdo de página entre `<html>` e `</html>`
- Cabeçalho (opcional) entre `<head>` e `</head>`
- Cabeçalho pode especificar título da página entre marcas `<title>` e `</title>`
- Corpo da página especificado entre `<body>` e `</body>`
- Marca `applet` provova iniciação de *applet*, quando a página é mostrada por *navegador*

# Marca applet

```
<applet code="MiniCalc.class" width=300 height=100>  
...  
</applet>
```

- Parâmetro `code` da marca `applet` especifica arquivo contendo *bytecodes* a serem interpretados
- Definidos também largura e altura da janela a ser usada para interface com programa (em “*pixels*”)



# Interface com *applet*

```
<applet code="MiniCalc.class" width=320 height=120>  
  <param name=ops value="mult exp fat"> </applet>
```

- Parâmetro code deve ser sempre especificado
- Argumentos para *applet* passados por meio da marca param, que tem “atributos” name e value
- Valores passados de página HTML para *applet* como cadeias de caracteres
- Valor passado por ops é "mult exp fat"

# Usando argumentos em *applets*

- Método *getParameter* da classe *JApplet* obtém argumento passado por meio da marca `param`
- Chamada a *getParameter* deve especificar como argumento nome do parâmetro — no exemplo, "ops"
- Valor do tipo *String* retornado — no exemplo, "mult exp fat"

# *JApplet*

- Programa iniciado a partir de página da Internet deve conter subclasse da classe *JApplet* — da biblioteca *Swing*
- ou *Applet* — da biblioteca *AWT*
- Deve redefinir um ou mais dos métodos *init*, *start*, *stop* e *destroy*
- Definidos na classe *JApplet* e *Applet*

# Iniciação de *applets*

`public void init()` : primeiro método a ser chamado, quando *applet* é iniciado. Chamado uma única vez.

Usado para atribuição de valores iniciais a variáveis do programa e para criação de objetos.

`public void start()` : Chamado após *init* e toda vez que página é “*visitada*”.

# Terminação de *applets*

`public void stop()` : Chamado sempre que página deixa de ser mostrada

Ex: navegador mostra outra página

`public void destroy()` : Chamado, após o método *stop*, quando o programa termina sua execução



# *Applets* e segurança

- Código de *applet* (*bytecodes*) carregado a partir de página da Web, e interpretado localmente
- Sujeito a restrições por questão de segurança



# *Applets* e segurança

- *Applet* em geral obtém informações fornecidas diretamente pelo usuário
- e produz dados, que podem ser armazenados apenas por solicitação explícita do usuário
- não pode alterar sistema no qual é executado, nem obter dados de forma não autorizada pelo usuário
- Em geral, *applets* não podem criar nem ler arquivos, nem executar outros programas.



# *AWT e Swing*

- *AWT* (*Abstract Window Toolkit*) compõe núcleo da *JFC* (*Java Foundation Classes*),
- da qual *Swing* também faz parte
- *Swing* definida com base em *AWT*
- e usa classes de *AWT* para gerenciamento da disposição de componentes em janelas (*LayoutManagers*) e interfaces com métodos para tratamento de eventos



# *AWT e Swing*

- Em geral, para cada componente *AWT* existe um componente *Swing* análogo
- precedido do caractere “*J*”
- Ex: *Applet*   *JApplet*  
          *Frame*    *JFrame*
- Existem, no entanto, muitos componentes *Swing* para os quais não existe componente *AWT* análogo.



# Objetivos do Projeto *AWT*

- Objetivos até certo ponto conflitantes:
  - ★ permitir que interfaces gráficas fossem criadas da mesma maneira, independentemente de ambiente ou sistema (Unix, Windows, Mac etc.)
  - ★ permitir que essas interfaces tivessem a mesma aparência das interfaces nativas desses ambientes

# Motivação do Projeto *AWT*

- Fazer com que programador não precisasse se preocupar com detalhes específicos de cada ambiente na implementação de interface gráfica
- Usado então esquema de *pares*: para cada componente *AWT* existe um par equivalente, implementado especificamente para cada plataforma.
- Problema: funcionalidade de componentes limitada à encontrada em *todos* os ambientes nos quais é usada.

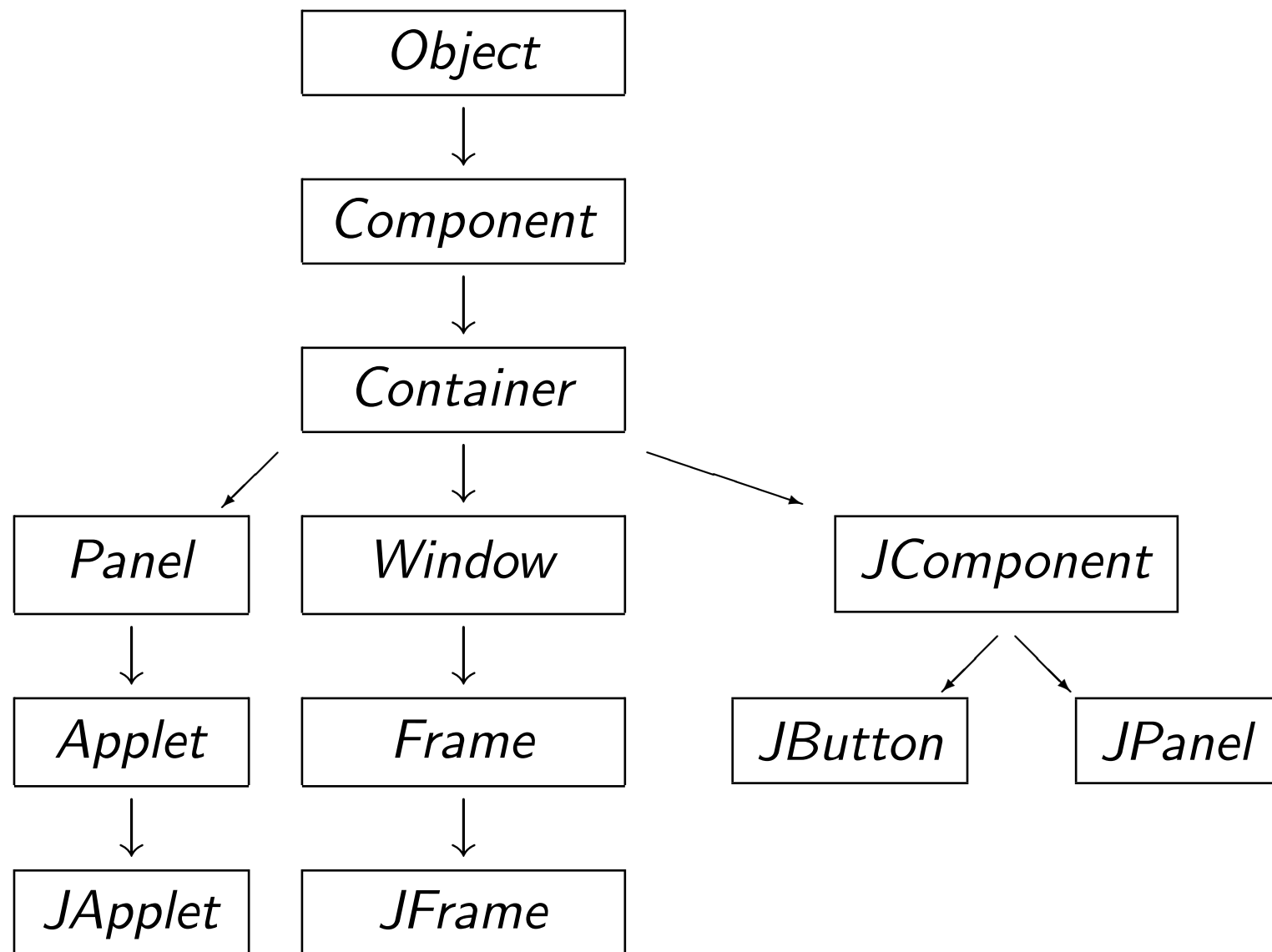


# Projeto *Swing*

- Escrita em Java, não dependendo assim de “código nativo” em dado ambiente, com dada funcionalidade
  - ★ Permite assim definição de mais componentes, com características e funcionalidades variadas
- Depende menos do sistema de interface de usuário nativo, e portanto está menos sujeita a erros na plataforma nativa
- Torna mais fácil usar interface de usuário com mesma aparência em plataformas diferentes, e permite mudança na aparência, para um mesmo código, devido ao uso da arquitetura *MVC*

# Arquitetura *MVC*

- Decomposição entre
  - ★ modelo: criação e manipulação de componentes
  - ★ visão: aspecto visual dos componentes
  - ★ controle: definição e tratamento de eventos associados aos componentes
- Arquitetura desenvolvida no final da década de 70
- Divulgada a partir da criação do ambiente de desenvolvimento da linguagem *Smalltalk*



# Eventos em Componentes

Método de delegação de tratamento de eventos:

1. Implementar método definido em interface, como:  
*ActionListener*, *WindowListener*, *MouseListener* etc.
2. Usar *addActionListener* para indicar objeto responsável por tratar eventos em componente:

```
cp.addActionListener(c);
```

# Implementando Interface *ActionListener*

Necessário implementar *actionPerformed*:

```
public class C implements ActionListener
{ ...
    public void actionPerformed (ActionEvent e)
    { /* código para tratamento do evento */
        ...e.getSource(); ... }
}
```







# *actionPerformed*

1. *actionPerformed* tem parâmetro do tipo *ActionEvent*, que descreve evento ocorrido.
2. Implementação envolve, em geral, obter o objeto que representa o componente no qual o evento ocorreu, usando *getSource*

# Disposição dos componentes

1. Disposição de componentes definida por meio do método *setLayout*, definido na classe *Container*
2. Bibliotecas *AWT* e *Swing* definem classes para suporte a diferentes modos de dispor componentes em painel


*FlowLayout*      *GridLayout*  
*BorderLayout*   *GridBagLayout*



# *FlowLayout*

1. Disposição seqüencial: esquerda para direita e de cima para baixo
2. Componente colocado na linha seguinte quando não cabe mais em uma linha
3. Construtor permite definir
  - número de espaços entre componente e o seguinte, na mesma linha, e número de espaços entre linhas adjacentes
  - alinhamento de componentes em cada linha (à esquerda, ao centro ou à direita)
  - se não especificado, alinhamento é centralizado (*CENTER*), e espaço entre componentes igual a 5 pontos

```
{ String texto = " Proemiatur apte, narrat aperte,"  
  "pugnat acriter, colligit fortifier, ornat excelse." +  
  "Postremo docet, delectat, afficit.";   
  getContentPane().setLayout(new FlowLayout(FlowLayout.RIGHT,10,5));  
  StringTokenizer tokens = new StringTokenizer(texto);  
  while (tokens.hasMoreTokens())  
    getContentPane().add(new  JButton(tokens.nextToken()));  
  setSize(400,200); } }
```



# *GridLayout*

1. Disposição em forma de tabela
2. Construtor define número de linhas e colunas

```
import java.awt.*; import java.applet.*;
import java.util.*;
public class ExemploGridLayout extends JApplet
{ public void init()
  { String texto = "Proemiatur apte, narrat aperte,"
    " pugnat acriter, colligit fortifier, ornat excelse,"
    " Postremo docet, delectat, afficit.";
  getContentPane().setLayout(new GridLayout(7,2));
  StringTokenizer tokens = new StringTokenizer(texto);
  while (tokens.hasMoreTokens())
    getContentPane().add(new JButton(tokens.nextToken()));
  setSize(400,200); } }
```





# *BorderLayout*

1. *BorderLayout* permite dispor componentes nas posições centro, norte, sul, leste e oeste de um painel
2. Classe *BorderLayout* implementa interface *LayoutManager2*, subtipo de *LayoutManager*







```
import java.awt.*; import javax.swing.*;
class ExemploBorderLayout extends JFrame
{ ExemploBorderLayout()
  { super("Exemplo de BorderLayout"); setSize(300,200);
    Container c = getContentPane();
    c.setLayout(new BorderLayout());
    c.add(new JButton("centro"), BorderLayout.CENTER);
    c.add(new JButton("norte"),      BorderLayout.NORTH);
    c.add(new JButton("sul"),        BorderLayout.SOUTH);
    c.add(new JButton("leste"),      BorderLayout.EAST);
    c.add(new JButton("oeste"),      BorderLayout.WEST);
    setVisible(true); }
  public static void main(String[] a) {new ExemploBorderLayout();}}
```



# *GridBagLayout*

1. *LayoutManager* mais poderoso e complexo
2. baseado em tabela (*grid*), mas
3. componentes podem ocupar várias linhas e colunas





# Usando *GridBagLayout*: Primeiro Passo

1. em um papel, divida o painel em linhas e colunas, contendo os componentes da interface gráfica
2. Componentes inseridos de acordo com número das linhas e colunas (números começam de zero)
3. Use *GridBagConstraints* para especificar posição



## *GridBagConstraints*

1. *gridx* / *gridy*: coluna / linha (nº) do canto superior esquerdo do componente
2. *gridwidth* / *gridheight* : nº de colunas / linhas ocupadas
3. *weightx* / *weighty*: espaço horizontal / vertical ocupado ao mudar de tamanho (em relação a outros componentes na mesma linha / coluna)



## *GridBagConstraints:fill*

Variável *fill* de objeto da classe *GridBagConstraints* controla mudança de tamanho:

1. *NONE*: não muda
2. *VERTICAL*: muda verticalmente
3. *HORIZONTAL*: muda horizontalmente
4. *BOTH*: muda em ambas as direções



## *GridBagConstraints.anchor*

Variável *anchor* de objeto da classe *GridBagConstraints* controla posição quando componente não ocupa toda área de linhas/colunas:

1. *NORTH*, *NORTHEAST*, *NORTHWEST*, *CENTER* (default), *EAST*, *SOUTHEAST*, *SOUTHWEST*.