

# **Programação Básica em Java**

**Apostila de**

**Patrícia Augustin Jaques**

# Programação Básica em Java

Autoria: **Patrícia Augustin Jaques** - *pjaques@unisinos.br*

Última atualização: Outubro de 2007.

Nenhuma parte desta apostila pode ser utilizada ou reproduzida, em qualquer meio ou forma, seja mecânico ou eletrônico, fotocópia, gravação, ou outros, sem autorização, prévia, expressa e específica do Autor. Essa apostila está protegida pela licença Creative Commons. Você pode usá-la para seus estudos, mas não pode usá-la para fins comerciais ou modificá-la. Bons estudos em Java!

Programação Básica em Java by [Patrícia Augustin Jaques](#) is licensed under a [Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License](#).

Meus agradecimentos a aluna Natali Silverio que me permitiu conhecer a licença Creative Commons.

Um especial agradecimento ao aluno Leandro Medeiros que ajudou na revisão do texto.

## Sumário

<b>1 Introdução à Linguagem Java</b>	<b>7</b>
1.1 Características da Linguagem Java	7
1.1.1 Parecida com C, C++:	7
1.1.2 Orientada a Objetos:	7
1.1.3 Compilada e Interpretada:	7
1.1.4 Segura:	8
1.1.5 Suporta concorrência:	8
1.2 O Ambiente de Programação Java	9
1.2.1 Instalando e configurando o ambiente de programação Java	9
1.2.2 Onde encontro o ambiente de programação Java?	9
1.2.3 Versão Java 1.1.x	9
1.2.4 Versão Java 1.2.x	9
1.2.5 Outras opções	10
1.2.6 Que ferramenta uso para criar os meus programas?	10
1.3 O Primeiro Programa Java	10
1.3.1 Passo a Passo	10
1.3.2 Que nome dar ao arquivo?	11
1.3.3 Compilando o código	11
1.3.4 Executando o programa HelloWorld	11
1.4 A Máquina Virtual Java	12
1.4.1 O Ciclo de Vida de uma Máquina Virtual Java	12
1.5 Exercícios	13
<b>2 Estruturas Fundamentais de Programação em Java</b>	<b>14</b>
2.1 Espaço em Branco	14
2.2 Comentários em Java	14
2.3 Palavras-chave reservadas	14
2.4 Identificadores	15
2.5 Convenções para nomes de identificadores	15
2.5.1 Variáveis e métodos	15
2.5.2 Constantes	15
2.5.3 Classes	15
2.6 Declaração Import	15
2.7 Tipos Primitivos de Dados (Tipos Nativos de Dados)	15
2.8 Conversões entre Tipos Numéricos	16
2.9 Declaração e Inicialização de Valores	17
2.10 Constantes	17
2.11 Operadores	17
2.11.1 Operadores Aritméticos	18
2.11.2 Operadores Relacionais	18
2.11.3 Operadores Lógicos	18
2.11.4 Operadores Bit-a-Bit	19
2.11.5 Operador if-then-else ternário	20
2.11.6 Precedência de Operadores	21
2.12 Controle de Fluxo	21
2.12.1 if-else	21
2.12.2 switch	22
2.12.3 while	23
2.12.4 do-while	23
2.12.5 for	24
2.12.6 continue	24
2.12.7 break	25
2.13 Exercícios	26

<b>3 Fundamentos da Orientação a Objetos .....</b>	<b>27</b>
3.1 Classes e Objetos.....	27
3.1.1 Variáveis.....	28
3.1.2 Métodos.....	28
3.1.3 Métodos Construtores.....	29
3.1.4 Finalização de um Objeto.....	30
3.1.5 Instanciando uma classe.....	30
3.1.6 Operador this .....	30
3.1.7 Acessando métodos e variáveis de um objeto.....	30
3.2 Sobrecarga de Métodos (Overloading) .....	31
3.3 Passagem de Parâmetros em Java.....	31
3.4 Visibilidade.....	32
3.5 Herança .....	33
3.5.1 super.....	34
3.6 Sobreposição (Overriding) .....	34
3.6.1 Dinamic Binding.....	35
3.7 O Modificador "final" .....	36
3.8 Classes Abstratas.....	36
3.9 Interfaces .....	38
3.9.1 Utilizando as Interfaces.....	39
3.10 Inner Classes.....	39
3.11 Classes Anônimas (Anonymous Class) .....	40
3.12 Garbage Collection .....	41
3.13 Exercícios.....	41
<b>4 Strings .....</b>	<b>44</b>
4.1 Criando uma string .....	44
4.1.1 Criando uma string vazia:.....	44
4.1.2 Criando uma string a partir de um array de caracteres:.....	44
4.1.3 Criando uma String a partir de um array de bytes:.....	44
4.2 Substituição de Caracteres.....	45
4.3 Concatenação de Strings .....	45
4.4 Substring .....	45
4.5 Alterando os caracteres de uma String para maiúsculo .....	46
4.6 Alterando os caracteres de uma String para minúsculo .....	46
4.7 Retirando espaços em branco .....	46
4.8 Extração de Caractere .....	46
4.9 Comparação de Strings.....	46
4.10 Tamanho de uma string.....	47
4.11 Identificando a posição de caracteres ou substrings em uma String .....	47
4.12 Ordem .....	47
4.13 Conversões de tipos primitivos de dados para Strings.....	48
4.14 Conversão de String para tipos primitivos:.....	48
4.15 As classes Wrappers.....	49
4.16 StringTokenizer.....	49
4.17 Exercícios.....	49
<b>5 Arrays .....</b>	<b>51</b>
5.1 Criando um array.....	51
5.2 Inicializando arrays .....	51
5.2.1 Inicialização Default.....	52
5.3 Acessando um elemento de uma Array .....	52
5.4 Obtendo tamanho de um array .....	53

5.5 Copiando o conteúdo de um array para outro array .....	53
5.6 Arrays Multidimensionais .....	53
5.6.1 Criando arrays multidimensionais.....	53
5.6.2 Inicializando um array multidimensional.....	53
5.7 Array de Objetos.....	54
5.8 Exercícios.....	54
<b>6 Vector .....</b>	<b>55</b>
6.1 Criando um Vector .....	55
6.2 Inserindo objetos em um Vector .....	55
6.2.1 Inserindo um elemento em uma determinada posição.....	55
6.2.2 Inserindo um elemento no final do Vector.....	55
6.3 Verifica a existência de um elemento no Vector .....	55
6.4 Retornando um elemento .....	56
6.5 Retornado a posição de um elemento .....	56
6.6 Substituindo um objeto em uma determinada posição.....	56
6.7 Removendo um elemento do Vector .....	56
6.8 Tamanho de um Vector .....	56
6.9 Convertendo um Vector para array .....	56
6.10 Exercícios .....	57
<b>7 Gerando documentação HTML do código com javadoc.....</b>	<b>58</b>
<b>8 Interfaces em Java – A biblioteca AWT.....</b>	<b>60</b>
8.1 Frame.....	60
8.2 Gerenciadores de Layout .....	61
8.2.1 FlowLayout.....	61
8.2.2 BorderLayout.....	62
8.2.3 GridLayout .....	63
8.2.4 CardLayout.....	63
8.2.5 GridBagLayout.....	64
8.2.6 Compreendendo os campos do GridBagConstraints.....	65
8.3 Dispondo Componentes sem Usar um Gerenciador de Layout .....	67
8.4 Panel .....	68
8.5 O Modelo de Delegação de Eventos.....	69
8.6 Adaptadores.....	70
8.7 Label .....	71
8.8 Definindo Fontes e Cores de Texto .....	71
8.9 Button .....	73
8.10 Checkbox .....	73
8.11 CheckboxGroup .....	74
8.12 Choice.....	74
8.13 Canvas.....	75
8.14 TextField .....	76
8.15 TextArea .....	77
8.16 List 78	
8.17 Dialog .....	79
8.18 FileDialog .....	80
8.19 ScrollPane .....	81
8.20 Menu .....	82
8.20.1 Adicionando Atalhos para os Menus.....	83
8.21 CheckBoxMenuItem .....	83

8.22 Imprimindo um Frame .....	84
8.23 Exercícios .....	85
<b>9 Applets .....</b>	<b>87</b>
9.1 Restrições de Segurança de um Applet .....	87
9.2 Hierarquia de Java .....	87
9.3 Carregando um Applet .....	88
9.4 O Ciclo de Vida de um Applet.....	88
9.5 O primeiro exemplo de um applet.....	89
9.6 Criando programas Java aplicativos e applets .....	90
9.7 Exibindo Som e Imagem em um Applet.....	91
<b>10 Interface Gráfica – Biblioteca Swing.....</b>	<b>93</b>
10.1 Determinando o Look-and-Feel de uma Interface Gráfica em Java.....	93
10.2 JFrame.....	94
10.2.1 Capturando os eventos de fechar janela no JFrame .....	96
10.3 Capturando eventos nos componentes Swing .....	96
10.4 Determinando o Layout Manager do Frame .....	96
10.5 JPanel .....	97
10.6 JButton .....	97
10.6.1 Um exemplo simples de botão:.....	97
10.6.2 Desenhando botões com Imagens: .....	98
10.7 JLabel .....	99
10.8 JTextField e JTextArea.....	99
10.9 JCheckBox.....	100
10.10 JRadioButton .....	102
10.11 JComboBox.....	104
10.12 JList.....	104
10.12.1 ListModel.....	106
10.12.2 ListCellRendering .....	109
10.13 Caixas de Diálogo (JDialog e JOptionPane) .....	110
10.13.1 JOptionPane.....	110
10.14 JDialog.....	111
10.15 JFileChooser .....	113
10.16 Menu .....	114
10.17 JApplet.....	117
10.17.1 Applet Context.....	119
<b>11 Endereços Úteis na Internet.....</b>	<b>122</b>
<b>12 Referências Bibliográficas .....</b>	<b>123</b>

# 1 Introdução à Linguagem Java

A linguagem Java surgiu em 1991 quando cientistas da Sun, liderados por Patrick Naughton e James Gosling, projetaram uma linguagem pequena para ser utilizada em eletrodomésticos em geral. O projeto se chamava "Green".

Como o pessoal da Sun vinha de uma geração UNIX, eles basearam a linguagem em C++ e ela foi chamada originalmente de "Oak". Oak é um tipo de árvore e a linguagem foi assim denominada porque havia uma árvore deste tipo, a qual Gosling gostava de apreciar, e que podia ser vista da janela de sua sala na Sun. Porém, posteriormente, eles verificaram que Oak era o nome de uma linguagem de programação existente e, por isso, trocaram o nome para Java.

O projeto Green passou os anos de 1993 e 1994 procurando uma empresa que quisesse comprar a sua tecnologia, mas não encontraram ninguém. Dizem que Patrick Naughton, uma das pessoas responsáveis pelo marketing, conseguiu acumular 300.000 milhas aéreas de bônus tentando vender a tecnologia Java.

Enquanto isso a World Wide Web crescia mais e mais. A chave para a Web é o browser que recebe a página hipertexto e o converte para a tela. Então, para mostrar o poder de Java, Patrick Naughton e Jonathan Payne construíram o browser HotJava. Este browser reconhecia applets e, por isso, tinha uma máquina virtual Java embutida em si. Em 1995, a Netscape decidiu tornar a próxima versão do seu browser (versão 2.0) apta a reconhecer applets Java. A partir de então, todas as versões posteriores do Netscape, inclusive do Internet Explorer, reconhecem Java.

## 1.1 Características da Linguagem Java

### 1.1.1 Parecida com C, C++:

A sintaxe da linguagem Java é muito semelhante a da linguagem C ou de C++. Logo, o programador que já conhece a linguagem C achará a transição para Java simples e fácil.

A sintaxe de Java fornece uma versão mais limpa do que C++. Em Java não existe aritmética de ponteiros, estruturas, uniões e etc.

### 1.1.2 Orientada a Objetos:

Java é uma linguagem voltada para a programação orientada a objetos e, por isso, todo o código está contido dentro de classes.

Java suporta herança simples, mas não herança múltipla. A ausência de herança múltipla pode ser compensada pelo uso de herança e interfaces, onde uma classe herda o comportamento de sua superclasse além de oferecer uma implementação para uma ou mais interfaces.

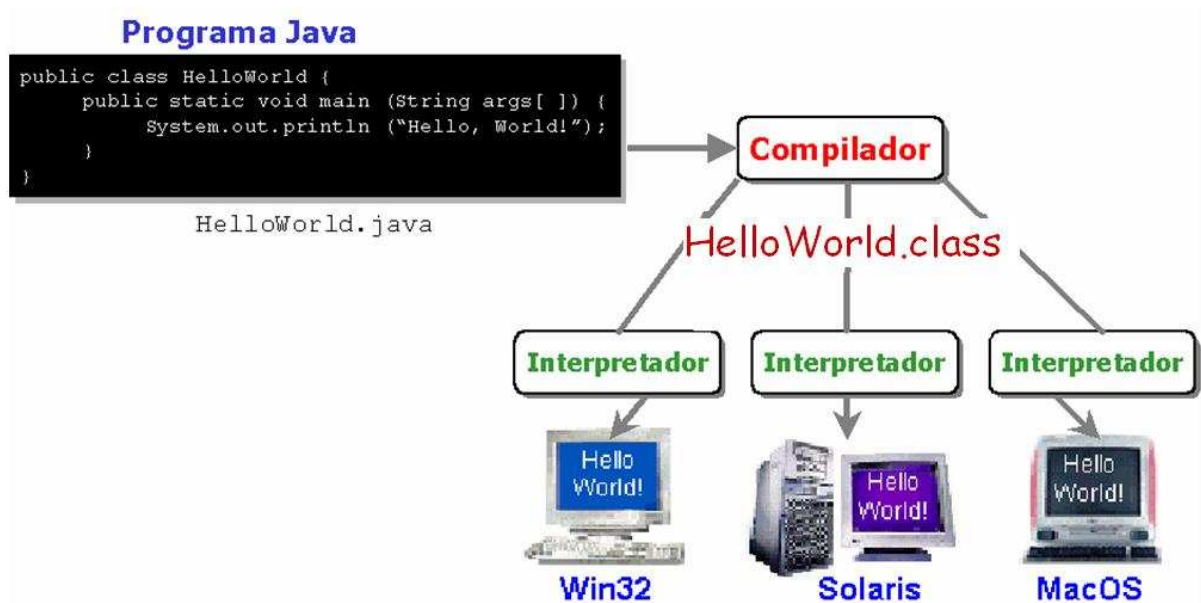
### 1.1.3 Compilada e Interpretada:

Todo programa Java é compilado e interpretado. Um programa em Java é compilado para um código composto por instruções chamadas de "bytecode". O "bytecode" é um código de uma máquina virtual, chamada Máquina Virtual Java (Java Virtual Machine - JVM), idealizada pelos criadores da linguagem. Os bytecodes são independentes de plataforma e são interpretados pela JVM para serem executados no computador.

Todo o interpretador Java ou browser que execute applets Java é uma implementação de uma Máquina Virtual Java. A JVM também pode ser implementada em hardware.

Além da JVM, a plataforma Java é composta também pela Java Application Programming Interface (Java API). A API Java é uma grande coleção de componentes de software disponibilizados que fornecem muitas capacidades interessantes e úteis, tais como, componentes de interface gráfica, conexão via sockets, etc. A API Java é agrupada em bibliotecas (*packages ou pacotes*) de componentes relacionados.

O código Java é compilado uma única vez, porém o programa é interpretado toda vez que for executado. Esse processo pode ser mais bem visualizado na figura a seguir.



#### 1.1.4 Segura:

A presença de coleta automática de lixo (Garbage Collection) evita erros comuns que os programadores cometem quando são obrigados a gerenciar diretamente a memória (C, C++, Pascal). A eliminação do uso de ponteiros, em favor do uso de vetores, objetos e outras estruturas substitutivas, traz benefícios em termos de segurança. O programador é proibido de obter acesso à memória que não pertence ao seu programa, além de não ter chances de cometer erros comuns tais como uso indevido de aritmética de ponteiros.

A presença de mecanismos de tratamento de exceções torna as aplicações mais robustas, não permitindo que elas aborrem, mesmo quando rodando sob condições anormais.

#### 1.1.5 Suporta concorrência:

Java suporta Multithreading que é a capacidade de um programa fazer mais de uma coisa ao mesmo tempo como, por exemplo, imprimir enquanto lê um fax. A linguagem permite a criação, de maneira fácil, de vários "threads" de execução, que eliminam o período de inatividade do processador executando concorrentemente ou tomando vantagem da execução paralela real em sistemas multiprocessadores.



## 1.2 O Ambiente de Programação Java

### 1.2.1 Instalando e configurando o ambiente de programação Java

O ambiente de programação Java disponibilizado pela Sun é composto basicamente pelas seguintes ferramentas:

Ferramenta	Descrição
java.exe	O interpretador Java que será usado para executar os aplicativos Java.
javac.exe	O compilador Java que transforma programas fonte escritos em Java (.java) em bytecodes (.class).
jdb.exe	Java Language Debugger – permite visualizar a execução passo-a-passo.
appletviewer.exe	Ferramenta que permite visualizar e testar os applets sem a necessidade de um browser.

Existem versões do ambiente Java para as plataformas mais conhecidas: Windows, Linux, UNIX e etc. Embora o código do Java seja portátil, a Máquina Virtual Java é dependente de plataforma. Por isso, tenha sempre o cuidado de copiar o ambiente de programação compatível com o Sistema Operacional que você está usando.

### 1.2.2 Onde encontro o ambiente de programação Java?

O ambiente de programação Java pode ser copiado da página da SUN <http://java.sun.com/products/>. Neste *site* você encontrará as últimas versões do ambiente de programação Java, bem como a documentação da biblioteca Java. A seguir, encontram-se instruções de como instalar este ambiente.

### 1.2.3 Versão Java 1.1.x

Até a versão 1.1.x do Java, o ambiente de programação se chamava **Java Development Kit (JDK)** e quando instalado ficava armazenado no diretório JDK1.1.x. Para essa versão era necessário configurar algumas variáveis de ambientes (**path** e **classpath**) para uso adequado. No Windows 95 e 98, essas variáveis deveriam ser alteradas no arquivo **autoexec.bat** que se encontra localizado no diretório raiz do seu HD.

As seguintes linhas de código deveriam ser adicionadas ao seu arquivo autoexec.bat para a versão 1.1.x do JDK:

```
PATH=%path%;jdk1.1.x\bin
set CLASSPATH=.;c:\jdk1.1.x\lib
```

Na primeira linha é descrito o diretório em que as ferramentas Java (java, javac, etc) podem ser encontradas. O %path% é usado para as configurações anteriores de PATH não serem perdidas. A variável CLASSPATH (segunda linha) contém o diretório em que estão instaladas as bibliotecas Java.



Sempre que declarar a variável CLASSPATH é importante colocar o ponto como um dos argumentos. Com isso você está definindo que o Java deve procurar por classes Java no diretório local.

### 1.2.4 Versão Java 1.2.x

Atualmente o ambiente de programação Java encontra-se na versão 1.4.2 (junho de 2004). A partir da versão 1.2.x este ambiente passou a se chamar **Java 2 Software Development Kit, Standard Edition (J2SE)**.

A partir desta versão é necessário apenas configurar a variável de ambiente PATH. A localização das bibliotecas Java é encontrada automaticamente pela Máquina Virtual Java através do diretório fornecido no PATH. A variável PATH é definida no arquivo autoexec.bat da seguinte maneira:

```
PATH=%PATH%;c:\jdk1.3\bin
```



Se for necessário utilizar uma outra biblioteca além das bibliotecas padrões fornecidas com o Java2 SDK (por exemplo, servlets), é necessário que você a defina no CLASSPATH.

## 1.2.5 Outras opções

A IBM disponibiliza gratuitamente na sua homepage <http://oss.software.ibm.com/developerworks/projects/jikes> o compilador Java IBM Jikes. Atualmente, o Jikes encontra-se na versão 1.21. Você vai observar que ele é bem mais rápido que o compilador javac da SUN. Vale a pena experimentar!

Para o Jikes é importante definir a variável de ambiente classpath no arquivo autoexec.bat, além de copiar o jikes para o diretório bin onde foi instalado o jdk:

```
PATH=%PATH%; c:\jdk1.3\bin
set CLASSPATH=.;c:\jdk1.3\jre\lib\rt.jar
```

## 1.2.6 Que ferramenta uso para criar os meus programas?

O código do programa Java que você criar pode ser digitado em qualquer editor de texto padrão como, por exemplo, o WordPad e o NotePad. Atualmente, existem alguns editores de texto específicos para criação de programas Java.

O Kawa, por exemplo, é um *shareware* e pode ser copiado do site <http://www.allaire.com/>.

Um outro bom editor que pode ser encontrado na versão freeware é o JCreator. Ele pode ser copiado do site <http://www.jcreator.com>.

## 1.3 O Primeiro Programa Java

Neste capítulo vamos escrever, compilar e executar o tradicional aplicativo "Hello World". O código deste programa será explicado posteriormente para que você possa começar a entender os fundamentos da programação Java.

```
1. /* Meu programa Java */
2. public class HelloWorld
3. {
4.     public static void main (String args[ ])
5.     {
6.         System.out.println ("Hello, World!");
7.     } // da rotina main
8. } // da class
```

### 1.3.1 Passo a Passo

**Linha 2:** public class HelloWorld

Esta linha utiliza a palavra reservada `class` para declarar que uma nova classe será definida aqui. `HelloWorld` é o nome usado para identificar a classe. Toda a definição da classe, inclusive todo o código e os dados, estará entre a chave de abertura “{” e a chave final “}” que se encontram nas linhas 5 e 8 deste exemplo.

**Linha 4:** `public static void main (String args[ ])`

A linha 3 contém a declaração do método `main`. O método `main` é simplesmente um ponto de partida para o interpretador Java. É por onde será iniciada a execução.

O método `main` deverá sempre ser declarado na forma acima.

**Linha 6:** `System.out.println (“Hello, World!”);`

Esta linha executa o método `println` do objeto `out`. Este objeto é uma instância da classe `OutputStream` e foi declarado como variável de classe (`static`) na classe `System`. Este método imprime na tela uma mensagem texto, no caso, “Hello, World!”.

Por causa do modelo de objeto, uma saída simples de console é complicada para entender. Por isso, até aprofundarmos o conteúdo suficientemente, pense apenas em `System.out.println` como um método para impressão na tela (saída do console).

### 1.3.2 Que nome dar ao arquivo?

Após digitar o código-fonte, você deve nomear o arquivo. Os arquivos que contém o código-fonte Java devem sempre ter a terminação “.java”. Geralmente, em Java, coloca-se uma classe dentro de cada arquivo. O arquivo conterá o mesmo nome da classe.

Cuidado, o compilador Java diferencia letras maiúsculas de minúsculas, por isso, preste atenção quando for nomear o arquivo.

Se você utilizar o modificador `public` para a classe, por exemplo, `public class HelloWorld`, o arquivo deve possuir o mesmo nome que a classe. Caso não utilize `public`, o arquivo pode ter outro nome.

### 1.3.3 Compilando o código

Para compilar o código acima você deve digitar:

```
C:\> javac HelloWorld.java
```

O compilador é chamado pelo comando `javac` seguido do nome do arquivo sempre com a terminação “.java”. Ao ser compilado, se o código não possuir nenhum erro, será gerado um arquivo chamado `HelloWorld.class` composto por `bytecodes`. Esse programa é independente de plataforma e, por isso, o seu programa `HelloWorld (.class)` pode ser executado em qualquer sistema operacional que possua a JVM instalada.

### 1.3.4 Executando o programa HelloWorld

Após a compilação do programa, em que foi gerado o arquivo `HelloWorld.class`, você pode executá-lo digitando o seguinte comando:

```
C:\>java HelloWorld
```

No código acima estamos chamando o interpretador Java (`java.exe`) para carregar a classe `HelloWorld` e executá-la. No comando de execução **não** se deve digitar a extensão do arquivo (`.class`). A seguinte saída será gerada para o programa acima:



> Hello, World!

## 1.4 A Máquina Virtual Java

Quando trabalhamos com Java, um dos termos que mais ouvimos é “Maquina Virtual Java” (MVJ) (ou *Java Virtual Machine*, em inglês). A Máquina Virtual Java é um computador abstrato definido por uma especificação. Para executar um programa Java, precisamos de uma implementação desta máquina virtual. Por exemplo, a Sun oferece uma implementação da MVJ que é o interpretador `java.exe` que vem juntamente com o ambiente de programação J2SDK.

Mas, quando ouvimos falar de Java Virtual Machine, na verdade, o termo pode estar se referindo a 3 coisas diferentes (Venners, 2000):

- a especificação abstrata da MVJ;
- uma implementação concreta desta máquina virtual. No caso, por exemplo, o programa `java` (interpretador) que vem com o ambiente de programação da Sun (J2SDK).
- uma instância em tempo de execução (ou instância *runtime*).

A **especificação abstrata** é um conceito descrito em detalhes no livro “*The Java Virtual Machine Specification*” (Lindholm et al., 1997) que define para os desenvolvedores de implementação de MVJ que características uma MVJ deve possuir. **Implementações concretas** existem para várias plataformas e podem vir de vários fornecedores. Elas também podem ser totalmente implementadas em software, ou ainda uma combinação de hardware e software. Por exemplo, o programa `java.exe` que vem no ambiente de programação fornecido pela SUN, o JSDK, é um exemplo de uma implementação da MVJ. Uma **instância em tempo de execução** é quando essa implementação da MVJ está rodando e executando uma aplicação. Por exemplo, quando o interpretador `java.exe` do JSDK está executando um programa Java. Toda a aplicação Java vai ser executada por uma instância da MVJ.

### 1.4.1 O Ciclo de Vida de uma Máquina Virtual Java

Uma instância em tempo de execução (ou instância *runtime*) tem a função de executar uma aplicação Java. Quando uma aplicação Java inicia a sua execução, uma instância *runtime* nasce. Ela morre no momento em que a aplicação completa a sua execução. Se nós executarmos três aplicações Java iguais, usando a mesma implementação de MVJ e na mesma máquina, nós teremos 3 instâncias de MJV executando. Cada aplicação Java irá executar na sua própria MVJ.

A instância da MVJ irá iniciar a execução de sua aplicação pelo método `main()`. O método `main()` deve ter a seguinte assinatura: `public static void main (String args[ ])` (como mostrado no exemplo listado na seção 1.3). Qualquer classe Java que contenha o método `main()` pode ser usada como ponto de partida para a execução de uma aplicação *stand-alone* Java.

Nós temos que, de alguma maneira, fornecer a MVJ o nome da classe que contém o método `main()` por onde a MVJ irá iniciar a execução. Um exemplo real de implementação de MVJ é o programa `java.exe` do ambiente Java 2 SDK da Sun. Para dizer a esta MVJ que nós gostaríamos de executar a classe `HelloWorld`, nós digitamos o comando:

```
C:\>java HelloWorld
```

no prompt do DOS (Windows). O primeiro parâmetro (`java`) está indicando que o sistema operacional pode iniciar a execução da JVM do J2SDK. O segundo parâmetro é o nome da classe inicial. Esta classe deve conter o método `main()`.

A MVJ é formada por um subsistema que carrega classes (*classloader*) que é um mecanismo para carregar as classes e interfaces dado os seus nomes. Ele carrega tanto classes do programa Java criado pelo

programador, também como classes da Java API. Apenas são carregadas as classes da Java API que são utilizadas pelo programa.

A MVJ também possui uma máquina de execução (*execution engine*) que é um mecanismo responsável por executar as instruções Java (*bytecodes*) contidos nos métodos das classes carregadas. Em uma MVJ implementada em software, o tipo mais simples de *execution engine* apenas interpreta os bytecodes um por um. Um outro tipo de *execution engine*, que é mais rápida, é o compilador *just-in-time* (JIT). Um compilador JIT compila os bytecodes para código em máquina nativa na primeira vez que o método é executado. O código nativo é guardado em memória e, então, pode ser reusado na próxima vez que o método é invocado. Ele mantém em memória tanto os bytecodes como o código nativo, por isso, embora seja mais rápido, requer mais memória.

Algumas vezes a JVM é chamada de interpretador já que, quando a MVJ é implementada em software, ela interpreta os bytecodes para executá-los. Porém, algumas MVJs usam técnicas diferentes, como é o caso dos compiladores JIT. Desta maneira, não é muito adequado chamar a MVJ de interpretador (Venners, 2000).

## 1.5 Exercícios

### Exercício 1 – Pede Argumento

Faça um programa Java leia um argumento passado para a Máquina Virtual Java. O programa deve ser formado por uma única classe e contém apenas o método `main ( )`. A classe deve se chamar `PedeArgumento` e você passará o seu nome como argumento. Por exemplo, se o seu nome é Nicolas, você deve executar o programa através do seguinte comando:

```
C:/> java PedeArgumento Nicolas
```

E o programa deve exibir na tela como resultado:



> Olá Nicolas!

---

**Dica:** Todo o parâmetro passado para a MVJ é recebido pela array de String do método `main()`. Para acessar um argumento de uma array, forneça a posição da array que você deseja acessar. Por exemplo, como nós fornecemos apenas uma String ela estará na primeira posição da array que é zero: `String nome = args [0];`

Se você digitasse duas strings (por exemplo, o seu nome e sobrenome, seria necessário acessar as duas primeiras posições da array: 0 e 1).

## 2 Estruturas Fundamentais de Programação em Java

### 2.1 Espaço em Branco

Java é uma linguagem de formato livre. Não é necessário endentar para que ela funcione corretamente. Por exemplo, o nosso programa HelloWorld poderia ser escrito da seguinte maneira:

```
public class HelloWorld{public static void main (String a[ ] )
{System.out.println ("Oi, mundo!");}}
```

Devemos, porém, tomar cuidado para que haja pelo menos um espaço, Tab ou uma nova linha entre palavras que não estejam separadas por um operador ou separador.

### 2.2 Comentários em Java

Os comentários em código-fonte Java podem ser de 3 tipos: de uma linha, de várias linhas e de documentação. Os comentários de uma linha começam com `//` (duas barras) e terminam no final da linha. Este tipo de comentário é útil para explicações de uma única linha de código. Nas linhas 7 e 8 do código exibido no início deste capítulo encontramos dois comentários desse tipo.

O comentário de várias linhas é usado para comentários mais longos que uma linha e é iniciado com `/*` (barra-asterisco) e finalizado com `*/` (asterisco-barra). Tudo que houver entre `/*` e `*/` é considerado um comentário e, portanto, ignorado pelo compilador.

Existe ainda o comentário de documentação que começa com `/**` (barra-asterisco-asterisco) e finaliza com `*/` (asterisco-barra). O uso da ferramenta Javadoc para gerar documentação HTML vai ser explicado no capítulo 7 (página 58).

### 2.3 Palavras-chave reservadas

As palavras-chave reservadas Java são usadas para identificar os tipos, modificadores e mecanismos de controle de fluxo. Essas palavras, juntamente com os operadores e separadores, formam a definição da linguagem Java. Elas não podem ser usadas como nome de variável, método ou classe.

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

## 2.4 Identificadores

Os identificadores são as palavras usadas para nomes de classes, métodos e variáveis. Um identificador pode ser qualquer sequência de caracteres de letras maiúsculas e minúsculas, números e caracteres de sublinhado e símbolo de cifrão. Mas, tome os seguintes cuidados:

- Eles não podem começar com números para não serem identificados como literais numéricos;
- Java é sensível a maiúsculas e minúsculas, por isso o identificador *Valor* é diferente de *valor*.

## 2.5 Convenções para nomes de identificadores

### 2.5.1 Variáveis e métodos

Segundo a convenção para identificadores Java, os métodos e variáveis devem ser nomeados com letras minúsculas. No caso do identificador ser formado por mais de um termo, o segundo termo e os termos seguintes devem iniciar com letra maiúscula. As variáveis são compostas por substantivos e adjetivos, enquanto que os nomes de métodos começam sempre com um verbo.

Exemplos: hora, horaDoDia, valorCorrente, obterHoraDoDia().

### 2.5.2 Constantes

Para as variáveis finais, que representam constantes em Java, usa-se todo o nome em letras maiúsculas. Quando as constantes forem formadas por mais de um termo, usa-se sublinhado para separá-los. Os nomes de constantes Java são formados por substantivos e adjetivos.

Exemplos: VALOR\_P1, DIA\_SEXTA, VERDE.

### 2.5.3 Classes

Nomes de classes em Java são escritos em minúsculo com a primeira letra em maiúscula. Para os nomes compostos por mais de um termo, usa-se começar os termos seguintes com letra maiúscula. Os nomes de classes Java são formados por substantivos e adjetivos.

Exemplos: InterfaceSimples, Interface.

## 2.6 Declaração Import

Para utilizar os pacotes Java, usa-se a declaração de importação que define onde o compilador pode encontrar as classes destes pacotes. A declaração de importação (**import**) deve preceder a declaração de todas as classes.

O compilador irá procurar por pacotes dentro dos diretórios especificados na variável de ambiente classpath. Um pacote pode ainda estar compactado dentro de um arquivo JAR.

## 2.7 Tipos Primitivos de Dados (Tipos Nativos de Dados)

Java tem oito tipos simples de dados que podem ser classificados em quatro grupos:

- **Inteiros:** byte, short, int e long que são usados para números inteiros;
- **Números de Ponto flutuante:** float e double que correspondem aos números com precisão de fração.
- **Caracteres:** char.
- **Valores lógicos:** boolean que é um tipo especial usado para representar valores lógicos.

Grupo	Tipo	Tamanho	Intervalo de Valores	Valor Default
Inteiros	int	4 bytes	-2.147.483.648 até 2.147.483.647	0
	short	2 bytes	-32.768 até 32.767	0
	long	8 bytes	-9.223.372.036.854.775.808L até 9.223.372.036.854.775.807L	0L
	byte	1 byte	-128 até 127	0
Ponto Flutuante	float	4 bytes	+ - 3,40282347E+38F (6-7 dígitos significativos)	0.0f
	double	8 bytes	+ - 1,79769313486231570E+308 (15 dígitos significativos)	0.0d
	char	2 bytes	representa um Unicode	'\u0000';
	boolean	1 bit	true ou false	false

## 2.8 Conversões entre Tipos Numéricos

Qualquer operação numérica em variáveis de tipos diferentes será realizada da seguinte maneira:

- Se um dos operandos é do tipo double, o outro será tratado como um double no escopo da operação;
- Senão, se um dos operandos for float, o outro será tratado como float;
- Senão, se um dos operandos é do tipo long, o outro será tratado como long.
- Senão, os operandos serão tratados como inteiros.

Mas em alguns casos pode ser necessário converter, por exemplo, um double em um inteiro. Toda conversão numérica em Java é possível, mas informações podem ser perdidas. Estas conversões podem ser realizadas com o uso de cast. A sintaxe do casting é colocar o tipo desejado entre parênteses antes do nome da variável. Por exemplo:

```
double x = 9,997;
int nx = (int) x;
```

Java permite ainda que certas conversões sejam feitas pela atribuição de valores de um tipo para outro sem que seja necessário o uso de cast. As conversões permitidas de forma automática pelo compilador são:

```
byte → short → int → long → float → double
```

Você pode atribuir o valor de uma variável que está à esquerda para aquela do tipo que está à direita.

Muito embora os chars não sejam usados como inteiros, você pode operá-los como se fossem inteiros. Desta maneira, é necessário o uso de cast para converter inteiros em caracteres. Caracteres podem ser considerados inteiros sem o uso de casting. Por exemplo:

```
int três = 3;
char um = '1';
char quatro = (char) (três+um);
```

A variável quatro finaliza com '4' armazenado nela. Observe que um foi promovido para int na expressão, exigindo a definição de tipo de char antes da atribuição para a variável quatro.

Se você desejar que uma variável inteira receba o valor inteiro que um determinado caractere representa, você pode usar o método digit (char, int) da classe Character, onde passamos como parâmetro o caractere e a base de conversão (decimal, binária, etc). Para o contrário, converter um inteiro para um caractere que represente aquele inteiro, utilize o método forDigit (int, int) onde o primeiro parâmetro representa o valor inteiro a ser convertido e o segundo parâmetro a base de conversão. O exemplo abaixo ilustra essas conversões:

```
public class ConverteCaracteres {
    public static void main (String args []) {
        int i = 3;
        char c = '1';
        // converte char em inteiro. Ex: '1' -> 1
        int res1 = Character.digit (c, 10);
```



```

        // converte inteiro em char. Ex: 1 -> '1'
        char res2 = Character.forDigit (i, 10);
        System.out.println ("char -> int = " + res1);
        System.out.println ("int -> char = " + res2);
    }
}

```

## 2.9 Declaração e Inicialização de Valores

As variáveis do tipo byte, short, int, long, float, double, char e boolean podem ser declaradas de acordo com uma das formas exibidas abaixo.

int a, b, c;	Declarando as variáveis a, b e c.
int d = 3, e, f=5;	Declarando d, e, f e inicializando d com 3 e f com 5.
double pi = 3.14159;	Declarando e inicializando pi com o valor 3.14159;
char x = 'x';	Declarando e inicializando x com o caractere 'x';

As variáveis membros (dentro de uma classe) não inicializadas recebem por default o valor 0 (zero). As variáveis locais (dentro de um método) não são inicializadas por default e, por isso, é gerado um erro de compilação quando não é atribuído nenhum valor inicial a elas e essas são acessadas.

O seguinte programa cria variáveis de cada tipo primitivo e exibe seus valores:

```

class SimpleTypes {
    public static void main (String args [ ]) {
        byte b = 5;
        short s = 5;
        int i = 334;
        long l = 34567L;
        float f = 35.76f;
        double d = 35.76;
        System.out.println ("byte b = "+b);
        System.out.println ("short s = "+s);
        System.out.println ("int i = "+i);
        System.out.println ("long l = "+l);
        System.out.println ("float f = "+f);
        System.out.println ("double d = "+d);
    }
}

```

## 2.10 Constantes

Em Java não podem ser definidas constantes locais para um determinado método como, por exemplo, o main. Ao invés disso, pode haver somente constantes para todos os métodos na classe. Elas são chamadas usualmente de constantes de classe. As constantes são sempre definidas através dos modificadores static final. Por exemplo:

```

public class UsesConstants {
    public static final double GRAVIDADE = 32;
    public static void main (String args [ ]) {
        System.out.println ("A gravidade é " +GRAVIDADE);
    }
}

```

## 2.11 Operadores

Existem quatro tipos de operadores em Java: aritméticos, bit-a-bit, relacional e lógico.

### 2.11.1 Operadores Aritméticos

Os operandos aritméticos só podem ser usados para operandos do tipo numérico. Não podem ser usados esses operadores para os tipos boolean, mas podem para char, uma vez que char é um subconjunto do tipo int.

Operador	Operação	Exemplo	Resultado
+	adição	x=1+2;	x=3
-	subtração	x=3-1;	x=2
*	multiplicação	x=2*3;	x=6
/	divisão	x=6/2;	x=3
%	módulo (resto da divisão inteira)	x=7%2;	x=1
++	incremento (equivalente a x=x+1)	x=1; x++;	x=2
	equivalente a y=x e x=x+1	x=1; y=0; y=x++;	x=2, y=1
	equivalente a x=x+1 e y=x	x=1; y=0; y=++x;	x=2, y=2
--	decremento (equivalente a x=x-1)	x=1; x--;	x=0
	equivalente a y=x e x=x-1	x=1; y=0; y=x--;	x=0; y=1
	equivalente a x=x-1 e y=x	x=1; y=0; y=--x;	x=0, y=0
+=	Soma e atribui (equivalente a i=i+2)	i=1; i+=2;	i=3
-=	Subtrai e atribui (equivalente a i=i-2)	i=1; i-=2;	i=-1
*=	Multiplica e atribui (equivalente a i=i*2)	i=1; i*=2;	i=2
/=	Divide e atribui (equivalente a i=i/2)	i=2; i/=2;	i=1
%=	Módulo e atribui (equivalente a i=i%2)	i=1; i%=2;	i=1

### 2.11.2 Operadores Relacionais

Para comparar valores são usados os seguintes operadores relacionais:

Operador	Operação
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Os operadores relacionais sempre retornam um valor boolean (true ou false) como resultado. Por exemplo:

```
int a=4;
int b=1;
boolean c = a<b;
```

O trecho de código acima mostra duas variáveis (a e b) sendo comparadas e a variável c recebendo como resultado o valor false.

### 2.11.3 Operadores Lógicos

Os operadores lógicos operam apenas com operandos (variáveis) do tipo boolean. Esses operadores servem para combinar expressões relacionais e devolvem como resultado um valor boolean (true ou false).

Operador	Resultado
&&	AND lógico
	OR lógico
!	Negação

Para os operadores acima, os resultados seguem a tabela exibida a seguir.

A	B	OR (  )	AND (&&)	NOT A (!A)
False	false	false	false	true
True	false	true	false	false
False	true	true	false	true
True	true	true	true	false

Por exemplo, no código abaixo a variável c será incrementada.

```
boolean a=true, b=false;
int c=0;
if (a && !b) c++;
```

## 2.11.4 Operadores Bit-a-Bit

Os tipos numéricos inteiros possuem um conjunto adicional de operadores que podem modificar e inspecionar os bits que formam seus valores. São eles:

Op.	Operação	Op.	Operação
~	NOT bit-a-bit		
&	AND bit-a-bit	&=	atribui e AND
	OR bit-a-bit	=	atribui e OR
^	OR exclusivo	^=	atribui e OR exclusivo
>>	deslocamento à direita (shift right)	>>=	atribui e deslocamento à direita
>>>	deslocamento à direita com zeros	>>>=	atribui e deslocamento à direita com zeros
<<	deslocamento à esquerda (shift left)	<<=	atribui e deslocamento à esquerda

A tabela a seguir mostra como cada operador bit-a-bit age em cada combinação de bits de operando.

A	B	OR ( )	AND (&)	XOR (^)	NOT A (~A)
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Abaixo, encontram-se alguns exemplos:

**NOT:** O operador unário NOT (~) inverte todos os bits de um operando. Por exemplo, a operação de NOT sobre o número 42:

a	00101010	42
~a	11010101	-43

**AND:** Gera 1 se os bits dos operandos forem 1, caso contrário gera 0 (zero).

a	00101010	42
b	00001111	15
a&b	00001010	10

**OR:** Se um dos bits dos operandos for 1 ele gera 1, caso contrário ele gera 0 (zero).

	00101010	42
	00001111	15
a   b	00101111	47

**XOR:** Combina os bits de tal maneira que se os bits dos operandos forem diferentes gera 1, caso contrário gera 0 (zero).

a	00101010	42
b	00001111	15
a^b	00100101	37

**<< (Deslocamento para a esquerda):** Move todos os bits do operando à esquerda no número de vezes especificado no segundo operando. Para cada bit deslocado, os bits à esquerda são perdidos e as posições à direita preenchidas com zeros. A operação de deslocar à esquerda tem o efeito de multiplicar um número por 2 tantas vezes quanto for efetuado o deslocamento. Além disso, ao se deslocar um bit 1 para a posição mais alta, o número torna-se negativo.

a	00000011	3
a<<1	00000110	6

**>> (Deslocamento para a direita):** Move todos os bits do operando da esquerda para a direita no número de posições de bits especificados pelo operando da direita. Os bits deslocados para direita se perdem.

Esta operação tem o efeito de que cada bit deslocado divide o valor do operando em 2 sem considerar o resto. Ao deslocar para a direita, os bits superiores (da extrema esquerda) são preenchidos com o conteúdo do bit anterior. Isto é chamado de extensão de sinal e serve para preservar o sinal de números negativos.

a	00100011	35
a>>2	00001000	8

**>>> Deslocamento para a direita sem sinal:** Desloca n bits à direita preenchendo os campos superiores (extrema esquerda) com zero.

A	11111111 11111111 11111111 11111111	-1
a>>>24	00000000 00000000 00000000 11111111	255



Todos os tipos de inteiros são inteiros com sinal. Isto significa que eles podem representar tantos valores negativos como positivos. Java usa uma codificação conhecida como complemento de dois, o que significa que os números negativos estão representados pela inversão (mudança de 1's para 0's e vice-versa) de todos os bits de um valor, somando-se depois 1 ao resultado. Por exemplo, 42 é 00101010 que invertendo os 0's e 1's fica 11010101, depois somando 1 resulta em 11010110 ou -42. Para decodificar um número negativo em binário para decimal, primeiro inverta todos os bits e depois some 1.

### 2.11.5 Operador if-then-else ternário

```
expressão ? declaração1 : declaração2
```

Onde expressão é qualquer expressão que o resultado seja um valor boolean. Se o resultado for true, a declaração 1 é executada e se for false a declaração 2 é executada. Por exemplo:

```
ratio = denom==0 ? 0 : num/denom;
```

Na expressão acima é avaliado se denom é igual a zero. Caso positivo ratio recebe zero, senão ratio recebe num/denom.

### 2.11.6 Precedência de Operadores

A tabela a seguir mostra a ordem de todas as operações possíveis em Java, de precedência mais alta para mais baixa. Os operadores que estão no mesmo nível, geralmente, são processados da esquerda para a direita na linha de comando.

Prioridade Mais alta	
! ~ ++ -- +(unário) -(unário) (cast)	direita para a esquerda
* / %	esquerda para a direita
+ -	esquerda para a direita
<< >> >>>	esquerda para a direita
< <= > >= instanceof	esquerda para a direita
== !=	esquerda para a direita
&	esquerda para a direita
^	esquerda para a direita
	esquerda para a direita
&&	esquerda para a direita
	esquerda para a direita
?:	esquerda para a direita
= += -= *= /= %=  = ^= <<= >>= >>>=	direita para a esquerda
Prioridade Menos Alta	

## 2.12 Controle de Fluxo

Nos exemplos vistos até aqui o fluxo do programa era linear, ou seja, a execução tinha que passar por todas as linhas de comando. Os programas, muitas vezes, vão exigir que certas partes do código sejam ou não executados de acordo com algumas condições. Para isto são usadas declarações de desvio, tais como, *if-else*, *while*, *for*, *switch*, que serão estudados neste capítulo.

### 2.12.1 if-else

```
boolean dadosDisponiveis = true;
if (dadosDisponiveis)
    processarDados ( );
else
    esperarMaisDados ( );
```

No exemplo acima o código verifica o valor da variável boolean. Se `dadosDisponiveis` é igual a `true` então executa o método `processarDados`, senão executa o método `esperarMaisDados` ( ).

A cláusula `else` é opcional. Se no exemplo acima não houvesse a cláusula `else`, nenhum método seria executado.

Ao invés de uma variável boolean, também poderia haver uma expressão que usasse operadores para produzir um resultado boolean. Por exemplo:

```
int bytesDisp = 5;
if (bytesDisp > 0) {
    processarDados ( );
    bytesDisp --;
}
else esperarMaisDados ( );
```

No exemplo anterior é verificado se "bytesDisp > 0". Caso positivo retorna o valor boolean true e então executa os comandos entre chaves, processarDados ( ) e byteDisp--. Caso contrário, retorna false e executa esperarMaisDados( ).

### 2.12.2 switch

A declaração switch é usada quando existem procedimentos diferenciados para determinados valores de uma única variável. Ela só pode ser usada para variáveis que sejam de tipo primitivos de dados, mais especificamente para variáveis do tipo int (inteiros) e char (caracteres). Por exemplo:

```
int mes = 4;
switch (mes) {
    case 1: System.out.println ("Janeiro");
            break;
    case 2 : System.out.println ("Fevereiro");
            break;
    case 3 : System.out.println ("Marco");
            break;
    case 4: System.out.println ("Abril");
            break;
    case 5 : System.out.println ("Maio");
            break;
    case 6 : System.out.println ("Junho");
            break;
    case 7: System.out.println ("Julho");
            break;
    case 8 : System.out.println ("Agosto");
            break;
    case 9 : System.out.println ("Setembro");
            break;
    case 10 : System.out.println ("Outubro");
            break;
    case 11 : System.out.println ("Novembro");
            break;
    case 12 : System.out.println ("Dezembro");
            break;
    default : System.out.println ("Não existe mês correspondente!");
}
```

No exemplo acima, será impresso "Abril" como resultado na tela. Isto porque o valor da expressão é comparado com cada um dos valores das declarações case. Se uma correspondência for encontrada, a sequência de código que vem depois da declaração case é executada. Se nenhum dos cases corresponder ao valor da expressão, então a declaração default é executada. Entretanto, a declaração default é opcional.

O break faz com que a execução pule para o final do switch. Se não fosse colocado o break, a execução seria passada para o próximo case até encontrar um break. Esse uso alternativo de cases pode ser útil. Veja o exemplo abaixo:

```
class SwitchEstacoes {
    public static void main (String args [ ]) {
        int mes = 4;
        String estacao;
        switch (mes) {
            case 12:
            case 1:
            case 2:
                estacao = "Verão";
            break;
            case 3:
            case 4:
            case 5:
                estacao = "Outono";
```

```

        break;
        case 6:
        case 7:
        case 8:
            estacao = "Inverno";
            break;
        case 9:
        case 10:
        case 11:
            estacao = "Primavera";
            break;
        default:
            estacao = "Desconhecida";
    }
    System.out.println ("Abril está na estação " + estação + ".");
}
}

```

### 2.12.3 while

A construção *while* executa repetidamente o mesmo bloco de código até que o resultado de uma expressão boolean seja verdadeiro. Por exemplo:

```

class WhileDemo {
    public static void main (String args [ ]) {
        int n = 5;
        while (n>0)
            System.out.println ("tick "+ n--);
    }
}

```

O programa anterior verifica se o valor da variável *n* é maior que zero. Caso positivo, imprime o valor de *n*, e decrementa seu valor. Realiza essa operação repetida vezes até que o valor de *n* seja menor ou igual a zero. O código acima reproduzirá a seguinte saída:



```

c:\> java WhileDemo
tick 5
tick 4
tick 3
tick 2
tick 1

```

### 2.12.4 do-while

Às vezes, é desejável executar o corpo de um laço *while* pelo menos uma vez, quando se quer testar a expressão de encerramento no final do laço ao invés do início (como acontece no *while*). Para isso, usamos o laço *do-while*.

```

class DoWhile {
    public static void main (String args []) {
        int n=5;
        do {
            System.out.println ("tick " + n);
        } while (--n>0);
    }
}

```

O programa acima produz a mesma saída que o anterior.

### 2.12.5 for

A declaração `for` é uma maneira compacta de construir um laço. Sua forma básica segue:

```
for (inicialização; encerramento; iteração) corpo;
```

Por exemplo:

```
class ForDemo {
    public static void main (String args []) {
        for (int i=1; i<=5; i++) {
            System.out.println ("i = "+i);
        }
    }
}
```

O programa acima produz o seguinte resultado:



```
c:\>java ForDemo
i = 1
i = 2
i = 3
i = 4
i = 5
```

Se a condição inicial ( $i=1$ ) não fizer com que o encerramento ( $i \leq 10$ ) retorne *true* pela primeira vez, então as declarações *corpo* (`System.out.println...`) e *iteração* ( $i++$ ) nunca serão executadas. A ordem de execução do `for` é:

- atribui valor inicial ( $i=1$ )
- verifica a condição de encerramento ( $i \leq 10$ )
- se *true*
- executa corpo do laço (`System.out.println...`)
- executa iteração ( $i++$ )
- volta ao passo 2
- se *false* → **fim**

### 2.12.6 continue

O *continue* é usado em laços quando se deseja, em uma determinada condição, pular para a próxima iteração. Por exemplo:

```
class ContinueDemo {
    public static void main (String args []) {
        for (int i=0; i<10; i++) {
            System.out.print (i + "");
            if (i%2 == 0) continue;
            System.out.println ("");
        }
    }
}
```

O laço do programa acima imprime dois números por linha. Ele usa o `System.out.print` para imprimir sem gerar uma quebra de linha. Depois, ele usa o `%` (operador de módulo) para verificar se o número é par ou ímpar (se a divisão por 2 não gerar resto é par). Se o número é par, ele pula para a próxima iteração do laço sem passar pelo comando de impressão de uma nova linha.

O resultado pode ser visualizado a seguir:





```
c:\>java ContinueDemo
0 1
2 3
4 5
6 7
8 9
```

### 2.12.7 break

O termo *break* é usado para interromper um bloco de código, ou seja, para sair desse bloco. É usado no *switch*, como vimos na seção 2.12.2, e para interromper laços (for, while, do-while).

```
class TestaBreak {
    public static void main (String args []) {
        int a = 0;
        while (true) {
            if (a == 5) break;
            System.out.println ("O valor de a é = " + a++);
        }
    }
}
```



```
c:\jdk1.3\bin\java TestaBreak
O valor de a é = 0
O valor de a é = 1
O valor de a é = 2
O valor de a é = 3
O valor de a é = 4
```

O *break* também é usado como uma substituição da declaração *goto*, contida em muitas linguagens. Para esse uso, devemos declarar um rótulo para onde a execução será desviada após passar pelo comando *break*.

Para nomear um bloco, é preciso escolher um identificador permitido pela linguagem e o colocar antes do bloco que se deseja rotular, o qual é seguido de dois pontos (:). Depois você pode referenciar esse rótulo como um argumento para uma declaração *break*, fazendo com que a execução continue depois do final do bloco rotulado.

```
Class TestaBreakRotulado {
    public static void main (String args []) {
        loop1: while (true) {
            for (int i=0; i<10; i++) {
                if (i==7) break loop1; // vai para o final do bloco do loop1
                System.out.println ("i= "+i);
            }
        } // do while do loop1
        loop2: while (true) {
            System.out.println ("Voltou para o while do loop2!");
            for (int j=0; j<10; ++j) {
                if (j==3) break loop2;
                for (int k=0; k<10; k++) {
                    // Volta a execução para o laço 2
                    if (k==3) continue loop2;
                    System.out.println ("k= "+k);
                }
            }
        } // do while do loop 2
    } // da void main
} // da classe
```



```
c:\jdk1.3\bin\>java TestaBreakRotulado
i= 0
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
Voltou para o while do loop2!
k= 0
k= 1
k= 2
Voltou para o while do loop2!
k= 0
k= 1
k= 2
Voltou para o while do loop2!
k= 0
k= 1
k= 2
Voltou para o while do loop2!
<Entra em loop por causa do comando continue loop2>
```

## 2.13 Exercícios

### Exercício 1: Fatorial

O fatorial de um número é calculado multiplicando-se os valores de um até o valor especificado. Um número fatorial é representado pelo número do fatorial seguido de um ponto de exclamação. Por exemplo o fatorial de 4 é representado por 4! e é igual a :  $1 \times 2 \times 3 \times 4 = 24$ . Faça um programa java que calcule a fatorial de 2, 4, 6 e 10. Dica : Use o laço for.

### Exercício 2: Distância entre dois pontos

Escreva uma classe Java que contenha apenas o método main( ). Você deve passar como parâmetro para o método main( ) as coordenadas x, y de pontos. O método calcula a distância Euclidiana ente estes dois pontos e exibe o resultado na tela.

Dicas:

Distância Euclidiana entre 2 pontos P e Q : 
$$d(P, Q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Para calcular o quadrado de um número, utilize o método pow da classe Math :

$3^2=9 \rightarrow \text{double res} = \text{Math.pow}(3,2);$

Para calcular a raiz quadrada de um número, utilize o método sqrt da classe Math:

raiz quadrada de 9  $\rightarrow \text{double res} = \text{Math.sqrt}(9);$

### 3 Fundamentos da Orientação a Objetos

A orientação a objetos é uma tecnologia de desenvolvimento composta por metodologias e linguagens usada na análise, no projeto e na implementação de programas (Albuquerque, 1999).

Para entendermos os conceitos de orientação a objetos é necessário entendermos o que é um objeto. Por exemplo, consideremos objetos do mundo real: seu cachorro, seu computador, sua bicicleta.

Os objetos do mundo real têm duas características: eles tem *estado* e *comportamento*. Por exemplo, cachorros tem estados (nome, cor, tamanho) e comportamentos (latir, andar, correr).

Objetos de software, assim como objetos do mundo real, possuem estado (variáveis) e comportamento (métodos). Podemos representar objetos do mundo real através de objetos de software (por exemplo, um livro), assim como também podemos modelar conceitos abstratos (por exemplo, uma ação do usuário em uma interface gráfica).

Por exemplo, considere um objeto que representa uma conta corrente em um sistema bancário. Através dos métodos, pode-se, por exemplo, depositar e sacar. Além dos métodos responsáveis pelos serviços, o objeto deve conter dados, como o nome do dono da conta e o valor do saldo. Como estes dados geralmente estão protegidos, e não podem ser acessados diretamente, o acesso aos dados é feito através dos métodos.

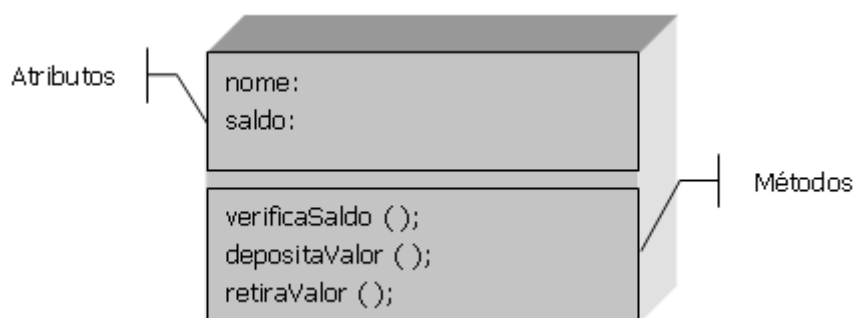


Figura 1: Exemplo de um Objeto

#### 3.1 Classes e Objetos

Em um programa orientado a objetos normalmente existem vários objetos de um mesmo tipo. Por exemplo, um programa de controle de estoque de uma livraria pode conter vários clientes e vários livros.

As **classes** descrevem as informações armazenadas e os serviços providos por um objeto. Em outras palavras, são padrões a partir dos quais os objetos são criados (código).

Em um programa orientado a objetos, um **objeto** é uma instância de uma classe. Os objetos que são instâncias de uma mesma classe armazenam os mesmos tipos de informações e apresentam o mesmo comportamento.

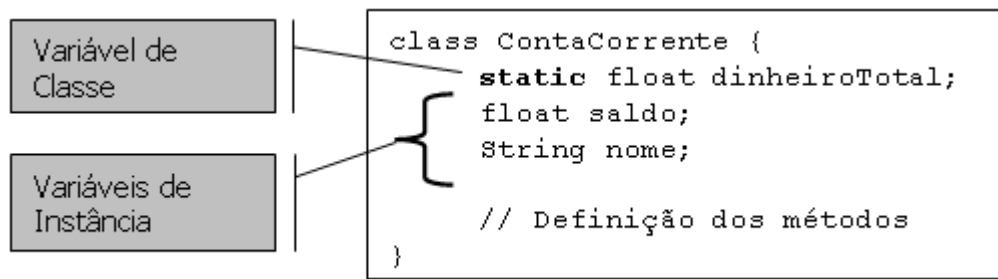
Em Java uma classe é definida pela palavra-chave *class* seguida do nome da classe. Para ilustrar os novos conceitos que serão vistos neste capítulo, vejamos o exemplo da conta corrente.

A conta corrente é uma classe que é definida da seguinte maneira:

```
class ContaCorrente {  
    // Definição dos métodos e atributos  
}
```

### 3.1.1 Variáveis

Cada variável tem um nome e um tipo que define o comportamento estático do objeto. Em Java para cada variável são especificados: um modificador (opcional), o nome, o tipo e o valor inicial (opcional) do atributo. Por exemplo, a classe *ContaCorrente* possui as variáveis *saldo* (saldo da conta corrente) e *nome* (nome do dono da conta). As variáveis de uma classe podem ser de instância ou de classe. Cada classe possui as suas próprias cópias das **variáveis de instância**, enquanto que as **variáveis de classe** são compartilhadas.



Cada instância da classe *ContaCorrente* (objeto) terá as mesmas variáveis (*saldo* e *nome*), porém, elas podem ter valores diferentes em cada instância. A variável de classe (*dinheiroTotal*), porém, é compartilhada por todos os objetos. Por isso, possui o mesmo valor para todos os objetos e se um objeto modificar o seu valor, o novo valor é visto por todos os outros objetos.

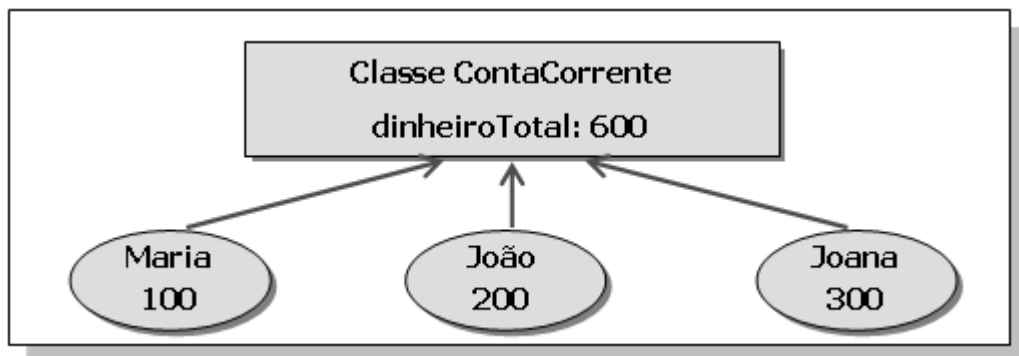


Figura 2: Múltiplas instâncias de uma classe

### 3.1.2 Métodos

Os métodos definem os serviços que podem ser solicitados a uma instância (objeto), ou seja, o comportamento dinâmico de um objeto. A definição de um método em Java inclui um modificador (opcional), o tipo do dado retornado após a execução do método, o nome do método, o nome e tipo dos parâmetros e o código delimitado por chaves ( `{ }` ).

Por exemplo, na classe *ContaCorrente* podem ser definidos os métodos:

- `verificaSaldo`: retorna o saldo da conta corrente;
- `depositaValor`: deposita um valor especificado *x* na conta;
- `retiraValor`: retira um valor especificado *x* da conta;

A definição da classe *ContaCorrente* com as variáveis e métodos é a seguinte:

```
class ContaCorrente {
    static float dinheiroTotal;
    float saldo;
    String nome;
    float verificaSaldo ( ) {
        return saldo;
    }
    void depositaValor (float valor) {
        saldo = saldo + valor;
        dinheiroTotal += valor;
    }
    void retiraValor (float valor) {
        if (saldo >= valor) {
            saldo = saldo - valor;
            dinheiroTotal -= valor;
        }
    }
}
```

### 3.1.3 Métodos Construtores

Quando uma classe é criada, pode ser necessário atribuir um valor inicial a algumas variáveis ou chamar um método. Isto pode ser feito através de um método construtor, o qual é automaticamente executado toda vez que a classe é instanciada.

Em Java os construtores têm o mesmo nome da classe a qual são membros. O método construtor não tem tipo de retorno, isso porque o tipo implícito de retorno de um objeto construtor é a instância da própria classe.

É importante lembrar que **não** é obrigatório criar um construtor. Caso não exista a declaração de um construtor, o compilador gera automaticamente um construtor *default*, assim como pode ser observado no exemplo anterior.

Uma mesma classe pode ter vários métodos construtores, desde que eles tenham quantidade diferente de parâmetros ou parâmetros de tipos diferentes.

Na classe `ContaCorrente` podemos definir um método construtor que seta o saldo em zero. A classe `ContaCorrente` ficaria da seguinte maneira:

```
class ContaCorrente {
    static float dinheiroTotal;
    float saldo;
    String nome;
    public ContaCorrente (String nomeDono) {
        nome = nomeDono;
        saldo = 0;
    }
    float verificaSaldo ( ) {
        return saldo;
    }
    void depositaValor (float valor) {
        saldo = saldo + valor;
        dinheiroTotal += valor;
    }
    void retiraValor (float valor) {
        if (saldo >= valor) {
            saldo = saldo - valor;
            dinheiroTotal -= valor;
        }
    }
}
```

### 3.1.4 Finalização de um Objeto

A memória de um objeto é automaticamente liberada quando o objeto não é mais utilizado. Isso é feito por um processo chamado *Garbage collection*. O Garbage Collection libera os recursos de memória utilizados pelo objeto, mas estes objetos podem possuir outros recursos tais como arquivos abertos e conexões de rede. Nesse caso, poderia ser interessante ter um método que realizasse a finalização de recursos, tais como, fechar arquivo e finalizar conexões de rede.

Para isso, existe o método *finalize*. Esse método não recebe argumentos e não retorna valor. Só pode existir um método finalize por classe e deve ser nomeado *finalize ( )*. O método finalize é chamado algum tempo, não preciso, após o objeto tornar-se não usado, mas seguramente antes do Garbage Collection. Como, muitas vezes, um interpretador Java pode finalizar a sua execução sem o Garbage Collection liberar todos os recursos, não há garantia de que o método finalize será chamado. Desta maneira, esse método não é muito utilizado.

### 3.1.5 Instanciando uma classe

Em um programa Java, os objetos são criados (as classes são instanciadas) através do operador *new*. A execução do comando *new* cria um objeto de um determinado tipo (classe) e retorna uma referência a este objeto. Por exemplo:

```
ContaCorrente minhaConta;  
minhaConta = new ContaCorrente ("Maria");
```

Na primeira linha tem-se o nome da variável (*minhaConta*) na qual o identificador do objeto será armazenado. Na segunda linha é criado o objeto do tipo *ContaCorrente* com o comando *new* que retorna o identificador do objeto que é armazenado na variável *minhaConta*. O enunciado acima poderia ser escrito também da seguinte maneira:

```
ContaCorrente minhaConta = new ContaCorrente ("Maria");
```

### 3.1.6 Operador this

Java inclui um valor de referência especial, chamado *this*, que é usado dentro de qualquer método para referir-se ao objeto corrente. O valor de *this* refere-se ao objeto do qual o método corrente foi chamado.

Em Java é permitido que variáveis locais (dentro de um método) tenham o mesmo nome de variáveis de instância. Para definir que se está acessando a variável de instância, usa-se o *this*. Por exemplo, a classe *ContaCorrente* poderia ter os seus métodos construtores declarados da seguinte maneira:

```
class ContaCorrente {  
    static float dinheiroTotal;  
    float saldo;  
    String nome;  
    public ContaCorrente (String nome) {  
        this.nome = nome;  
        saldo = 0;  
    }  
    ... // restante do código  
}
```

### 3.1.7 Acessando métodos e variáveis de um objeto

Após criar um objeto, queremos manipular os seus dados acessando as variáveis e métodos deste objeto. Os métodos e variáveis são chamados de uma instância de outra classe através de operador ponto ( *.* ).

Por exemplo, digamos que criamos um objeto do tipo `ContaCorrente` e queremos modificar os seus valores, definindo o nome do dono da conta e modificando e visualizando o saldo.

```
class AcessaContaCorrente {
    public static void main (String args [ ]) {
        ContaCorrente minhaConta = new ContaCorrente ("Maria");
        float saldo;
        saldo = minhaConta.verificaSaldo ( );
        System.out.println (saldo);
        minhaConta.depositaValor (200);
        saldo = minhaConta.verificaSaldo ( );
        System.out.println (saldo);
    }
}
```

O código acima cria um objeto do tipo `ContaCorrente` com valores iniciais de "Maria" para nome e 1000 para valor. Após, verifica o saldo da conta, deposita 200 e verifica novamente o saldo.

## 3.2 Sobrecarga de Métodos (Overloading)

É possível, em Java, e, muitas vezes, desejável criar métodos com mesmo nome, mas lista de parâmetros diferentes. Isto é chamado de sobrecarga de métodos (ou *overloading*) e é reconhecido em tempo de compilação. É devido a esta propriedade que podemos ter dois métodos construtores com parâmetros diferentes.

Por exemplo, na classe `ContaCorrente` podemos ter o método construtor sobrecarregado:

```
class ContaCorrente {
    static float dinheiroTotal;
    float saldo;
    String nome;
    public ContaCorrente (String nome) {
        this.nome = nome;
        saldo = 0;
    }
    public ContaCorrente (String nome, float saldo) {
        this.nome = nome;
        this.saldo = saldo;
    }
    ... // restante do código
}
```

## 3.3 Passagem de Parâmetros em Java

A passagem de parâmetros em Java é por valor e não por referência. Por exemplo, não é possível alterar o valor de um parâmetro recebido do tipo primitivo dentro de um método, pois os dados primitivos são passados por valor. Isso significa que o método não tem acesso a variável que foi usada para passar o valor.

Quanto aos objetos, as referências aos objetos também são passadas por valor. Desta maneira, você não pode alterar a variável que referência um objeto, ou seja, não pode fazer com que a variável que referencia o objeto aponte para outro objeto. Mas, pode-se alterar o conteúdo do objeto a qual essa variável referencia, alterando o valor de um de seus atributos. Para entender melhor, veja o exemplo a seguir:

```
class TestaPassagemParametros{
    public static void trocaValorPrimitivo (int num){
        num = num +6;
    }
    public static void trocaValorObjeto (ContaCorrente minhaConta){
        minhaConta.saldo = 300;
    }
    public static void trocaReferenciaObjeto (ContaCorrente minhaConta){
        minhaConta = new ContaCorrente ("Mulher Maravilha", 200);
    }
}
```

```

    }
    public static void main (String args []){
        int val = 11;
        TestaPassagemParametros.trocaValorPrimitivo (val);
        System.out.println ("val = "+val);

        ContaCorrente minhaConta = new ContaCorrente ("SuperHomem");
        System.out.println (minhaConta.saldo);
        TestaPassagemParametros.trocaValorObjeto (minhaConta);
        System.out.println (minhaConta.saldo);
        TestaPassagemParametros.trocaReferenciaObjeto (minhaConta);
        System.out.println (minhaConta.nome);
    }
} // da class

```



```

c:\jdk1.3\bin\java.exe    TestaPassagemParametros
val = 11
0
300
SuperHomem

```

E para o método `trocaReferenciaObjeto` nós passássemos um objeto `String` em vez de um objeto `ContaCorrente`, o que você acha que aconteceria?

O mesmo que aconteceu para o objeto `ContaCorrente`. Não é atribuído o novo valor para a string. Isto acontece porque em Java uma nova instância é criada para toda a literal `String`. Desta maneira, quando fizemos

```
str = "azul";
```

é igual a fazer:

```
str = new String ("azul");
```

Por isso, é possível manipular um literal `String` como se fossem objetos `String`. Por exemplo:

```
int tam = "abc".length ( );
```

### 3.4 Visibilidade

Em Java existem modificadores para **determinar a visibilidade das variáveis e métodos**. Por exemplo, uma variável ou método *public* pode ser acessada por outros objetos, porém se for *private* somente pode ser chamada dentro do próprio objeto. Um bom hábito de programação consiste em deixar privado (*private*) tudo aquilo que não precisa ser acessado por um método de outro objeto. Veja na tabela abaixo, os modificadores existentes em Java:

Encapsulamento	Visibilidade
<b>private</b>	Podem ser acessados somente por métodos da mesma classe.
<b>protected</b>	Podem ser acessados pelas classes do mesmo pacote, como também pelas subclasses do mesmo pacote e de outros pacotes.
<b>sem modificador</b>	Podem ser acessados por subclasses do mesmo pacote ou por outras classes do mesmo pacote.
<b>public</b>	Podem ser acessados de qualquer classe.

Por exemplo, a classe `ContaCorrente` poderia ter os seus dados privados e os métodos públicos para que outras classes possam chamá-los. Desta maneira, os dados só podem ser acessados apenas pelo código dos métodos. Essa propriedade de proteger os dados privados com métodos públicos é chamada de **encapsulamento**.

A classe `ContaCorrente` ficaria:

```

class ContaCorrente {
    private static float dinheiroTotal;

```



```
private float saldo;
private String nome;
public ContaCorrente (String nomeDono) {
    nome = nomeDono;
    saldo = 0;
}
public float verificaSaldo ( ) {
    return saldo;
}
public void depositaValor (float valor) {
    saldo = saldo + valor;
    dinheiroTotal += valor;
}
public void retiraValor (float valor) {
    if (saldo >= valor) saldo = saldo - valor;
    dinheiroTotal -= valor;
}
}
```

## 3.5 Herança

Chamamos de herança quando novas classes herdam propriedades (dados e métodos) de outras classes existentes. Esse é um importante recurso da orientação a objetos que permite a reutilização de código. As classes que herdam as propriedades são chamadas *subclasses* e a classe pai é chamada *superclasse*. Em Java é usada a palavra *extends* para implementar a herança.

Por exemplo, digamos que temos uma classe Ponto que implemente um ponto com duas coordenadas x e y e queremos construir uma classe Ponto3D que represente um ponto tri-dimensional (com 3 coordenadas).

```
class Ponto {
    int x, y;
    public Ponto (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Ponto ( ) {
        this (-1, -1); // this também pode ser usado para chamar construtores
    }
    public void print ( ) {
        System.out.print (x +", "+ y);
    }
}
class Ponto3D extends Ponto {
    int z;
    public Ponto3D (int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public Ponto3D ( ) {
        this (-1, -1, -1);
    }
}
```

Na declaração acima, Ponto é a superclasse e Ponto3D é a subclasse. As variáveis x e y não precisam ser declaradas em *Ponto3D*, pois foram herdadas de *Ponto*.



Métodos construtores não são herdados!

### 3.5.1 super

No exemplo anterior, a classe *Ponto3D* possui uma certa quantidade de código repetido. Por exemplo, o código que inicializa *x* e *y* está em ambas as classes:

```
this.x = x;  
this.y = y;
```

Nesse caso, podemos usar a declaração **super** que faz a chamada do método construtor da superclasse. Uma versão melhorada da classe *Ponto3D* seria:

```
class Ponto3D extends Ponto {  
    int z;  
    public Ponto3D (int x, int y, int z) {  
        super (x, y); // chama o construtor Ponto (x, y)  
        this.z = z;  
    }  
    public Ponto3D ( ) {  
        this (-1, -1, -1);  
    }  
}
```

A referência *super* também pode ser usada para chamar métodos da superclasse. O *super* pode ser usado como uma versão do *this*, ao que se refere à superclasse. Por exemplo, a declaração *super.print ( )* significa chamar o método *print* da superclasse da instância *this*. Veja o método *printPonto3D* abaixo:

```
class Ponto3D extends Ponto {  
    int z;  
    public Ponto3D (int x, int y, int z) {  
        super (x, y); // chama o construtor Ponto (x, y)  
        this.z = z;  
    }  
    public Ponto3D ( ) {  
        this (-1, -1, -1);  
    }  
    public void printPonto3D ( ) {  
        super.print ();  
        System.out.print ("", "+z");  
    }  
    public static void main (String args []) {  
        Ponto3D p3 = new Ponto3D ( );  
        p3.printPonto3D ( );  
    }  
}
```

## 3.6 Sobreposição (Overriding)

Como pode ser observado no exemplo anterior, foi necessário reescrever o método *print* na classe *Ponto3D* para que ele agora imprimisse a coordenada do ponto em 3 eixos (*x*, *y*, *z*). Podemos utilizar a propriedade de sobrecarga e reescrever o método *print* na classe *Ponto3D* com o mesmo nome ao invés de outro como utilizado (*printPonto3D*). Mas, observe que o método deve ter a mesma assinatura (nome, argumentos e valor de retorno), senão não se trata de redefinição e sim de sobrecarga (overloading) do método.

A classe Ponto3D ficaria:

```
class Ponto3D extends Ponto {
    int z;

    public Ponto3D (int x, int y, int z) {
        super (x, y); // chama o construtor Ponto (x, y)
        this.z = z;
    }
    public Ponto3D ( ) {
        this (-1, -1, -1);
    }
    public void print ( ) {
        System.out.println ("\nMétodo print da classe Ponto3D");
        super.print ( );
        System.out.print (" ", "+z");
    }
    public static void main (String args []) {
        Ponto3D p3 = new Ponto3D ( );
        p3.print ( );
    }
}
```

O programa acima produz o seguinte resultado:



```
c:\jdk1.3\bin\java.exe    Ponto3D
Método print do Ponto3D
-1, -1, -1
```

Você pode observar que foi chamado o método *print* da classe Ponto3D. Porém, se chamássemos um método da classe Ponto que não fosse sobrescrito na classe Ponto3D (por exemplo, se não existisse o método *print* na classe Ponto3D), automaticamente, seria chamado o método da superclasse (Ponto).


### 3.6.1 Dinamic Binding

Quando chamamos o método *print* do objeto referenciado pela variável *p3*, em tempo de compilação é verificado se esse método está definido na classe Ponto3D que é o tipo deste objeto. Porém, em tempo de execução, a referência do objeto poderia estar se referindo a alguma subclasse do tipo de referência declarado. Nesse caso, a decisão de qual método será executado é feita em tempo de execução (dinamic binding). Essa propriedade do objeto decidir qual método aplicará para si na hierarquia de classes é chamada de **polimorfismo**.

Para entender melhor, veja o exemplo a seguir:

```
class Polimorfismo {
    public static void main (String args []) {
        Ponto p1 = new Ponto (1,1);
        Ponto p2 = new Ponto3D (2,2,2);
        p1.print ( );
        p2.print ( );
    }
}
```

O programa anterior produz o seguinte resultado:

	<pre>c:\jdk1.3\bin\java.exe Polimorfismo 1, 1 Método print do Ponto3D 2, 2, 2</pre>	Resultado de p1.print ()
		Resultado de p2.print ()

No método main declaramos a variável p2 como sendo do tipo Ponto e depois armazenamos nela uma referência a uma instância da classe Ponto3D. Quando chamamos o método print( ) (p2.print ( ) ), o interpretador Java verifica que, na verdade, p2 é uma instância de Ponto3D e, portanto, chama o método print da classe Ponto3D e não da classe Ponto.

Observe que como Ponto é a superclasse, ele pode referenciar objetos de subclasses, mas o contrário (uma variável do tipo Ponto3D referenciar um objeto do tipo Ponto) gera erro em tempo de compilação.

Por exemplo, podemos fazer:

```
Ponto p1 = new Ponto ( );
Ponto p2 = new Ponto3D ( );
```

Mas **não** podemos fazer:

```
Ponto3D p3 = new Ponto ( );
```

Temos que usar *casting* se queremos que uma variável do tipo Ponto3D receba a referência de um objeto Ponto3D contida em uma variável do tipo Ponto. Caso contrário, será gerado erro em tempo de compilação.

```
Ponto p2 = new Ponto3D ( );
Ponto3D p4 = (Ponto3D) p2;
```



#### Regras para métodos sobrescritos:

Um método **não** pode ser menos acessível que o método que ele sobrescreve. No exemplo explicado anteriormente, não é possível o método print da classe Ponto ser public e o da classe Ponto3D ser private. Mas, o contrário é válido.

## 3.7 O Modificador "final"

Todos os métodos podem ser sobrepostos. Se você quiser declarar que não deseja mais permitir que as subclasses sobreponham métodos de uma classe, deve declara-las como *final*. Por exemplo, se o método *print* da classe *Ponto* fosse declarado final:

```
public final void print ( ) {
```

seria gerado um erro em tempo de compilação, pois ele é redefinido na classe Ponto3D.

Enquanto os métodos final não podem ser sobrepostos, as variáveis declaradas como final não podem ter os seus valores alterados no código do programa. Isso gera erro em tempo de compilação.

## 3.8 Classes Abstratas

Em uma hierarquia é útil padronizar os serviços providos pelas classes. Por exemplo, suponha que você deseje implementar algumas classes que representem polígonos: *Retângulo*, *Quadrado*, *Elipse* e *Triângulo*. Estes polígonos terão dois métodos básicos: *área* ( ) e *circunferência* ( ). Agora, para ser fácil trabalhar com um array de polígonos, seria útil que todos os polígonos possuíssem uma mesma superclasse, *Shape*. Para isso, nós queremos que a classe *Shape* contenha todas as estruturas que nossos polígonos tenham em comum (os

métodos *área ( )* e *circunferência ( )*). Porém, esses métodos não podem ser implementados na classe *Shape*, pois eles têm comportamentos diferentes. Por exemplo, a área de um triângulo é diferente da área de um quadrado. Java lida com essas situações através de *métodos abstratos*.

Java permite definir um método sem implementá-lo declarando o método com o modificador *abstract*. Um método *abstract* não possui corpo; ele possui somente a definição da assinatura (nome, argumentos e tipo de retorno) seguida de um ponto-e-vírgula (;). Toda classe que contém pelo menos um método *abstract* é uma classe abstrata e deve ser declarada como *abstract*. Veja algumas regras para classes abstratas:

- Uma classe abstrata **não** pode ser instanciada.
- Uma subclasse de uma classe abstrata pode ser instanciada somente se implementar todos os métodos abstratos.
- Se uma subclasse de uma classe abstrata não implementa todos os métodos abstratos que herda, então ela também é uma classe abstrata.
- Métodos *static*, *private* e *final* não podem ser abstratos (*abstract*), pois não podem ser sobrescritos na subclasse.
- Classes abstratas podem ter variáveis e métodos não abstratos.

Assim, se a classe *Shape* possui os métodos abstratos *circunferência ( )* e *área ( )*, é necessário que todas as subclasses de *Shape* implementem estes métodos para que possam ser instanciadas. Veja as definições destas classes no exemplo abaixo:

```
public abstract class Shape {
    String cor;
    public abstract double area ( );
    public abstract double circumference ( );

    public String getType ( ) {
        return this.getClass().getName();
    }
} // da class Shape

class Circle extends Shape {
    public static final double PI = 3.141592656358979323846;
    protected double r;

    public Circle (double r) {
        this.r = r;
    }
    public double getRadius ( ) {
        return r;
    }
    public double area ( ) {
        return PI*r*r;
    }
    public double circumference ( ) {
        return 2*PI*r;
    }
} // da class Circle

class Rectangle extends Shape {
    protected double w, h;
    public Rectangle (double w, double h) {
        this.w = w;
        this.h = h;
    }
    public double getWidth ( ) {
        return w;
    }
    public double getHeight ( ) {
        return h;
    }
    public double area ( ) {
```

```

        return w*h;
    }
    public double circunference () {
        return 2*(w+h);
    }
} // da class Rectangle

class ClassesAbstratas {
    public static void main (String args []) {
        Shape [] shapes = new Shape [3];
        shapes [0] = new Circle (2);
        shapes [1] = new Rectangle (1, 3);
        shapes [2] = new Rectangle (4.0, 2);
        double totalArea = 0;
        for (int i=0; i<shapes.length; i++) {
            System.out.print ("O objeto "+i+ " é um ");
            System.out.println ("O objeto "+i+ " é um "+shapes
[i].getType());
            // ou utilizo instanceof
            //if (shapes [i] instanceof Circle) System.out.println ("É um
Circle");
            totalArea += shapes[i].area();
        }
        System.out.println ("Área total de todos os objetos é "+totalArea);
    }
}

```

O programa acima produz o seguinte resultado:



```

c:\jdk1.3\bin\java.exe  ClassesAbstratas
O objeto 0 é um Circle
O objeto 1 é um Rectangle
O objeto 2 é um Rectangle
Área total de todos os objetos é 23.566370625435916

```

Note que uma classe abstrata é implementada através da palavra-chave **extends**:

```
class Rectangle extends Shape {
```

A classe *Rectangle* não é abstrata e por este motivo implementa todos os métodos abstratos da classe *Shape*. Além disso, como a classe *Rectangle* herda todas os métodos e variáveis da superclasse ela pode utilizar o método *getType()* implementado na classe abstrata *Shape*.

## 3.9 Interfaces

A declaração de uma interface é muito semelhante a uma classe, com a diferença de que as palavras *abstract class* são substituídas pela palavra *interface*.

Uma interface, ao contrário de uma classe abstrata, não possui nenhuma implementação, todos os métodos são abstratos. Mesmo que o programador omita o modificador *abstract*, os métodos serão considerados abstratos por *default*.

Todos os métodos de uma interface são públicos, mesmo que o modificador *public* seja omitido. Declarar um método de uma interface como *private* ou *protected* gera erro em tempo de compilação.

Além disso, como uma interface é apenas uma especificação de uma classe e não uma implementação, os métodos não podem ser *final* ou *static*.

Todas as variáveis declaradas em uma interface são constantes, mesmo que não possuam os modificadores *static final*. Além disso, como elas são constantes, elas devem ser inicializadas. Desta maneira,

você pode usar uma interface para compartilhar constantes entre várias classes. Essa utilização da interface poderia ser comparada como usar um arquivo de cabeçalho com um grande número de constantes em C.

Uma interface é declarada da seguinte maneira:

```
interface MinhaInterface {  
    int getSize ( );  
    void setSize (int param);  
}
```

Não é possível criar uma instância a partir de uma interface. Primeiramente, a interface precisa ser implementada. Definimos que uma classe implementa uma interface através da palavra **implements**:

```
class MinhaClasse implements MinhaInterface {  
    int size;  
    public int getSize ( ) {  
        return size;  
    }  
    public void setSize (int tam) {  
        size = tam;  
    }  
}
```

Uma classe pode ser subclasse (extends) apenas de uma classe, mas pode implementar (implements) várias interfaces. Na declaração de uma classe que implementa várias interfaces, os nomes das interfaces devem estar separados por vírgula (,). Além disso, uma classe pode derivar uma única superclasse e implementar várias interfaces. Por exemplo:

```
class MinhaClasse extends MinhaSuperclasse implements MinhaInterface1,  
MinhaInterface 2 {
```

### 3.9.1 Utilizando as Interfaces

As interfaces são bastante usadas na amarração dinâmica de métodos (dynamic binding). Para que o interpretador possa decidir qual classe ele chamará em tempo de execução, todas as classes devem possuir o método para que não seja gerado nenhum erro em tempo de compilação (ver seção 3.6.1).

Na verdade, a nossa classe abstrata Shape (vista na seção 3.8) poderia ser uma interface. Temos que tomar o cuidado, porém, para que nenhum método seja definido em uma interface. Para que a classe abstrata Shape pudesse ser uma interface teríamos que retirar a implementação do método getType () e a variável cor.

Além disso, como em Java não existe conceitualmente a herança múltipla, uma classe pode ser subclasse (extends) apenas de uma única classe. Nas situações em que é útil herança múltipla, são usadas interfaces.

## 3.10 Inner Classes

Inner class, ou nested class, é um conceito que surgiu a partir da versão JDK 1.1. Nas versões anteriores do JDK eram permitidas apenas classes *top-level* que pertenciam a pacotes.

A partir da versão JDK 1.1, o Java suporta classes que são membros de outras classes, ou seja, classes que são definidas dentro do escopo de uma outra classe ou são anônimas, definidas dentro de uma expressão.

Inner classes são um fenômeno do compilador e não do interpretador. Ou seja, o compilador gera o código necessário para que a inner class acesse os dados da enclosing class.

As inner classes tem as seguintes propriedades:

- O nome da inner class tem que ser diferente do nome da enclosing classe (classe o qual está dentro).

- A inner classe pode usar variáveis e métodos da enclosing classe.
- A inner classe pode ser definida como abstract.
- A inner classe pode ser uma interface implementada por outra inner classe.
- Inner classes podem ser declaradas private ou protected. Os modificadores de proteção não previnem a inner classe de acessar variáveis membros de outras classes e da enclosing class.
- Outras classes, que não a enclosing classe, não enxergam a inner classe.
- Inner classes podem acessar variáveis private da enclosing class.
- Se uma inner classe for declarada como static, ela se torna uma top-level classe e perde a capacidade de acessar variáveis da enclosing class.
- Inner classes não podem possuir variáveis membros static. Se a inner classe precisa usar uma variável static, essa variável deve ser declarada na enclosing class.

As inner classes são mais usadas, por conveniência, para criar adaptadores de eventos.

```
/* Código adaptado da apostila da SUN SL-276 */
import java.awt.*;
import java.awt.event.*;

public class TestaInnerClass {
    private Frame f;
    private TextField tf;
    public static void main (String args []) {
        TestaInnerClass inner = new TestaInnerClass ();
        inner.inicio();
    }
    public void inicio () {
        f = new Frame ("Exemplo de Inner Class");
        f.add (new Label ("Clique e arraste o mouse aqui."),
            BorderLayout.NORTH);
        tf = new TextField (30);
        f.add (tf, BorderLayout.SOUTH);
        f.addMouseMotionListener (new Tratador1());
        f.addMouseListener (new Tratador2());
        f.setSize(300, 200);
        f.setVisible(true);
    }
    // A classe Tratador1 é uma inner classe
    public class Tratador1 extends MouseMotionAdapter {
        public void mouseDragged (MouseEvent e) {
            String s = "Mouse Dragging: X = "+e.getX()+" Y = "+e.getY();
            tf.setText (s);
        }
    } // da inner class Tratador1
    public class Tratador2 extends MouseAdapter {
        public void mouseEntered (MouseEvent e) {
            String s = "Mouse entrou";
            tf.setText (s);
        }
        public void mouseExited (MouseEvent e) {
            String s = "Mouse saiu";
            tf.setText (s);
        }
    } // da inner class Tratador2
} // da enclosing class TestaInnerClass
```

### 3.11 Classes Anônimas (Anonymous Class)

É também possível incluir toda a definição de uma classe dentro do escopo de uma expressão, a que chamamos de anonymous class. Por exemplo, vamos redefinir a classe TestaInnerClass para usar uma classe anônima.

```
f.addMouseMotionListener (new MouseMotionAdapter ( ) {
    public void mouseDragged (MouseEvent e) {
```



```

        String s = "Mouse Dragging: X = "+e.getX( )+" Y = "+ e.getY( );
        tf.setText (s);
    }
};

```

Classes Anônimas não possuem métodos construtores. Os parâmetros são repassados ao construtor da classe pai.

## 3.12 Garbage Collection

Quando um programador aloca uma área de memória em C ou C++ é sua responsabilidade liberar a memória quando o programa não precisar mais dela. Se ele não liberar esse espaço de memória, poderá haver falta de memória, ou seja, terá menos espaço de memória de que quando começou. Os problemas de falta de memória são difíceis de serem encontrados e podem fazer com que a máquina trave de repente.

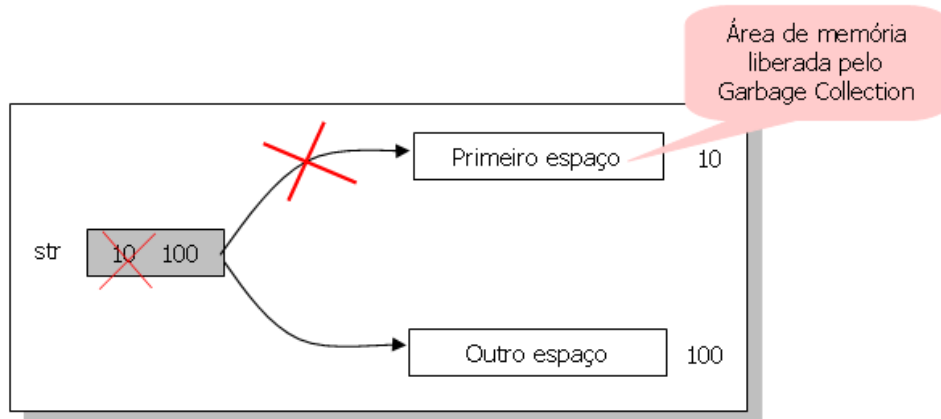
Java resolve o problema de falta de memória com o Garbage Collection (Coleta de lixo). Em Java não é necessário o programador liberar a área de memória de objetos não mais usados, o Java faz isso automaticamente por meio do processo Garbage Collection. Por exemplo, considere o código a seguir que aloca a memória para uma string, a utiliza e depois a libera:

```

String str = "Primeiro espaço";
System.out.println ("Usando memória original: "+str);
str = "Outro espaço";
System.out.println ("Usando outro espaço de memória: "+str);

```

No exemplo acima, Java ocupa uma certa área de memória para colocar a string "Primeiro espaço". Essa string é referenciada pela variável str. Após isso, é criada uma outra string ("Outro espaço") que ocupará uma outra área de memória. A variável str passa a referenciar a nova string, ficando a primeira String sem referência e não mais acessível. Como essa área não será mais utilizada, ela é liberada pela coleta de lixo.



Java executa a coleta de lixo varrendo periodicamente a área de memória em busca de posições que não sejam mais referenciadas. O processo de coleta de lixo é uma thread de baixa prioridade. Você não notará quando ele estiver executando.

## 3.13 Exercícios

### Exercício 1 – Criando a conta do João da Silva

Implemente a classe `ContaCorrente` da seção 3.1.2. Implemente um método `print()` que exiba na tela os dados do dono da conta corrente na classe `ContaCorrente`.

Crie uma outra classe que se chama `AcessaContaJoao`. No método `main()`, crie uma conta para o “João da Silva” com saldo = 1000. Exiba os dados da conta chamando o método `print()`. Faça um saque de 135. Exiba novamente os dados.

## Exercício 2 – Classe Pessoa

Implemente a classe `Pessoa`. Uma pessoa possui: um nome; uma data de nascimento; um endereço. Escreva um método `print()` que exiba na tela os dados da pessoa. No método `main()` da classe, crie dois objetos do tipo `Pessoa`: um que represente você e outro que represente o colega do lado. Exiba na tela os dados da primeira pessoa, apenas acessando os seus dados e os dados da segunda pessoa chamando o método `print()`.

## Exercício 3 – Encapsulamento

Agora que você aprendeu o conceito de encapsulamento, encapsule os atributos da classe `Pessoa`. Os dados devem possuir modificador `private` e seus valores podem ser acessados e modificados por métodos *mutator* (`set...`) e *acessor* (`get...`).

## Exercício 4 – Herança

Implemente a classe `Aluno`. A classe `Aluno` é subclasse da classe `Pessoa`. A classe `Aluno`, além de conter os dados de uma `Pessoa`, vai ter também uma nota e uma turma. Crie métodos *acessor* e *mutator* para os atributos nota e turma.

Escreva um método `print2()` que chame o método `print()` da classe pai (`Pessoa`) que irá exibir nome, data de nasc. e endereço do aluno e exiba na tela a nota e a turma do aluno.

No método `public static void main`, crie 1 objeto aluno que represente você como aluno do curso Java. Que nota você se dá? :o) Exiba os seus dados.

## Exercício 5 – Sobreposição

Na classe `Aluno` (criada no exercício anterior) faça a sobreposição do método `print()` da classe pai (classe `Pessoa`), substituindo o método `print2()`.

## Exercício 6 – Agregação

Implemente a classe `Ponto`. Um ponto tem dois atributos do tipo inteiro: As coordenadas `x` e `y`. Além disso, um ponto também possui os métodos:

- método construtor: Inicializa as coordenadas `x` e `y` com os valores passados como parâmetro para o método.
- `setaCoordenadas(int x, int y)`: que recebe como parâmetro os novos valores das coordenadas.
- `exibe()`: Que exibe os valores dos atributos `x` e `y`.

Implemente a classe `Reta`. Uma `Reta` possui 2 atributos do tipo `Ponto`: o ponto inicial e o ponto final da `Reta`. Uma `Reta` possui os métodos:

- método construtor: recebe 4 valores inteiros e inicializa as coordenadas `x` e `y` de cada ponto.
- `exibe()`: exibe as coordenadas `x` e `y` de cada ponto.

Implemente a classe `TestaReta` que possui apenas o método `main()`. Dentro do método `main()`:

- crie um objeto do tipo `Reta` e inicialize o Ponto Inicial com `x=2, y = 3` e o Ponto Final com `x=5, y=6`.
- chame o método `exibe()` da classe `Reta`.

Neste exemplo, podemos observar uma importante propriedade da orientação a objetos que é a **agregação**. A classe `Reta` tem um relacionamento do tipo agregação com a classe `Ponto`. Incluir objetos de classe como membros de outra classe é conhecido como agregação ou composição. A agregação permite a reutilização de código.

## Exercício 7 – Classes Abstratas

Implemente uma classe Hexagon. Essa classe também herda propriedades da classe abstrata Shape. A classe Hexagon possui apenas um dado que o comprimento do lado  $s$  (todos os lados tem comprimentos iguais) que é uma variável double.

Implemente os métodos `area()` e `circunference()` do Hexagon:

- A área de um hexágono é:  $\frac{3}{2} * \text{Math.sqrt}(3) * s * s$ .
- A circunferência de um hexágono é:  $6 * s$ .

Implemente o método `main()` e crie um hexágono de lado  $s=5$ .

Qual é a área e circunferência deste hexágono? Mostre estas informações na tela.

## 4 Strings

String é uma sequência ou cadeia de caracteres. Java tem uma classe embutida no pacote *java.lang* a qual encapsula a estrutura de dados de uma string. Este pacote não precisa ser importado, já que isso é realizado automaticamente.

Como as strings são objetos, existem métodos para manipulá-las que permitem, entre outras coisas, comparar o conteúdo de duas strings, pesquisar caracteres específicos de uma string, obter seu tamanho, etc.

Além disso, o conteúdo das strings é imutável. Isso quer dizer que o conteúdo da instância String não pode ser alterado depois que ela é criada. Entretanto, uma variável declarada como uma referência a String pode ser alterada para apontar para um outro objeto String. Isto porque as variáveis de referência a um objeto não representam o objeto, mas sim um manipulador (*handle*) deste objeto.

### 4.1 Criando uma string

Uma string pode ser criada através dos seguintes construtores:

#### 4.1.1 Criando uma string vazia:

```
String s = new String ( );
```

#### 4.1.2 Criando uma string a partir de um array de caracteres:

Java permite construir uma string a partir de um array de caracteres:

```
char chars [ ] = { 'a', 'b', 'c' };  
String s = new String (chars);
```

Além disso, podemos criar uma string que tenha como conteúdo uma faixa de caracteres de um array. Para tanto, usa-se o construtor String (char chars[ ], int startIndex, int numChars), onde startIndex corresponde a posição inicial e numChars o número de caracteres a serem usados.

```
char chars [ ] = { 'a', 'b', 'c', 'd', 'e', 'f' };  
String s = new String (chars, 2, 3);
```

O código acima produz a string "cde".

#### 4.1.3 Criando uma String a partir de um array de bytes:

Podemos criar uma string a partir de um array de bytes através do construtor String (byte [ ] bytes).

```
byte bytes [ ] = { 65, 66, 67 };  
String s = new String (bytes);
```

O compilador Java permite instanciar Strings simplesmente definindo um valor literal a ela. Um literal String é um texto arbitrário entre um par de aspas (" "). Os exemplos de string incluem "Oi,Mundo!", "duas\nlinhas" ou "\"Texto entre aspas\"" quando queremos definir um literal string com aspas.

```
String str = "abc"; // cria o objeto str e inicializa com o literal string "abc"
```

```
String str2 = ""; // cria uma string vazia
```

## 4.2 Substituição de Caracteres

Para substituir caracteres de uma string, usa-se o método `replace` da classe `String`. Esse método substitui todas as ocorrências de um determinado caractere por outro. Por exemplo:

```
String str = "Hello";
String resultado = str.replace ('l', 'w'); → "Hewwo"
```

A variável `resultado` receberá a string "Hewwo".

## 4.3 Concatenação de Strings

Em Java o sinal `+` (mais) age como operador de concatenação dos objetos `String`. Por exemplo:

```
String str = "Oi";
String texto = str + "Mundo";
```

Quando um dos operandos não for uma string, Java automaticamente realiza a conversão do tipo para uma representação de `String`. Por exemplo:

```
int três = 3;
String final = 3 + " palavra " + três;
```

O código acima resulta na string "3 palavra 3" armazenada na variável `final`.

Podemos também realizar a concatenação de strings através do método `concat()` da classe `String`. Por exemplo:

```
String pal = "Hello";
pal.concat (" World");
```

Após esta operação, a variável `pal` terá como valor a string "Hello World".

## 4.4 Substring

Para extrair um texto de uma `String`, pode ser usado o método `substring`, que retorna uma nova string com o trecho selecionado nas posições indicadas. Em Java a primeira posição de uma string é zero (0). Por exemplo:

```
String str = "Hello World";
```

H	e	l	l	o		W	o	r	l	D
0	1	2	3	4	5	6	7	8	9	10

```
String resultado = str.substring (6); → "World"
```

No exemplo acima, a variável `resultado` conterá uma nova string formada por todos os caracteres a partir da posição 6 até o final da string contida na variável `str`.

```
String resultado = str.substring (3, 8) → "lo Wo"
```

Nesse último exemplo, a variável `resultado` receberá uma nova string que contém os caracteres que estão da posição 3 até a posição 8 exclusive (sem o caractere da posição 8).

## 4.5 Alterando os caracteres de uma String para maiúsculo

Para converter todos os caracteres de uma string para maiúsculo usa-se o método `toUpperCase ( )`. Por exemplo:

```
String str = "Hello";  
String resultado = str.toUpperCase ( ); → "HELLO"
```

Após a chamada do método, a variável `resultado` receberá a string "HELLO".

## 4.6 Alterando os caracteres de uma String para minúsculo

Para converter todos os caracteres de uma string para minúsculo usa-se o método `toLowerCase ( )`. Por exemplo:

```
String str = "Hello";  
String resultado = str.toLowerCase ( ); → "hello"
```

Após a chamada do método, a variável `resultado` receberá a string "hello".

## 4.7 Retirando espaços em branco

Para retirar espaços em branco ao final e início de uma string, usa-se o método `trim ( )`.

```
String str = " Hello ";  
String resultado = str.trim ( ); → "Hello"
```

Após essa operação, a variável `resultado` receberá a string "Hello".

## 4.8 Extração de Caractere

Para extrair um caractere de uma string, usa-se o método `charAt ( )`. Este método retorna o caractere na posição da string especificada. Exemplo;

```
String str = "Hello";  
char c = str.charAt (1); → 'e'
```

Essa operação retorna o caractere 'e' para a variável `b`.

## 4.9 Comparação de Strings

Para comparar se duas strings são iguais, podemos usar o método `equals` da classe `String`. Existe também o método `equalsIgnoreCase` que compara duas strings ignorando maiúsculas e minúsculas. Esses dois métodos retornam como resultado um valor boolean. Veja os exemplos:

```
String s1 = "Hello";  
String s2 = "HELLO";  
boolean b1 = s1.equals ("Hello"); → true  
boolean b2 = s1.equals (s2); → false  
boolean b3 = s1.equalsIgnoreCase (s2); → true  
boolean b4 = s1.equalsIgnoreCase ("azul"); → false
```



O método *equals* e o operador `==` fazem dois testes de igualdade totalmente diferentes. Enquanto o método *equals* compara os caracteres dentro de uma *String*, o operador `==` compara duas referências de objeto para ver se elas se referem exatamente à mesma instância.

## 4.10 Tamanho de uma string

O tamanho de uma string, ou seja, o número de caracteres pode ser obtido pelo método `length`. Este método retorna um número inteiro.

```
String s = "abc";  
int tam = s.length(); → 3
```

O código acima retorna 3, já que a string *s* possui 3 caracteres.

## 4.11 Identificando a posição de caracteres ou substrings em uma String

Podemos buscar a posição de caracteres ou substrings em uma *String* através dos métodos `indexOf` e `lastIndexOf`. Estes métodos retornam o índice do caractere que está sendo procurado ou índice do início da substring buscada. Em qualquer um dos casos, se a busca não teve sucesso, esses métodos retornam `-1`.

```
String str = "Hello World World2";  
int pos = str.indexOf('l'); → 2
```

retorna o índice da primeira ocorrência de 'l'.

```
int pos = str.lastIndexOf('l'); → 15
```

retorna o índice da última ocorrência de 'l'.

```
int pos = str.indexOf("World"); → 6
```

retorna o índice da ocorrência do primeiro caractere da string "World".

```
int pos = str.lastIndexOf("World"); → 12
```

retorna o índice do primeiro caractere da última ocorrência da string "World".

## 4.12 Ordem

O método `String compareTo (String)` pode ser usado para determinar a ordem de strings: maior que, igual ou menor que. Este método retorna um número inteiro:

- `<0` (negativo): se a string é menor do que parâmetro;
- `>0` (positivo): se a string é maior que o parâmetro;
- `=0` (zero): se as strings são iguais.

O programa a seguir classifica um array de strings usando o método `compareTo (String)` a fim de determinar a ordem de classificação usando o método "Bolha" (*bubblesort*).

```

class SortString {
    public static void main (String args [ ])    {
        String arr [ ] = { "Now", "is", "the", "time", "for", "mem", "to", "come",
                           "to", "aid", "of", "their", "country" };
        for (int j=0; j<arr.length; j++) {
            for (int i=j+1; i<arr.length-1; i++)
                if (arr[i].compareTo(arr[j])<0) {
                    String t = arr [j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            System.out.println (arr[j]);
        } // do for
    } // da main
} // da class

```

O resultado deste programa é a lista de strings classificadas por ordem alfabética.



```

c:\>java SortString
Now
aid
all
come
country
for
good
is
mem
of
the
their
time
to
to

```



Como você deve ter percebido no resultado deste exemplo, o método `compareTo` considera letras maiúsculas e minúsculas. A palavra "Now" veio antes de todas as outras porque começa com letra maiúscula que tem um valor menor no conjunto de caracteres ASCII. Isso ocorre porque esse método compara os valores UNICODE dos caracteres das strings.

Para a comparação de métodos ignorando letras em maiúscula e minúscula usar o método **`compareToIgnoreCase (String)`**.

Para criar rotinas de busca e ordenamento de texto em linguagem natural ver classe **`Collator`** (package `java.text`). Essa classe realiza comparação de String locale-sensitive. Por exemplo, permite comparar palavras não diferenciando palavras acentuadas de não acentuadas.

## 4.13 Conversões de tipos primitivos de dados para Strings

```
String s = String.valueOf (tipo);
```

onde tipo pode ser uma variável do tipo `int`, `long`, `float`, `double`, `boolean`.

## 4.14 Conversão de String para tipos primitivos:

Para converter Strings para tipos primitivos de dados, devemos utilizar os métodos de conversão das classes Wrappers.



De String para inteiro:

```
int i = Integer.parseInt (String);
```

De String para long:

```
long l = Long.parseLong (String);
```

De String para double:

```
double d = Double.parseDouble (String);
```

## 4.15 As classes Wrappers

Os *wrappers* são classes que tem os mesmos nomes que os tipos primitivos de dados, só que, geralmente, com a primeira letra em maiúscula (por exemplo, Long, Double, Float, Short, Integer, Character, Byte) e possuem funcionalidades semelhantes aos tipos primitivos de dados. Os wrappers existem porque em POO não devem existir tipos de dados primitivos, mas apenas objetos e classes. As classes *wrappers* possuem métodos para conversão de Strings em tipos primitivos de dados e vice-versa.

## 4.16 StringTokenizer

Para quebrar uma string em tokens (conjunto de caracteres separado por algum caractere delimitador pré-definido) devemos utilizar a classe StringTokenizer. Essa classe pode ser empregada, por exemplo, na migração de dados em um arquivo texto para banco de dados.

O exemplo abaixo ilustra o uso da classe StringTokenizer para separar tokens de uma string. Este exemplo usa o método construtor que separa os tokens com delimitadores *default* que são: tab, nova linha, *carriage-return*, *form-feed*.

```
import java.util.*;
class BuscaPalavrasEmUmaString {
    public static void main (String args[ ]) {
        String texto = "Isto é um texto de teste";
        // usando delimitadores default
        StringTokenizer st = new StringTokenizer (texto);
        // usando # como delimitador
        // StringTokenizer st = new StringTokenizer (texto, "#");
        while (st.hasMoreTokens( )) {
            String palavra = st.nextToken();
            System.out.println (palavra);
        }
    }
}
```

## 4.17 Exercícios

### Exercício 1 – StringTokenizer

Faça um programa que busca palavras por palavra dentro de um texto e as imprima em maiúscula. Considere um texto uma string inicial.

### Exercício 2 – Gerador de Frases

Faça um programa que gere frases aleatoriamente. Uma frase é formada por uma combinação de um artigo + um substantivo + um verbo + uma preposição + um artigo + um substantivo, sendo que seus valores são lidos de arrays de String. As arrays devem ser povoadas com os seguintes valores:

```
String artigo [] = {"o", "a", "um", "uma", "algum", "qualquer"};
String substantivo [] = {"menino", "menina", "cachorro", "cidade", "carro"};
String verbo [] = {"dirigiu", "pulou", "correu", "caminhou", "saltou"};
String preposicao [] = {"para", "de", "acima de", "debaixo de", "sobre"};
```

Para formar uma frase, o programa deve escolher randomicamente um elemento da respectiva array. O texto deve gerar 10 frases diferentes.

**Dica:** Para gerar randomicamente um número inteiro use o método `nextInt(int)` da classe `Random`. Como a classe `Random` pertence ao pacote `java.util`, você deve importá-lo. Você deve fornecer um parâmetro inteiro para o método que é o valor máximo que pode ser gerado. Por exemplo, se você fornecer 5, ele vai gerar um número entre 0 (inclusive) e 5 (exclusive).

## 5 Arrays

Arrays são grupos de variáveis do mesmo tipo. As arrays podem armazenar variáveis de qualquer tipo (tipo de dado primitivo ou objetos), mas é importante lembrar que todos os valores devem ser de um único tipo de dado.

Em Java arrays são implementadas como objetos, por isso usamos o operador `new` para instanciar uma nova array. Os colchetes (`[]`) são usados para declarar um tipo array.

### 5.1 Criando um array

```
int diaMes [ ] = new int [4];  
int [ ]diaMes = new int [4];
```

No código acima estamos criando um array de 4 posições que armazena variáveis do tipo inteiro. Podemos visualizar um array como um conjunto ordenado de variáveis do mesmo tipo, como mostra a figura abaixo:

Elementos do Vetor	0	0	0	0
Posições do Vetor	0	1	2	3

Uma outra maneira é:

```
int diaMes [ ]; // declarando um array de inteiros (apontador)  
diaMes = new int [4]; //criando um array de int com 4 posições
```

### 5.2 Inicializando arrays

Para inicializar um array é preciso atribuir um valor para cada posição do array. A primeira posição de um array é o índice zero (0). Veja os exemplos:

```
class Array {  
    public static void main (String args [ ]) {  
        int diaMes [ ] = new int [4];  
        diaMes [0] = 31;  
        diaMes [1] = 28;  
        diaMes [2] = 31;  
        diaMes [3] = 30;  
        System.out.println ("Janeiro tem "+ diaMes[0] + "dias.");  
    } // do main  
} // da class
```

O programa acima exibirá:



```
c:\>java Array  
Janeiro tem 31 dias.
```

Um array pode ser inicializado automaticamente da mesma forma que os tipos simples. A inicialização de um array é feita por um conjunto de expressões separadas por vírgula (,) e entre chaves ({ }). O array será

criado com tamanho suficiente para conter o número de elementos inicializados. O array do código acima poderia também ser inicializado da seguinte maneira:

```
class Array {  
    public static void main (String args [ ]) {  
        int diaMes [ ] = {31, 28, 31, 30 };  
        System.out.println ("Janeiro tem "+ diaMes[0] + "dias.");  
    } // do main  
} // da class
```

O programa acima terá saída idêntica ao anterior.

### 5.2.1 Inicialização Default

Quando você cria um array de um tipo de dado primitivo, ele é inicializado por default. Por exemplo, a declaração

```
int diaMes [ ] = new int [4];
```

cria um array com 4 posições (de 0 a 3) com todos os elementos inicializados com zero (0).

## 5.3 Acessando um elemento de uma Array

Para acessar um elemento de um array deve se indicar o índice desejado entre colchetes. Acessar um elemento fora da dimensão do array causa um erro em tempo de execução (`ArrayIndexOutOfBoundsException`). Por exemplo:

```
int vetor [ ] = {2, 4, 6, 6};
```

Elementos do Vetor	2	4	6	6
Posições do Vetor	0	1	2	3

```
v [2] = 3;
```

Essa operação resultará em um vetor com os seguintes valores:

Elementos do Vetor	2	4	3	6
Posições do Vetor	0	1	2	3



Não é possível redimensionar um array, mas apenas eliminá-la da memória e criar uma nova instância. Por exemplo,

```
int v[4] = {0, 1, 2, 3}; //criando um array de 4 posições e inicializado
```

```
v = new int [2]; // fazendo com que v aponte para um novo array de 2 posições
```

Assim, quando fizemos `v = new int [2]`, perdemos todos os valores armazenados anteriormente.

## 5.4 Obtendo tamanho de um array

O tamanho de um array pode ser obtido através do atributo `length`. Por exemplo:

```
int v [ ] = new int [10];
int tam = v.length; → 10
```

## 5.5 Copiando o conteúdo de um array para outro array

Podemos copiar o conteúdo de um array para outro array através do método:

```
System.arraycopy (sourceArray, sourcePosition, destinationArray,
destinationPosition, numberOfEntryToCopy);
```

No exemplo abaixo, estamos copiando 5 elementos da array `vet1` a partir da posição 2 para a array `vet2` a partir da posição 1.

```
System.arraycopy (vet1, 2, vet2, 1, 5);
```

## 5.6 Arrays Multidimensionais

Ao criar um array multidimensional (matriz) em Java, na verdade, estamos criando um array de array.

### 5.6.1 Criando arrays multidimensionais

O código a seguir cria um array multidimensional de 16 posições *double*.

```
double matriz [ ] [ ] = new double [4] [4];
```

A array multidimensional acima pode ser visualizada como uma matriz, assim como mostra a figura a seguir:

	0	1	2	3
0				
1				
2				
3				

### 5.6.2 Inicializando um array multidimensional

Uma matriz pode ser declarada e inicializada da seguinte forma:

```
int m [ ] [ ] = { {0, 1, 2, 3} ,
{4, 5, 6, 7},
{8, 9, 10, 11};
{12, 13, 14, 15} };
```

Uma outra forma de inicializar uma matriz é atribuir um valor para cada posição da matriz. Por exemplo:

```
m [0] [0] = 0;
m[0] [1] = 1;
m [0] [2] = 2;
```

Quando queremos atribuir um mesmo valor para todos os elementos de uma matriz, podemos usar um laço for:

```
for (int i =0; i<m.length; i++)
    for (int j=0; j<m[i].length; j++)
        m [i] [j] = 0;
```

Como um array multidimensional é um array de array, podemos determinar tamanhos diferentes. Por exemplo:

```
int mat [ ] [ ] = new int [2] [ ];
mat [0] = new int [4];
mat [1] = new int [3];
```

	0	1	2	3
0				
1				

## 5.7 Array de Objetos

Podemos criar arrays não só de tipos primitivos de dados, mas também de objetos. Porém, ao criar um array de objeto, estamos criando apenas "referências" para objetos. Desta maneira, é necessário instanciar o objeto para cada posição da array. Por exemplo:

```
Data d [ ] = new Data [3];
d [0] = new Data ( );
d [1] = new Data ( );
d [2] = new Data ( );
```

Ou podemos usar um laço for para otimizarmos essa tarefa:

```
for (int i=0; i<d.length; i++) d [i] = new Data ( );
```

## 5.8 Exercícios

### Exercício 1 - Array de Alunos (Dificuldade Fácil)

Faça uma classe que contenha apenas um método main( ). Nesse método crie uma array de 5 posições do tipo Pessoa. Crie um objeto do tipo Aluno para cada posição da array. Especifique um nome, um endereço, uma data de nascimento, uma nota e uma turma para cada aluno. Imprima os dados de todos os alunos utilizando um laço for.

### Exercício 2 – Classe Disciplina (Dificuldade Difícil)

Faça a classe Disciplina que é composta por um array de alunos e pelo nome da disciplina. Crie métodos para inserir alunos, remover alunos, listar todos os alunos e verificar o aluno mais velho.

## 6 Vector

Em Java é possível criar um array que armazene elementos de tipos de objetos diferentes usando a classe **Vector** do pacote `java.util`. Por exemplo, eu posso armazenar um objeto do tipo `Data` na posição 0 (zero) e do tipo `Pessoa` em outra posição.

### 6.1 Criando um Vector

```
Vector v = new Vector ( );  
Vector v = new Vector (int capacidadeInicial);  
Vector v = new Vector (int capacidadeInicial, int capacidadeIncremento);
```

A declaração acima cria um `Vector` com capacidade inicial de armazenar 10 elementos. Em um `Vector` não é preciso definir tamanho, ele aumenta seu tamanho automaticamente à medida que novos elementos forem definidos. Se não for definido um valor de incremento, ele dobrará de tamanho toda vez que atingir a sua capacidade.

### 6.2 Inserindo objetos em um Vector

#### 6.2.1 Inserindo um elemento em uma determinada posição

```
v.add (3, "azul");  
v.insertElementAt ("azul", 3);
```

No exemplo acima estamos inserindo na posição 3 do `Vector` a string "azul". Todos os elementos a partir da posição 3 serão automaticamente empurrados para a próxima posição.

#### 6.2.2 Inserindo um elemento no final do Vector

```
v.add ("azul");  
v.addElement ("azul");
```

Qualquer um dos comandos acima insere um objeto na última posição do `Vector`.

### 6.3 Verifica a existência de um elemento no Vector

```
boolean b = v.contains ("azul");
```

Verifica se existe a string "azul" no `Vector`. Retorna um valor boolean; *true* caso exista e *false* se o `Vector` não possui o objeto.

Retorna *true* se o elemento existe no `Vector`, determinado pelo método `equals( )` do objeto. Se o objeto não sobrescreve o método `equals( )` da classe `Object`, compara referências de objetos.

## 6.4 Retornando um elemento

```
String str = (String) v.elementAt (3);  
String str = (String) v.get (3);
```

Retorna o elemento do Vector que está na posição 3. Observe que é necessário usar casting para armazenar o objeto retornado em uma variável, já que o Vector armazena os objetos como do tipo Object.

## 6.5 Retornado a posição de um elemento

```
int i = v.indexOf ("azul");
```

Retorna a primeira posição em que o elemento foi encontrado, determinado pelo método equals( ) do objeto. Retorna -1 se o elemento não existe.

## 6.6 Substituindo um objeto em uma determinada posição

```
Object o = v.set (3, "azul");  
v.setElementAt ("azul", 3);
```

Substitui o objeto na posição 3 pela string "azul". O método set (int, object) retorna o objeto que se encontrava previamente na posição indicada.

## 6.7 Removendo um elemento do Vector

```
boolean b = v.remove ("azul");  
boolean b = v.removeElement ("azul");
```

Os comandos acima removem a primeira ocorrência da string "azul" do Vector. Retornam true se o Vector possuía o objeto.

```
String str = (String) v.remove (3);  
v.removeElementAt (3);
```

Remove o elemento da posição 3. O método *remove* retorna o elemento removido.

## 6.8 Tamanho de um Vector

```
int tam = v.size ( );
```

Retorna o tamanho do Vector, ou seja, quantos elementos o Vector possui.

## 6.9 Convertendo um Vector para array

```
Object [ ] elementos = v.toArray ( );
```

Retorna um array de **Object** contendo todos os elementos do Vector.





É importante ressaltar que um objeto Vector só armazena objetos e não armazena tipos primitivos de dados. Para armazenar um tipo primitivo de dado (int, char, boolean, long, short, byte, double) é necessário armazenar seus valores em *wrappers* (Long, Double, Float, Short, Integer, Character, Byte).

Para entender melhor essa seção, implemente a classe a seguir. Esta classe manipula os elementos de um Vector.

```
class ObjetoVetor {
    public static void main (String args [ ])    {
        Vector v = new Vector ( );
        Float f = new Float (32.0f);
        v.addElement (f);
        Integer i = new Integer (12);
        v.addElement (i);
        Float f = (Float) v.elementAt (0);
        Integer i = (Integer) v.elementAt (1);
        System.out.println ("v [0] = "+f.toString());
        System.out.println ("v [1] = "+i.toString());
    } // do main
} // da class
```

A saída será a impressão dos valores armazenados:



```
c:\>java ObjetoVector
v [0] = 32.0
v [1] = 12
```

## 6.10 Exercícios

### Exercício 1 – Vector de Alunos

Refaça o exercício 1 do capítulo sobre array (Array de alunos) utilizando a classe Vector. Não esqueça de importar o pacote `java.util.*`;

## 7 Gerando documentação HTML do código com javadoc

Uma outra forma especial de comentário é usada pela ferramenta javadoc.exe, que gera automaticamente a documentação de uma classe ou interface pública. A convenção de comentários usada para suportar o javadoc é que antes da classe pública, do método e das declarações de variável seja usado `/**` (barra-asterisco-asterisco) para iniciar e `*/` (asterisco-barra) para finalizar o comentário da documentação.

O javadoc reconhece diversas variáveis especiais, as quais são denotadas por um `@` (sinal de arroba) dentro desses comentários.

Tag	Significado
<code>@see</code>	utilizado para especificar uma nota SeeAlso
<code>@author</code>	utilizado para especificar uma nota Autor
<code>@param</code>	Descreve um parâmetro de um método
<code>@throws</code>	Especifica as exceções disparadas por este método, um <code>@throws</code> deve ser fornecido para cada exceção disparada pelo método
<code>@return</code>	Especifica uma nota Returns na documentação em HTML

Por exemplo:

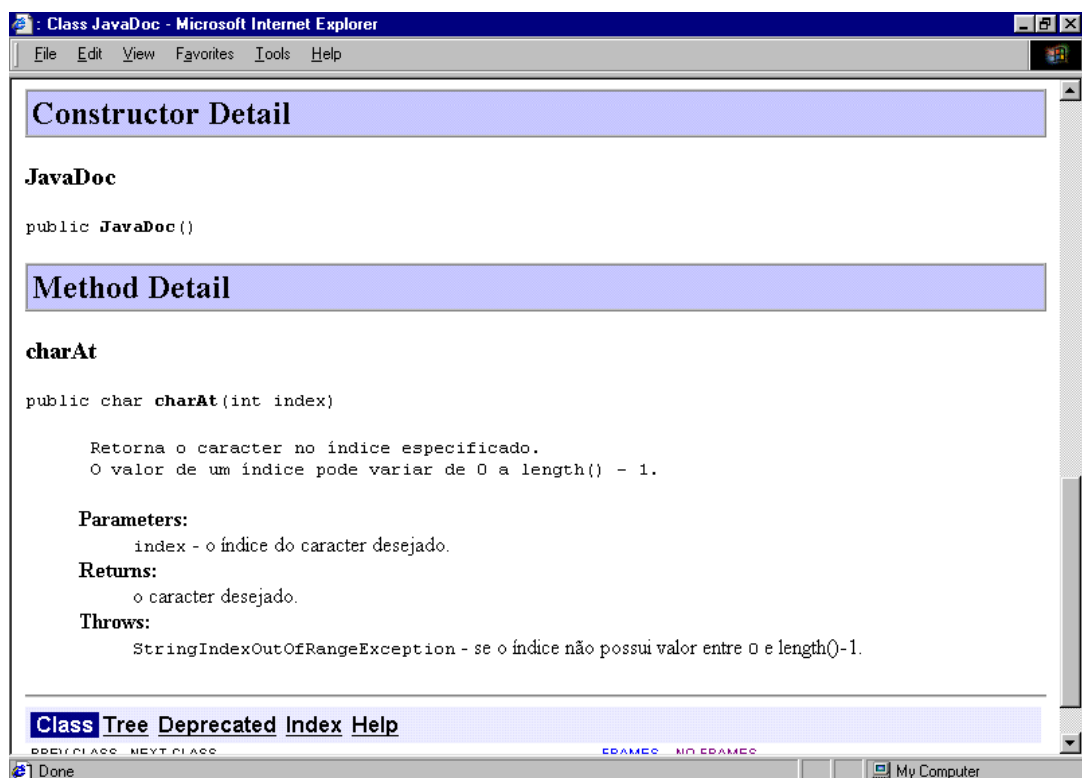
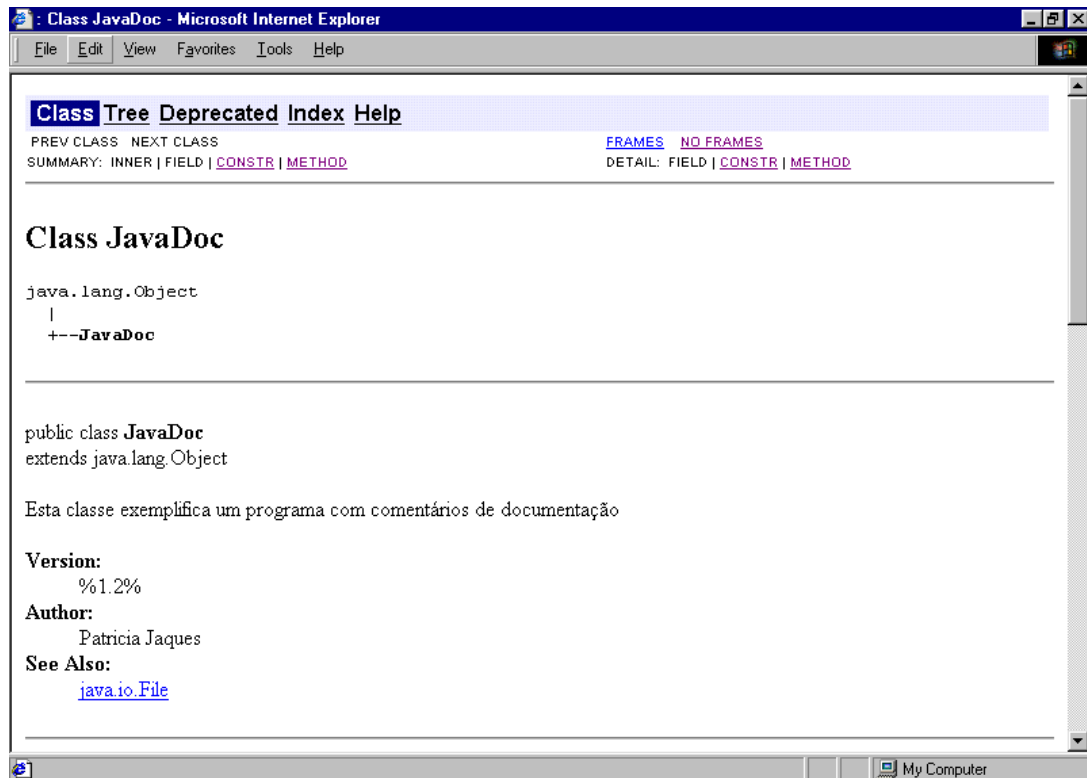
```
import java.lang.*;
/** Esta classe exemplifica um programa com comentários de documentação
 * @see <a href="C:\jdk1.3\docs\api\java\io\File.html">java.io.File</a>
 * @author Patricia Jaques
 * @version %1.2%
 */
public class JavaDoc {
    /** <pre> Retorna o caractere no índice especificado.
     * O valor do índice pode variar de <code>0</code> a <code>length() - 1</code>.
     </pre>
     * @param index o índice do caractere desejado.
     * @return o caractere desejado.
     * @exception StringIndexOutOfBoundsException
     * se o índice não possui valor entre <code>0</code> e <code>length()-1</code>.
     */

    public char charAt(int index) { • • • }
}
```

Para documentar um código fonte Java, use o comando:

```
javadoc -author -version JavaDoc.java
```

Nas figuras abaixo, podemos ver uma das páginas HTML de documentação (visão Class) gerada pela ferramenta JavaDoc.

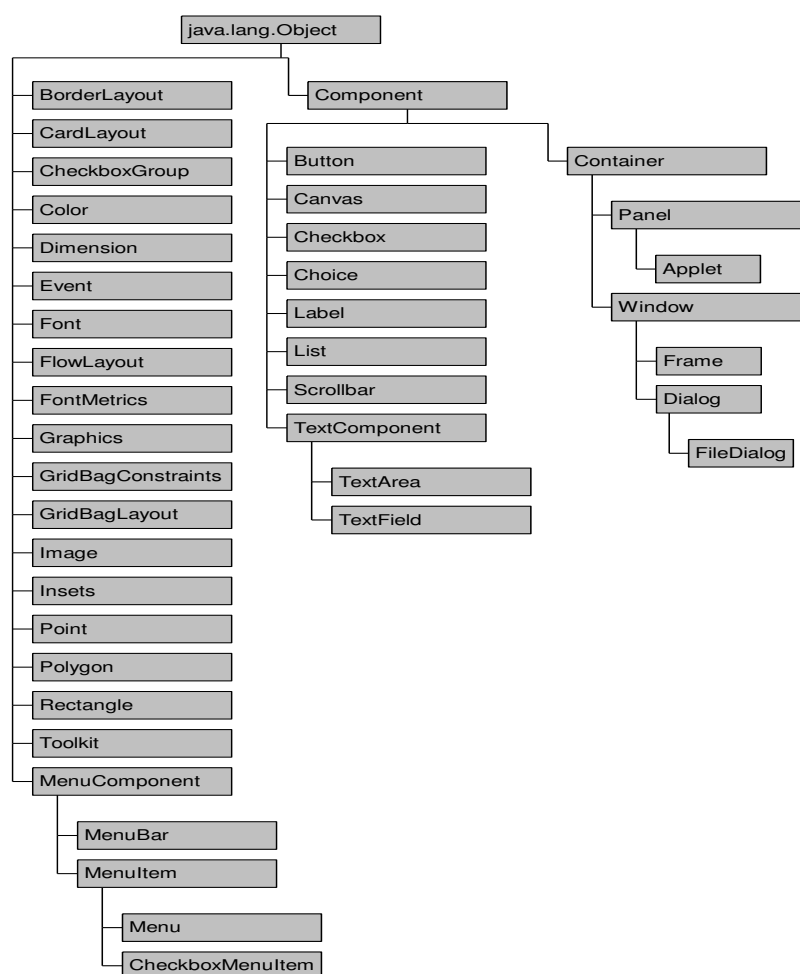


## 8 Interfaces em Java – A biblioteca AWT

Neste capítulo será introduzida a biblioteca AWT (Abstract Window Toolkit) que permite criar programas Java com interfaces gráficas (GUI).

A biblioteca AWT é composta por vários componentes que permitem criar vários tipos de objetos em uma interface, por exemplo, botão, scrollbar, label, ect. Todos esses elementos da interface do usuário são subclasses de Component, uma classe abstrata. Neste capítulo visualizaremos os componentes mais usados para criação de interfaces gráficas.

O diagrama da hierarquia de classes do pacote de interface AWT é demonstrado na Figura 4.

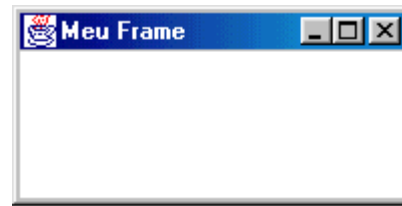


**Figura 4: Diagrama das classes do pacote de interface AWT.**

### 8.1 Frame

Um frame é normalmente visto como uma janela, com opções de minimizar, maximizar e fechar. Em uma aplicação stand-alone, os objetos de interface gráfica devem estar em um Frame para que possam ser visualizados. Um applet pode criar um Frame, porém este aparecerá com uma faixa "Untrusted Java Applet Window". Para aplicações stand-alone, essa mensagem não aparece. Veja o exemplo abaixo:

```
import java.awt.*;
public class FrameDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        f.setSize (200, 200);
        f.setVisible (true);
    }
}
```



## 8.2 Gerenciadores de Layout

O Layout dos componentes em um container (Frame ou Panel) pode ser governado por um gerenciador de layout. Este layout irá determinar as posições dos componentes (botões, label, área de texto, etc) em um Frame ou Panel. Os layouts default de Panels e Applets é o FlowLayout e de Frames é o BorderLayout. É possível que o programador determine um outro layout instanciando um novo objeto gerenciador de layouts.

Nesta apostila iremos ver os seguintes layouts:

- **FlowLayout:** Layout default para *Panels* e *Applets*;
- **BorderLayout:** Layout default para *Frames* e *Dialogs*;
- GridLayout;
- CardLayout;
- GridBagLayout;

### 8.2.1 FlowLayout

O gerenciador FlowLayout posiciona os objetos linha por linha, centralizados na janela. Quando uma linha é preenchida, ele inicia uma nova linha. O FlowLayout possui três métodos construtores:

```
FlowLayout ( );
FlowLayout (int align);
FlowLayout (int align, int hgap, int vgap);
```

O parâmetro *align* permite especificar um novo alinhamento para os objetos. Os valores possíveis para este parâmetro são:

- FlowLayout.LEFT: Alinhamento à esquerda;
- FlowLayout.CENTER: Alinhamento centralizado;
- FlowLayout.RIGHT: Alinhamento à direita;

Além disso, os parâmetros *hgap* e *vgap* permitem especificar o espaçamento horizontal e vertical entre os objetos em pixel.

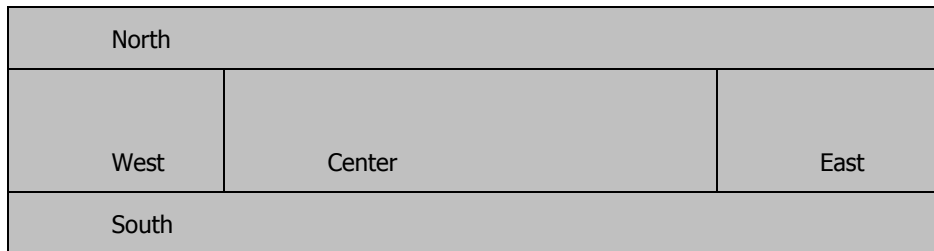
O código abaixo insere quatro botões em um Frame com layout FlowLayout.

```
import java.awt.*;
class GerenciadorFlowLayout {
    public static void main (String args[]) {
        Frame f = new Frame ();
        FlowLayout layout = new FlowLayout (FlowLayout.LEFT);
        f.setLayout (layout);
        Button b1 = new Button ("um");
        Button b2 = new Button ("dois");
        Button b3 = new Button ("três");
        Button b4 = new Button ("quatro");
        f.add (b1);
        f.add (b2);
        f.add (b3);
        f.add (b4);
        f.pack();
        f.setVisible (true);
    }
}
```



## 8.2.2 BorderLayout

O gerenciador BorderLayout divide o Container em cinco áreas e adiciona os objetos de forma a ocupar inteiramente cada uma dessas áreas.



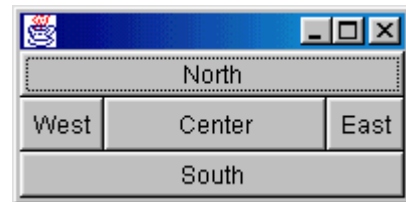
Java organiza as áreas "North", "South", "West" e "East" em função do tamanho dos seus componentes. O objeto adicionado na área "Center" terá o tamanho da área restante. No BorderLayout quando o Frame é redimensionado, os objetos também são redimensionados. Isto não acontece no FlowLayout.

Para adicionar componentes ao objeto, use o método:

```
add (Component, String área);
```

O programa a seguir cria um Frame com cinco botões e com gerenciador BorderLayout.

```
import java.awt.*;
class GerenciadorBorderLayout {
    public static void main (String args[]) {
        Frame f = new Frame ();
        BorderLayout layout = new BorderLayout ();
        f.setLayout (layout);
        Button b1 = new Button ("North");
        Button b2 = new Button ("Center");
        Button b3 = new Button ("South");
        Button b4 = new Button ("West");
        Button b5 = new Button ("East");
        f.add (b1, BorderLayout.NORTH);
        f.add (b2, BorderLayout.CENTER);
        f.add (b3, BorderLayout.SOUTH);
        f.add (b4, BorderLayout.WEST);
        f.add (b5, BorderLayout.EAST);
        f.setSize (200, 100);
        f.setVisible (true);
    }
}
```

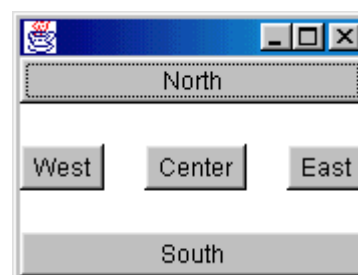


Além disso, é possível adicionar espaços entre os componentes. Para tanto é necessário chamar o seguinte método construtor do BorderLayout:

```
BorderLayout (int hgap, int vgap);
```

Onde *hgap* e *vgap* determinam o espaçamento horizontal e vertical, em pixels, entre os componentes. Por exemplo, para colocar espaço entre os componentes no código acima é necessário modificar a linha oito (8) do código por:

```
BorderLayout layout = new BorderLayout (20, 20);
```



### 8.2.3 GridLayout

O GridLayout organiza os componentes em uma grade imaginária. Pode ser usado um dos dois métodos construtores:

```
GridLayout (int rows, int cols);
GridLayout (int rows, int cols, int hgap, int vgap);
```

Os parâmetros *rows* e *cols* especificam o número de linhas e colunas da grade. Os parâmetros *hgap* e *vgap* determinam o espaçamento horizontal e vertical entre os componentes.

O programa abaixo cria um Frame com cinco botões e layout BorderLayout.

```
import java.awt.*;

class GerenciadorGridLayout {
    public static void main (String args[]) {
        Frame f = new Frame ();
        GridLayout layout = new GridLayout (2, 2);
        // GridLayout layout = new GridLayout (2, 2, 10, 10);  -> com
        espaçamento

        f.setLayout (layout);
        Button b1 = new Button ("um");
        Button b2 = new Button ("dois");
        Button b3 = new Button ("três");
        Button b4 = new Button ("quatro");
        Button b5 = new Button ("cinco");
        f.add (b1);
        f.add (b2);
        f.add (b3);
        f.add (b4);
        f.add (b5);
        f.pack();
        f.setVisible (true);
    }
}
```



### 8.2.4 CardLayout

O CardLayout permite alternar entre várias janelas, onde é exibida uma única janela de cada vez. Para criar um objeto CardLayout pode ser usado um dos seguintes métodos construtores:

```
CardLayout ();
CardLayout (int hgap, int vgap);
```

No CardLayout *hgap* e *vgap* especificam a distância horizontal e vertical entre o componente exibido e as bordas do Frame ou Panel.

Para adicionar componentes, isto é, páginas ao CardLayout, é usado o método *add* da classe *Container*:

```
add (String nomeDaPagina, Component comp);
```

Para visualizar uma nova página, você deve usar o método *show* da classe *CardLayout*:

```
show (Container container, String nome);
```

Onde *frame* é o container (Frame, Panel ou Applet) e *nome* é o nome da página a ser exibida. Para entender melhor o funcionamento do CardLayout, veja o exemplo a seguir:

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

```

import java.io.*;
class GerenciadorCardLayoutSimples {
    public static void main (String args[]) {
        Frame f = new Frame ();
        CardLayout layout = new CardLayout ();
        Panel p1 = new Panel ();
        p1.setBackground (Color.yellow);
        Panel p2 = new Panel ();
        p2.setBackground (Color.blue);
        Panel p3 = new Panel ();
        p3.setBackground (Color.green);
        f.setLayout (layout);
        f.add (p1, "Um");
        f.add (p2, "Dois");
        f.add (p3, "Três");
        f.setSize (300, 100);
        f.setVisible(true);
        while (true) {
            String str = Console.read ("Digite o nome do Panel que você quer
ver: ");

            int i = Integer.parseInt (str);
            if (i==1) layout.show (f, "Um");
            else if (i==2) layout.show (f, "Dois");
            else if (i==3) layout.show (f, "Três");
            else if (i==4) layout.previous (f); // mostra página anterior
                else if (i==5) layout.next (f); // mostra próxima página
            else System.exit (0); // finaliza execução
        }
    }
}

class Console {
    public static String read (String str) {
        InputStream in = System.in;
        InputStreamReader is = new InputStreamReader (in);
        BufferedReader console = new BufferedReader (is);
        System.out.print (str);
        String ret = null;
        try {
            ret = console.readLine ();
        }
        catch (IOException e) {
            ret = "<" + e + ">";
        }
        return ret;
    }
} // da class

```

### 8.2.5 GridBagLayout

O GridBagLayout permite que interfaces com layouts mais complexos sejam construídas. O GridBagLayout, assim como o GridLayout, divide a interface em uma grade, porém permite que um componente ocupe o tamanho desejado na célula, ao invés de ocupar a célula inteira. O GridBagLayout também permite que um único componente ocupe mais de uma célula.

Para utilizar o GridBagLayout você deve primeiramente criar um objeto GridBagLayout através do construtor e defini-lo como gerenciador de layout:

```

GridBagLayout gbLayout = new GridBagLayout ();
setLayout (gbLayout);

```

Após, você deve criar um objeto GridBagConstraints que contém vários campos que são utilizados para especificar a geometria de cada componente e a relação com os outros componentes no GridBagLayout.



Tendo feito isso, você deve definir os campos `GridBagConstraints` para cada componente no `GridLayout` e aplica-los ao layout utilizando o método `setConstraints` do `GridLayout`: Após isso, o objeto será adicionado ao container com o método `add`.

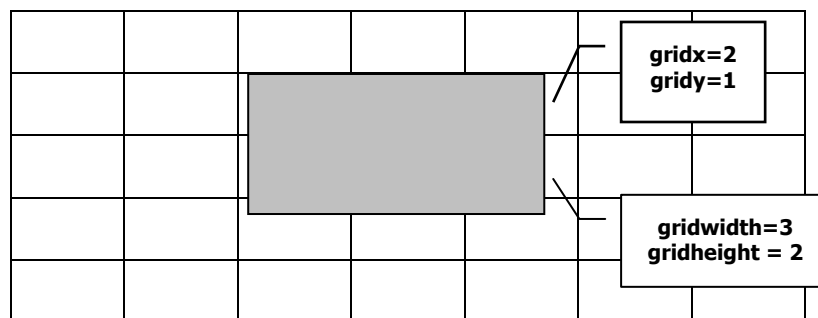
```
Frame f = new Frame ();
GridLayout gbLayout = new GridLayout ();
f.setLayout (gbLayout);
GridBagConstraints constraints= new GridBagConstraints ();
Button b1 = new Button ("Botão 1");
constraints.anchor = GridBagConstraints.CENTER;
gbLayout.setConstraints (b1, constraints);
f.add (b1);
```

## 8.2.6 Compreendendo os campos do GridBagConstraints

A classe `GridBagConstraints` define a localização e o tamanho de um componente em um `GridLayout`. A seguir veremos para que servem cada um dos campos desta classe.

### a) gridx e gridy

Os campos `gridx` e `gridy` especificam a posição da linha e da coluna a partir do canto superior esquerdo. Por exemplo, `gridx=2` e `gridy=1` determinam que este componente irá ocupar uma posição na terceira coluna e na segunda linha. Isto porque a indexação começa a partir do *zero*.



Além de especificar um valor inteiro para `gridx` e `gridy`, é possível especificá-los como `GridBagConstraints.RELATIVE` (valor default), o que faz com que o gerenciador de layout os coloque a direita (para `gridx`) ou abaixo do objeto adicionado anteriormente (`gridy`).

### b) gridwidth e gridheight

Os campos `gridwidth` e `gridheight` especificam o número de células que o componente irá ocupar na linha e na coluna. O valor default é 1. Use `GridConstraints.REMAINDER` para determinar que este é o último componente na linha (`gridwidth`) ou na coluna (`gridheight`). Use `GridConstraints.RELATIVE` para informar que este componente é o próximo após a linha (`gridwidth`) ou coluna (`gridheight`) do último componente colocado.

### c) weightx e weighty

Os campos `weightx` e `weighty` determinam como o gerenciador de layout distribui os espaços adicionais entre os componentes. Um valor maior significa que o componente obtém uma participação no espaço. O valor zero (valor default) determina que o componente não obtém espaço adicional. O espaço restante é colocado entre a grade e as bordas do container. Os valores podem variar entre 0 e 100.

### d) fill

O campo `fill` redimensiona os componentes para preencher a área de exibição do componente. Pode ter um dos seguintes valores:

Valor	Efeito
GridBagConstraints.NONE	Não redimensiona para preencher. Este é o valor default.
GridBagConstraints.HORIZONTAL	Estica para preencher horizontalmente.
GridBagConstraints.VERTICAL	Estica para preencher verticalmente.
GridBagConstraints.BOTH	Estica para preencher horizontal e verticalmente.

### e) anchor

O campo *anchor* especifica a posição do objeto na área de exibição. Os valores podem ser:

Valor	Efeito
GridBagConstraints.CENTER	Centro
GridBagConstraints.SOUTHEAST	Em cima e à esquerda
GridBagConstraints.NORTHWEST	Em baixo e à direita
GridBagConstraints.EAST	À esquerda
GridBagConstraints.WEST	À direita
GridBagConstraints.NORTHEAST	Em cima e à esquerda
GridBagConstraints.SOUTHWEST	Em baixo e a direita
GridBagConstraints.NORTH	Em cima
GridBagConstraints.SOUTH	Ao sul

### f) Insets

O campo *insets* adiciona um espaçamento externo ao objeto. Ele não aumenta o tamanho do componente, mas apenas aumenta o tamanho da área de exibição (células) onde ele está contido. Podem ser especificados espaçamentos superior, inferior, esquerdo e direito.

```
GridBagConstraints constraints = new GridBagConstraints ();
constraints.insets.top = 10;           // superior
constraints.insets.bottom=20;         // inferior
constraints.insets.left=5;            // à esquerda
constraints.insets.right=40;          // à direita
```

```
import java.awt.*;
class GridBagLayout {
    public static void main (String args []) {
        Frame f = new Frame ();
        GridBagLayout layout= new GridBagLayout ();
        f.setLayout (layout);
        Button b1 = new Button ("Botão 1");
        Button b2 = new Button ("Botão 2");
        Button b3 = new Button ("Botão 3");
        Button b4 = new Button ("Botão ");
        Button b5 = new Button ("Botão 5");
        Button b6 = new Button ("Botão 6");
        Button b7 = new Button ("Botão 7");
        Button b8 = new Button ("Botão 8");
        Button b9 = new Button ("Botão 9");
        Button b10 = new Button ("Botão 10");
```



```
GridBagConstraints constraints = new GridBagConstraints ();
constraints.weightx = 100;
constraints.fill = GridBagConstraints.BOTH;
layout.setConstraints (b1, constraints);
layout.setConstraints (b2, constraints);
layout.setConstraints (b3, constraints);

constraints.gridwidth = GridBagConstraints.REMAINDER; // fim da linha
layout.setConstraints (b4, constraints);
layout.setConstraints (b5, constraints);
```

```

        constraints.gridwidth = GridBagConstraints.RELATIVE; /* em baixo do
último componente adicionado */
        layout.setConstraints (b6, constraints);

        constraints.gridwidth = GridBagConstraints.REMAINDER; // fim da linha
        layout.setConstraints (b7, constraints);

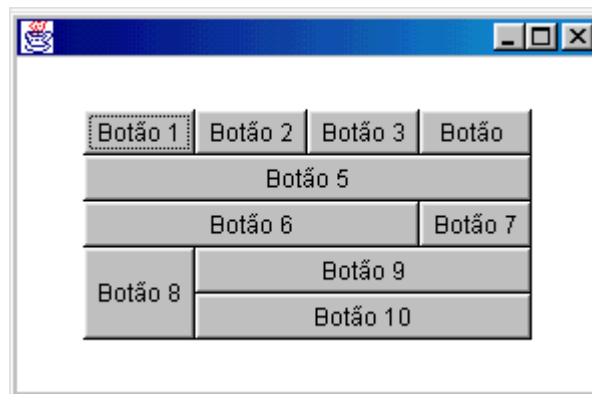
        constraints.gridwidth = 100;
        constraints.gridheight = 2; // ocupar duas linhas
        constraints.weighty = 100;
        layout.setConstraints (b8, constraints);
        constraints.gridwidth = GridBagConstraints.REMAINDER; // fim da linha
        constraints.gridheight = 1;
        constraints.weighty = 100;
        layout.setConstraints (b9, constraints);
        layout.setConstraints (b10, constraints);

        f.add (b1);
        f.add (b2);
        f.add (b3);
        f.add (b4);
        f.add (b5);
        f.add (b6);
        f.add (b7);
        f.add (b8);
        f.add (b9);
        f.add (b10);

        f.pack ();
        f.setVisible (true);
    }
}

```

Os campos `weightx` e `weighty` são usados para determinar quais componentes teriam seus tamanhos alterados em altura e largura quando o frame fosse redimensionado. Se deixássemos em comentário estas linhas, quando o frame fosse redimensionado, o tamanho dos componentes não seria alterado e o espaço resultante seria colocado entre a grade e as bordas do Frame. Veja como ficaria na figura abaixo:



### 8.3 Dispondo Componentes sem Usar um Gerenciador de Layout

Muitas vezes desejamos colocar os componentes em uma determinada posição sem usar um gerenciador de layout. Muitos ambientes de programação Java se utilizam disso. Para determinar que não será usado nenhum gerenciador de layout use:

```

Frame f = new Frame ();
f.setLayout (null);

```

O método `setBounds` é usado para determinar o tamanho e localização do objeto:

```
setBounds (int x, int y, int width, int height);
```

Onde:

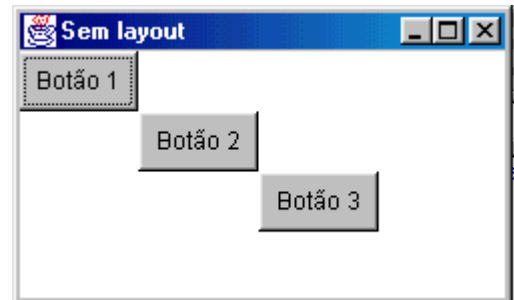
- x: A coordenada x deste componente em pixels.
- y: A coordenada y deste componente em pixels.
- width: A largura deste componente em pixels.
- height: A altura deste componente em pixels.

```
import java.awt.*;

class SemGerenciadorLayout {
    public static void main (String args []) {
        Frame f = new Frame ("Sem layout");
        f.setLayout (null);

        Button b1 = new Button ("Botão 1");
        Button b2 = new Button ("Botão 2");
        Button b3 = new Button ("Botão 3");

        b1.setBounds (3,23, 60, 30);
        b2.setBounds (63, 53, 60, 30);
        b3.setBounds (123, 83, 60, 30);
        f.add (b1);
        f.add (b2);
        f.add (b3);
        f.setSize (250, 150);
        f.setVisible (true);
    }
}
```

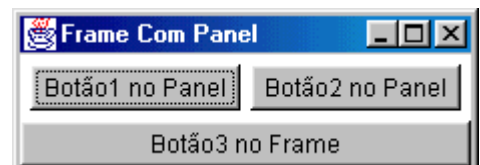


## 8.4 Panel

Muitas vezes, é necessário separar o frame em áreas para adicionar outros objetos, onde, em cada uma dessas áreas, estariam um conjunto de objetos gráficos (botões, label, etc). Podemos fazer isso com um Panel. O Panel, como os Frames, permite que outros objetos sejam adicionados nele. Em uma aplicação stand-alone, um Panel, ou vários Panels, devem estar contidos em um Frame.

```
import java.awt.*;

public class FrameComPanel {
    public static void main (String args []) {
        Frame frame = new Frame ("Frame Com Panel");
        Panel panel = new Panel ();
        Button botao1 = new Button ("Botão1 no Panel");
        Button botao2 = new Button ("Botão2 no Panel");
        Button botao3 = new Button ("Botão3 no Frame");
        frame.setLayout (new BorderLayout ());
        panel.setLayout (new FlowLayout());
        panel.add (botao1);
        panel.add (botao2);
        frame.add (panel, BorderLayout.CENTER);
        frame.add (botao3, BorderLayout.SOUTH);
        frame.pack();
        frame.setVisible (true);
    }
}
```



## 8.5 O Modelo de Delegação de Eventos

Nos programas vistos até agora não era possível fechar a janela do Frame ou quando você clicava nos botões não acontecia nada. Isso porque os nossos programas não tratavam os eventos que podiam ocorrer, tal como, clique do mouse em cima de um botão, digitar uma tecla, etc.

Eventos são objetos que descrevem o que aconteceu. Existem classes diferentes de eventos para tratar categorias diferentes de ações do usuário. Os eventos irão gerar um objeto que conterá o resultado de alguma ação do usuário.

Para tratar os eventos ocorridos em um objeto é necessário que haja um tratador de eventos associado a ele. Assim quando for realizado um evento sobre um objeto, a JVM chamará esse tratador de eventos passando as informações necessárias a ele.

Por exemplo, vamos associar um tratador de eventos ao Frame para que possamos finalizar a execução do programa quando o usuário apertar o botão de fechar do frame.

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

public class FrameComEvento {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        WindowHandler handler = new WindowHandler ();
        f.addWindowListener (handler);
        f.setSize (200, 100);
        f.show ();
    }
}

class WindowHandler implements WindowListener {
    public void windowClosing (WindowEvent e) {
        System.exit (0);
    }
    public void windowOpened (WindowEvent e) { }
    public void windowIconified (WindowEvent e) { }
    public void windowDeiconified (WindowEvent e) { }
    public void windowClosed (WindowEvent e) { }
    public void windowActivated (WindowEvent e) { }
    public void windowDeactivated (WindowEvent e) { }
}
```

Observe que como a classe tratadora do evento tem que implementar um Listener que é uma interface, é necessário que todos os métodos sejam declarados, mesmo que não usados.

Por exemplo, a interface WindowListener é responsável por tratar os eventos da janela do Frame. Cada um dos seus métodos tem uma determinada função. Para tratar o evento de fechar a janela (botão fechar do frame), usamos o método windowClosing que é chamado pela JVM quando a janela do frame a qual ele está associado está se fechando.

Para cada um dos tipos de componentes (botão, label, etc) existe um Listener que captura os eventos deste tipo de objeto. Na tabela a seguir eles são demonstrados. Na coluna Categoria está o nome dos componentes que ele captura os eventos. Na coluna nome da interface, encontra-se o nome do Listener. Na coluna Métodos, encontram-se os nomes dos métodos desta interface que devem ser implementados pela classe tratadora. Na última coluna está especificado o nome do método que permite associar um tratador a este componente.

Categoria	Interface	Métodos	Método do componente
Button Label, List, Menu, TextField	ActionListener	actionPerformed(ActionEvent)	addActionListener (ActionListener)
CheckBox CheckboxMenuItem Choice, Label, List	ItemListener	itemStateChanged(ItemEvent)	addItemListener (ItemListener)
Usado para capturar teclas digitadas.	KeyListener (Pode ser associado a qualquer objeto seja subclasse de um Component)	keyPressed(KeyEvent)	addKeyListener (KeyListener)
		keyReleased(KeyEvent)	
		keyTyped(KeyEvent)	
Trata eventos de pressionar o Mouse.	MouseListener (Pode ser associado a qualquer objeto seja subclasse de um Component)	mousePressed(MouseEvent)	addMouseListener ( <a href="#">MouseListener</a> )
		mouseReleased(MouseEvent)	
		mouseEntered(MouseEvent)	
		mouseExited(MouseEvent)	
Trata eventos de movimentação do Mouse.	MouseMotionListener (Pode ser associado a qualquer objeto seja subclasse de um Component)	mouseDragged(MouseEvent)	addMouseMotionListener ( <a href="#">MouseMotionListener</a> )
		mouseMoved(MouseEvent)	
TextArea TextField	TextListener	textValueChanged(TextEvent)	addTextListener ( <a href="#">TextListener</a> )
Frames	WindowListener	windowClosing(WindowEvent)	addWindowListener ( <a href="#">WindowListener</a> )
		windowOpened(WindowEvent)	
		windowIconified(WindowEvent)	
		windowDeiconified(WindowEvent)	
		windowClosed(WindowEvent)	
		windowActivated(WindowEvent)	
		windowDeactivated(WindowEvent)	

Cada um destes listeners será demonstrado juntamente com os componentes que serão vistos.

## 8.6 Adaptadores

Você deve ter observado que para tratar o evento de fechar a janela, nós tivemos que definir um Listener para o Frame: WindowHandler. Para captar os eventos de janela, esta classe tem que implementar a interface WindowListener.

Quando uma classe implementa uma interface, temos que implementar todos os métodos definidos nesta interface. Embora tenhamos apenas usado o método *windowClosing*, tivemos que implementar todos os outros métodos definidos na Interface WindowListener. Caso contrário, seria gerado um erro de compilação.

Para nos poupar esse trabalho, Java implementa estas classes, são os adaptadores. Desta maneira, ao invés da nossa classe tratadora de eventos implementar uma interface Listener, ela vai derivar uma classe adaptadora. Porém, como Java não permite herança múltipla, a classe tratadora pode derivar um único adaptador e implementar as demais interfaces.

O exemplo anterior ficaria:

```
import java.awt.*;
import java.awt.event.*;
public class FrameComEvento {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        WindowHandler handler = new WindowHandler ();
    }
}
```

```

        f.addWindowListener (handler);
        f.setSize (200, 100);
        f.show ();
    }
}
class WindowHandler extends WindowAdapter {
    public void windowClosing (WindowEvent e) {
        System.exit (0);
    }
}

```

Interface	Adaptador
ActionListener	Não há adaptador para esta interface, pois só possui um método.
ItemListener	Não há adaptador para esta interface, pois só possui um método.
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
TextListener	Não há adaptador para esta interface, pois só possui um método.
WindowListener	WindowAdapter

## 8.7 Label

A classe Label permite desenhar uma string em um campo não editável. Os métodos mais usados deste componente são:

- `setForeground (Color c)` – define cores da string. Algumas opções para cor são: `Color.blue`, `Color.green`, `Color.magenta`, `Color.black`.
- `setBackground (Color c)` – permite definir a cor de fundo do label.

```

import java.awt.*;
class LabelDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Label label = new Label ("MinhaLabel", Label.CENTER);
        label.setForeground (Color.blue);
        label.setBackground (Color.white);
        f.add (label);
        f.pack ();
        f.show ();
    }
}

```



## 8.8 Definindo Fontes e Cores de Texto

No exemplo, anterior você aprendeu a criar um label com a fonte padrão. Porém, é possível definir fontes diferentes para o seu texto. Para definir uma nova fonte para o seu texto, você deve criar um objeto do tipo `Font`:

```
Font minhaFonte = new Font (String name, int style, int size);
```

Onde *name* especifica o nome da fonte, por exemplo, "TimesRomam". O parâmetro *style* define o estilo da fonte como itálico e negrito. O último parâmetro é o tamanho da fonte.

Você pode usar como parâmetro de estilo do texto as seguintes opções:

Parâmetro	Estilo
Font.PLAIN	Normal
Font.BOLD	Negrito
Font.ITALIC	Itálico
Font.BOLD + Font.ITALIC	Você pode ainda usar uma combinação de dois estilos, por exemplo, negrito e itálico.

Java define cinco nomes de fontes independentes de plataforma, que você pode usar: *Courier*, *Dialog*, *Helvetica*, *Times New Roman* e *Symbol*. Quando um programa usa uma fonte, Java transforma a fonte em uma fonte dependente de plataforma. Por exemplo, a fonte Helvetica é transformada em Arial no Windows. Você pode definir uma fonte diferente destas cinco que são padrão. Se a fonte não existir no sistema operacional que sua aplicação está rodando, Java a substitui por uma outra automaticamente. Vejamos agora, como fica o nosso Label com uma fonte TimesRoman, em negrito e itálico e com tamanho 18.

```
import java.awt.*;
class FontDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Label label = new Label ("MinhaLabel", Label.CENTER);
        label.setForeground (Color.blue);
        label.setBackground (Color.white);
        Font font = new Font ("TimesRoman", Font.BOLD + Font.ITALIC, 18);
        label.setFont (font);
        f.add (label);
        f.pack ();
        f.show ();
    }
}
```



É possível obter uma lista com o nome de todos os objetos do tipo Font disponíveis. O trecho do código retorna um array composto pelos objetos Font disponíveis em Java. Veja o código abaixo:

```
import java.awt.*;
class TodasAsFontes {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        String fontes[] = f.getToolkit().getFontList ();
        int nroFont = fontes.length;
        f.setLayout (new GridLayout (nroFont, 1));
        for (int i=0; i<nroFont; i++) {
            Label label = new Label (fontes[i], Label.CENTER);
            label.setForeground (Color.blue);
            label.setBackground (Color.cyan);
            Font fonte = new Font (fontes[i], Font.PLAIN, 12);
            label.setFont (fonte);
            f.add (label);
        }
        f.pack ();
        f.show ();
    }
}
```



Você observou que definimos as cores do Label com os métodos:

- `setForeground`: define cor do texto;
- `setBackground`: define cor de fundo.

Como padrão, existem as seguintes opções de cores:

Color.black	Color.blue	Color.cyan	Color.darkGray	Color.gray
Color.green	Color.lightGray	Color.magenta	Color.orange	Color.pink
Color.red	Color.white	Color.yellow		



É possível ainda definir uma nova cor, conhecendo os seus valores RGB. Nesse caso, é necessário criar um objeto do tipo `Color`:

```
Color minhaCor = new Color (int r, int g, int b);
```

ou

```
Color minhaCor = new Color (int rgb);
```



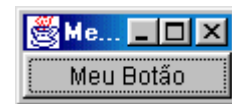
#### Entendendo a classe Toolkit

A classe `Toolkit` é uma classe abstrata que você pode usar para obter informações sobre seu sistema, tais como as fontes disponíveis, a resolução e o tamanho da tela. Para acessar o objeto `Toolkit`, o container (Frame ou applet) deve chamar a função `getDefaultToolkit()` que retorna um objeto `Toolkit`. Além da função `getFontList()` do `Toolkit`, é possível usar as funções `getScreenResolution()`, que retorna a resolução da tela e `getScreenSize`, que retorna um objeto do tipo `Dimension`, cujas variáveis membros contêm largura e a altura da tela.

## 8.9 Button

Um `Button` é um componente que pode ser usado para invocar alguma ação quando o usuário o pressiona e solta. Um `Button` é rotulado com uma `String` que é sempre centralizada no button.

```
import java.awt.*;
import java.awt.event.*;
class ButtonDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Button button = new Button ("Meu Botão");
        button.addActionListener (new TratadorDeBotao());
        f.add (button);
        f.pack();
        f.show ();
    }
}
class TratadorDeBotao implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        if (str.equals ("Meu Botão")) System.out.println ("Botão
pressionado.");
    }
}
```



## 8.10 Checkbox

`Checkbox` são objetos associados a um texto disponibilizados para o usuário fazer uma seleção do tipo "on/off" com o mouse.

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

class CheckBox {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Checkbox cb1 = new Checkbox ("Windows");
        Checkbox cb2 = new Checkbox ("Solaris");
        Checkbox cb3 = new Checkbox ("Mac");
        CheckboxHandler handler = new CheckboxHandler();
        cb1.addItemListener (handler);
        cb2.addItemListener (handler);
    }
}
```

```

        cb3.addItemListener (handler);
        f.setLayout (new FlowLayout ());
        f.add (cb1);
        f.add (cb2);
        f.add (cb3);
        f.pack ();
        f.show ();
    }
}

class CheckboxHandler implements ItemListener {
    public void itemStateChanged (ItemEvent e) {
        String nomeItem = e.getItem ().toString();
        String estado = "desabilitado";
        if (e.getStateChange ()==ItemEvent.SELECTED) estado = "habilitado";
        System.out.println (nomeItem + " foi "+estado);
    }
}

```



## 8.11 CheckboxGroup

Quando desejamos que o usuário possa escolher apenas uma das opções de CheckBox, associamos a estes um CheckboxGroup. A aparência dos objetos irá mudar para um "radio button".

Para associar um checkbox a um CheckboxGroup, você pode usar o seguinte construtor:

```
Checkbox (String label, CheckboxGroup nomeGrupo, boolean state);
```

O parâmetro *label* especifica o nome do Checkbox, o *nomeGrupo* especifica o objeto do tipo CheckboxGroup e *state* determina o estado inicial, que é *true* para ficar habilitado e *false* caso contrário.

```

import java.awt.*;
import java.awt.event.*;
class CheckBoxGroup {
    public static void main (String args []) {
        Frame f = new Frame ("CheckboxGroup");
        CheckboxGroup group = new CheckboxGroup ();
        Checkbox cb1 = new Checkbox ("Windows", group, true);
        Checkbox cb2 = new Checkbox ("Solaris", group, true);
        Checkbox cb3 = new Checkbox ("Mac", group, false);
        CheckboxHandler handler = new CheckboxHandler();
        cb1.addItemListener (handler);
        cb2.addItemListener (handler);
        cb3.addItemListener (handler);
        f.setLayout (new FlowLayout ());
        f.add (cb1);
        f.add (cb2);
        f.add (cb3);
        f.pack ();
        f.show ();
    }
}

```



## 8.12 Choice

O choice permite escolher um valor de uma lista de valores.

```

import java.awt.*;
import java.awt.event.*;
class ChoiceDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Choice");

```



```

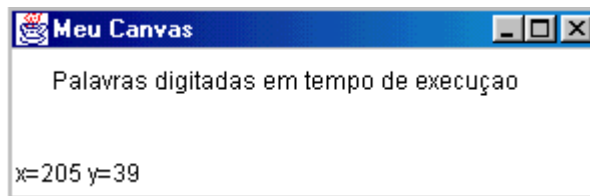
        Choice c = new Choice ();
        c.addItem ("Primeiro");
        c.addItem ("Segundo");
        c.addItem ("Terceiro");
        CheckboxHandler handler = new CheckboxHandler();
        c.addItemListener (handler);
        f.setLayout (new FlowLayout ());
        f.add (c);
        f.pack ();
        f.show ();
    }
}

```

Método	Descrição
add(String tx)	adiciona a String tx no final da lista.
insert(String tx, int pos)	adiciona a String tx na posição pos.
int getItemCount()	obtem a quantidade de itens na lista.
int getSelectedIndex()	retorna índice do elemento selecionado.
remove(int pos)	remove elemento da posição pos.
removeAll()	remove todos os elementos da lista.
select(int pos)	seleciona item na posição pos.
String getItem(int pos)	retorna o item na posição pos.
String getSelectedItem();	retorna nome do elemento selecionado na lista.

## 8.13 Canvas

Um Canvas provê um espaço colorido para desenhar, escrever texto, ou receber entrada do teclado ou do mouse.



```

import java.awt.*;
import java.awt.event.*;
class MeuCanvas {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Canvas");
        Canvas canvas = new Canvas ();
        Label label = new Label ("Label");
        f.add (canvas, BorderLayout.CENTER);
        f.add (label, BorderLayout.SOUTH);
        CanvasHandler handler = new CanvasHandler (canvas, label);
        canvas.addMouseMotionListener (handler);
        canvas.addKeyListener (handler);
        f.setSize (300, 100);
        f.setVisible (true);
    }
}
class CanvasHandler implements KeyListener, MouseMotionListener {
    Canvas canvas;
    Label label;
    String s = "";
    public CanvasHandler (Canvas canvas, Label label) {
        this.canvas = canvas;
        this.label = label;
    }
    public void keyTyped (KeyEvent e) {
        s += String.valueOf (e.getKeyChar ());
    }
}

```

```

        canvas.getGraphics().drawString (s, 20, 20);
    }
    public void keyPressed (KeyEvent e) { }
    public void keyReleased (KeyEvent e) { }
    public void mouseMoved (MouseEvent e) {
        int x = e.getX ();
        int y = e.getY ();
        String coord = "x="+x+ " y="+y;
        label.setText (coord);
    }
    public void mouseDragged (MouseEvent e) { }
}

```

## 8.14 TextField

O TextField é uma única linha editável de texto.

É possível associar um ActionListener a este objeto para informar quando o botão Enter ou Return foi pressionado. O método `e.getActionCommand()` da classe `ActionEvent` retorna a string contida no TextField.

```

import java.awt.*;
import java.awt.event.*;
class TextFieldDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        TextField tf = new TextField ("Texto Inicial");
        tf.addActionListener (new TratadorDeTextField ());
        f.add (tf);
        f.pack ();
        f.show ();
    }
}
class TratadorDeTextField implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        System.out.println (str);
    }
}

```



Para manipulação do texto contido em um TextField existem os métodos:

Método	Descrição
<code>boolean echoCharIsSet ( )</code>	retorna true se existe um caractere de echo setado. Caractere de echo é um caractere que irá aparecer no TextField no lugar do caractere digitado. Esse recurso é usado para textfield que pegam senhas.
<code>boolean isEditable( )</code>	retorna true se o componente é editável e false caso contrário.
<code>char getEchoChar( )</code>	retorna caractere de eco
<code>int getCaretPosition( )</code>	retorna posição do cursor.
<code>select (int posI, int posF)</code>	seleciona texto do TextField a partir da posição posI até a posição posF (exclusive). Primeira posição é zero.
<code>selectAll( )</code>	seleciona todo texto do TextField.
<code>set caretPosition(int)</code>	coloca o cursor na posição determinada. Posição inicial é zero.
<code>setEchoChar( )</code>	seta caractere de eco.
<code>setEditable(boolean)</code>	especifica se o componente é editável (true) ou não.
<code>setText(String)</code>	atribui uma string a um TextField.
<code>String getSelectedText( )</code>	retorna texto selecionado no TextField.
<code>String getText()</code>	retorna o texto contido em um TextField em uma String.

## 8.15 TextArea

Um `TextArea` é um campo editável de múltiplas linhas e colunas. Você pode criar um `TextArea` com o construtor:

```
TextArea (String texto, int linhas, int colunas);
```

Onde *texto* é o texto inicial que vai ser exibido no `TextArea` e *linhas e colunas* é o número de linhas e colunas da área de exibição do `TextArea`.

**O `TextArea` exibe, por default, barras de rolagem vertical e horizontal. A exibição ou não de barras de rolagem pode ser determinada somente no método construtor, ou seja, não pode ser modificada após a construção do componente. Para tanto pode ser usado o construtor:**

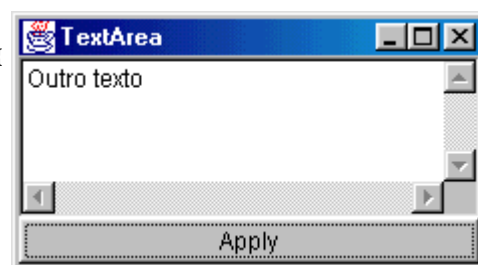
```
TextArea (String texto, int linhas, int colunas, int scrollbars);
```

Onde *scrollbars* pode ser uma das seguintes constantes:

- `TextArea.SCROLLBARS_BOTH`: com barra de rolagem horizontal e vertical.
- `TextArea.SCROLLBARS_NONE`: sem barra de rolagem.
- `TextArea.SCROLLBARS_VERTICAL_ONLY`: apenas com barra de rolagem vertical.
- `TextArea.SCROLLBARS_HORIZONTAL_ONLY`: apenas com barra de rolagem horizontal.

**No `TextArea`, pressionar o botão "Enter" gera o caractere de nova linha. Portanto, nestes casos, se você necessitar obter um texto digitado, use um botão ao invés de associar um `ActionListener`, como foi feito para o `TextField`.**

```
import java.awt.*;
import java.awt.event.*;
class TextAreaDemo {
    public static void main (String args []) {
        Frame f = new Frame ("TextArea");
        TextArea ta = new TextArea ("Texto Inicial", 4, 30);
        Button b1 = new Button ("Apply");
        b1.addActionListener (new TratadorDeTextField (ta));
        f.add (ta, BorderLayout.CENTER);
        f.add (b1, BorderLayout.SOUTH);
        f.pack ();
        f.show ();
    }
}
class TratadorDeTextField implements ActionListener {
    TextArea ta;
    public TratadorDeTextField (TextArea ta) {
        this.ta = ta;
    }
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        System.out.println (ta.getText ());
    }
}
```



Método	Descrição
<code>setText(String)</code>	atribui uma string a um <code>TextArea</code> .
<code>String getText()</code>	retorna o texto contido em um <code>TextArea</code> em uma <code>String</code> .
<code>int getCaretPosition()</code>	retorna posição do cursor.
<code>setCaretPosition(int)</code>	coloca o cursor na posição determinada. Posição inicial é zero.
<code>int getCaretPosition()</code>	retorna posição do cursor.
<code>select (int posI, int posF)</code>	seleciona texto do <code>TextArea</code> a partir da posição <code>posI</code> até a posição <code>posF</code> (exclusive). Primeira posição é zero.
<code>String getSelectedText()</code>	retorna texto selecionado no <code>TextArea</code> .
<code>selectAll()</code>	seleciona todo texto do <code>TextArea</code> .
<code>setEditable(boolean)</code>	especifica se o componente é editável ( <code>true</code> ) ou não.

<code>boolean isEditable( )</code>	retorna true se o componente é editável e false caso contrário.
<code>append(String tx)</code>	anexa texto tx ao final do texto contido no TextArea.
<code>insert(String tx, int pos)</code>	insere texto tx na posição pos.
<code>replaceRange(String tx, int posI, int posF)</code>	Substitui o texto contido entre as posições posI e posF pelo texto tx.

## 8.16 List

Uma lista permite apresentar opções textuais que são exibidas em uma região onde vários itens sejam exibidos em uma única vez. As listas apresentam barra de rolagem e permitem que mais de uma opção sejam selecionadas.

```
List l = new List (int tam, boolean múltiplas_escolhas);
```

Onde *tam* é o número de linhas visíveis da lista e *múltiplas\_escolhas* é um valor boolean que é *true* quando se deseja que mais de uma opção possam ser selecionadas na lista.

```
import java.awt.*;
import java.awt.event.*;
class ListDemo {
    public static void main (String args []) {
        Frame f = new Frame ("List");
        List list = new List (4,true);
        list.add ("Primeiro");
        list.add ("Segundo");
        list.add ("Terceiro");
        list.add ("Quarto");
        list.add ("Quinto");
        list.add ("Sexto");
        Tratador handler = new Tratador (list);
        list.addItemListener (handler);
        list.addActionListener (handler);
        f.setLayout (new FlowLayout ());
        f.add (list);
        f.pack ();
        f.show ();
    }
}
class Tratador implements ItemListener, ActionListener {
    List list;
    public Tratador (List list) {
        this.list = list;
    }
    // ActionListener pega duplo clique
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        System.out.println ("Pegou duplo clique de " + str + ".");
    }
    // ItemListener pega clique único
    public void itemStateChanged (ItemEvent e) {
        String nomeItem = e.getItem ().toString();
        String estado = "desabilitado";
        if (e.getStateChange ()==ItemEvent.SELECTED) estado = "habilitado";
        System.out.println ("Na itemStateChanged: "+nomeItem + " foi
"+estado);

        int []ind = list.getSelectedIndexes ();
        String []items = list.getSelectedItems();
        for (int i=0; i<ind.length; i++)
            System.out.println (ind[i] + " " + items[i]);
    }
}
```



O `ItemListener` capta o clique único sobre um determinado valor da lista e o `ActionListener` captura duplo clique.

Método	Descrição
<code>add(String tx, int pos)</code>	adiciona a String tx na posição pos.
<code>add(String tx)</code>	adiciona a String tx no final da lista.
<code>int[] getSelectedIndexes()</code>	retorna um array com os índices dos elementos selecionados.
<code>String[] getSelectedItems ( );</code>	retorna um array com o nome dos elementos selecionados na lista.
<code>int getItemCount()</code>	obtem a quantidade de itens na lista.
<code>select(int pos)</code>	seleciona item na posição pos.
<code>deselect(int pos)</code>	retira a seleção do item na posição pos.
<code>String getItem(int pos)</code>	retorna o item na posição pos.
<code>remove(int pos)</code>	remove elemento da posição pos.
<code>removeAll()</code>	remove todos os elementos da lista.
<code>setMultipleMode(boolean)</code>	ativa (se true) ou não a lista para permitir seleção de vários elementos.
<code>boolean isMultipleMode()</code>	retorna true se a lista está operando em modo de seleção múltipla.

## 8.17 Dialog

Um `Dialog` é uma janela similar a um `Frame` com a diferença de que é possível determiná-lo como modal. Quando definimos uma caixa de diálogo como modal, o usuário não pode passar para outras janelas no mesmo aplicativo até que feche a caixa de diálogo. Se ela não for modal é possível passar para outra janela simplesmente dando um clique nela.

É possível criar uma caixa de diálogo com um dos seguintes construtores:

```
Dialog (Frame frame, boolean modal);
Dialog (Frame frame, String titulo, Boolean modal);
```

Onde *frame* é o `Frame` ao qual esta caixa de diálogo está associada, *modal* é *true* caso a caixa seja modal e *false* caso contrário, e *título* é o título da caixa de diálogo.

```
import java.awt.*;
import java.awt.event.*;
public class DialogDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Button b = new Button ("Botão");
        f.add (b);
        Handler handler = new Handler (f);
        b.addActionListener (handler);
        f.addWindowListener (handler);
        f.setLayout (new FlowLayout ());
        f.setSize (200, 100);
        f.show ();
    }
}
class Handler implements WindowListener, ActionListener {
    Frame f;
    public Handler (Frame f) {
        this.f = f;
    }
    public void actionPerformed (ActionEvent e) {
        Dialog d = new Dialog (f, "Dialog", true);
        Label l = new Label ("Label na Caixa de Diálogo");
        d.add (l);
        d.addWindowListener (new WindowAdapter() {
            d.setVisible (false); } });
        d.pack();
        d.setVisible (true);
    }
}
```

```

    }
    public void windowClosing (WindowEvent e)
    { System.exit (0); }
    public void windowOpened (WindowEvent e) { }
    public void windowIconified (WindowEvent e) { }
    public void windowDeiconified (WindowEvent e) { }
    public void windowClosed (WindowEvent e) { }
    public void windowActivated (WindowEvent e) { }
    public void windowDeactivated (WindowEvent e) { }
}

```

Assim, como um `Frame` um `Dialog` é uma subclasse de um `Window`. O seu `Layout default` é `BorderLayout`. Para fechar um `Dialog` você pode usar o método `dispose`.

## 8.18 FileDialog

O `FileDialog` permite selecionar arquivos do sistema através de uma janela específica.

O `FileDialog` retorna apenas uma `String` contendo o nome e diretório onde o arquivo está localizado. Ele não retorna o arquivo em si.

Ao criar a caixa de diálogo, podemos determinar o tipo através do seu construtor:

```
FileDialog d = new FileDialog (Frame f, String nome, int mode);
```

Onde `f` é o frame ao qual esta caixa de diálogo está associada. Uma caixa de diálogo é sempre modal. Nome é o título da caixa de diálogo e `mode` é um dos tipos:

- `FileDialog.SAVE`: caixa de diálogo salvar;
- `FileDialog.LOAD`: caixa de diálogo de abertura de arquivo;

```

import java.awt.*;
import java.awt.event.*;
public class FileDialogDemo {
    public static void main (String args []) {
        Frame f = new Frame ("Meu Frame");
        Button b = new Button ("Botão");
        f.add (b);
        Handler handler = new Handler (f);
        b.addActionListener (handler);
        f.addWindowListener (handler);
        f.setLayout (new FlowLayout ());
        f.setSize (200, 100);
        f.show ();
    }
}
class Handler implements WindowListener, ActionListener {
    Frame f;
    public Handler (Frame f) {
        this.f = f;
    }
    public void actionPerformed (ActionEvent e) {
        FileDialog d = new FileDialog (f, "FileDialog", FileDialog.SAVE);
        d.setVisible (true);
        String nome = d.getFile ();
        String dir = d.getDirectory ();
        System.out.println ("Foi selecionado arquivo "+nome+" no diretório
"+dir);
    }
    public void windowClosing (WindowEvent e) {
        System.exit (0);
    }
    public void windowOpened (WindowEvent e) { }
}

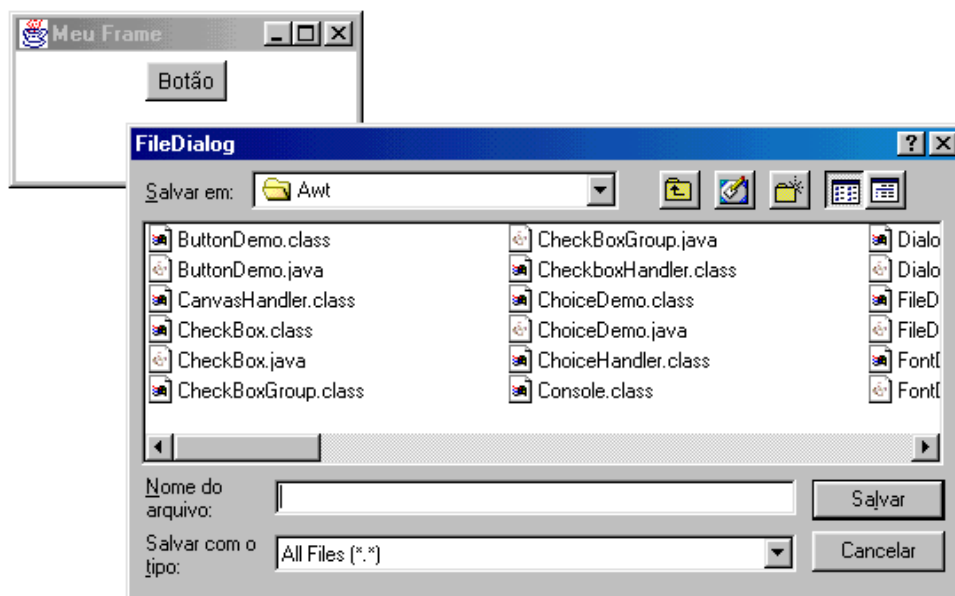
```



```

    public void windowIconified (WindowEvent e) { }
    public void windowDeiconified (WindowEvent e) { }
    public void windowClosed (WindowEvent e) { }
    public void windowActivated (WindowEvent e) { }
    public void windowDeactivated (WindowEvent e) { }
}

```



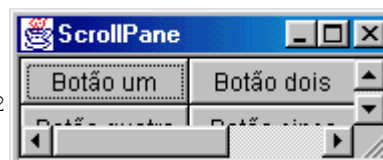
## 8.19 ScrollPane

Esta classe fornece um container com barras de rolagem. O ScrollPane não permita que seja definido um layout a ele ou que sejam adicionados objetos. Se você quiser adicionar algum objeto (por exemplo, botão), deve criar um Panel, adicionar os objetos ao Panel e colocar o Panel no ScrollPane.

```

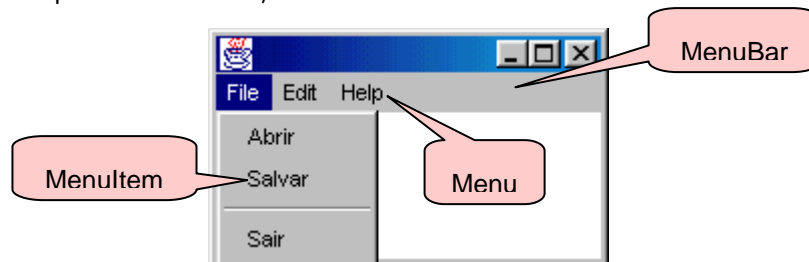
import java.awt.*;
class ScrollPaneDemo {
    public static void main (String args[]) {
        Frame f = new Frame ("ScrollPane");
        Panel p = new Panel ();
        GridLayout layout = new GridLayout (2, 2);
        p.setLayout (layout);
        Button b1 = new Button ("Botão um");
        Button b2 = new Button ("Botão dois");
        Button b3 = new Button ("Botão três");
        Button b4 = new Button ("Botão quatro");
        Button b5 = new Button ("Botão cinco");
        p.add (b1);
        p.add (b2);
        p.add (b3);
        p.add (b4);
        p.add (b5);
        ScrollPane sp = new ScrollPane ();
        sp.add (p);
        f.add (sp);
        f.setSize (100, 100);
        f.show();
    }
}

```



## 8.20 Menu

Os Menus são componentes que podem ser adicionados unicamente em Frames. Na verdade um Menu é composto por 3 componentes: MenuBar, Menu e MenuItem.



Para criar um Menu, você deve primeiramente criar uma barra de Menu, onde serão inseridos os elementos de Menu:

```
MenuBar bar = new MenuBar ();
```

Após isso, você deve definir os seus menus como objetos do tipo Menu e adiciona-los a barra de Menu:

```
Menu menuFile = new Menu ("File");
bar.add (menuFile);
```

A seguir, você pode adicionar os itens (MenuItem) de cada menu:

```
menuFile.add (item1);
menuFile.add (item2);
menuFile.addSeparator ();
menuFile.add (item3);
```

Você pode usar um separador (linha separadora) para dividir itens de menu. Para isso, basta usar o método *addSeparator* do objeto Menu.

Além disso, você pode usar o método *setHelpMenu* do objeto MenuBar para adicionar um menu *Help* a sua barra. O menu Help adicionado terá a aparência de acordo com a sua plataforma. Por exemplo, em sistemas do tipo X/Motif ele coloca o Menu Help a extrema direita da barra de Menu.

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

class MenuDemo {
    public static void main (String args []) {
        Frame frame = new Frame ();
        MenuBar bar = new MenuBar ();
        Tratador handler = new Tratador ();

        Menu menuFile = new Menu ("File");
        MenuItem item1 = new MenuItem ("Abrir");
        MenuItem item2 = new MenuItem ("Salvar");
        MenuItem item3 = new MenuItem ("Sair");
        item1.addActionListener (handler);
        item2.addActionListener (handler);
        item3.addActionListener (handler);
        menuFile.add (item1);
        menuFile.add (item2);
        menuFile.addSeparator ();
        menuFile.add (item3);
        Menu menuEdit = new Menu ("Edit");
        Menu menuHelp = new Menu ("Help");
        bar.add (menuFile);
```

```

        bar.add (menuEdit);
        bar.setHelpMenu (menuHelp);
        frame.setMenuBar (bar);
        frame.setSize (200, 100);
        frame.setVisible (true);
    }
}
class Tratador implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        System.out.println ("Foi acionado o Item de Menu: "+str);
    }
}

```

### 8.20.1 Adicionando Atalhos para os Menus

Para criar um atalho para os itens de menu devemos criar uma instância de objeto `MenuItemShortcut` que representa o atalho e passa-lo ao método construtor do `MenuItem`.

```
MenuItem item3 = new MenuItem ("Sair", new MenuItemShortcut (KeyEvent.VK_R, false));
```

A linha acima define que se forem pressionadas as teclas CTRL+R, será acionado o item de menu Sair. O primeiro argumento do método construtor de `MenuItemShortcut` é a tecla de atalho. Para Java, serão teclas de atalhos as combinações da tecla CTRL com alguma outra tecla. Na tabela a seguir são colocados alguns exemplos das constantes usadas para definir teclas de atalho. Elas estão definidas na classe `java.awt.event.KeyEvent`. O segundo argumento determina se a tecla SHIFT faz parte do atalho. Se no exemplo acima, o segundo argumento fosse true, o usuário teria que digitar CTRL+SHIFT+R para acionar esse item de menu.

Virtual Key Constante	Descrição
VK_0 .. VK_9	Teclas de 0..9
VK_A .. VK_Z	Teclas de A ..Z.
VK_F1..VK_F12	Teclas de F1 a F12
VK_ENTER	Teclas Enter

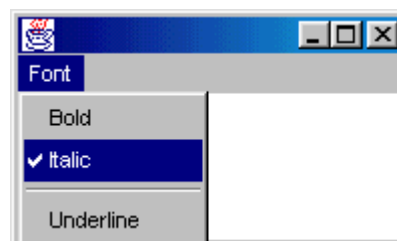
## 8.21 CheckBoxMenuItem

É possível criar itens de menu para seleção on/off, assim como no checkbox.

```

import java.awt.*;
import java.awt.event.*;
class CheckBoxMenuItemDemo {
    public static void main (String args []) {
        Frame frame = new Frame ();
        MenuBar bar = new MenuBar ();
        Handler handler = new Handler ();
        Menu menuFile = new Menu ("Font");
        CheckBoxMenuItem item1 = new CheckBoxMenuItem ("Bold");
        CheckBoxMenuItem item2 = new CheckBoxMenuItem ("Italic");
        CheckBoxMenuItem item3 = new CheckBoxMenuItem ("Underline");
        item1.addItemListener (handler);
        item2.addItemListener (handler);
        item3.addItemListener (handler);
        menuFile.add (item1);
        menuFile.add (item2);
        menuFile.addSeparator ();
        menuFile.add (item3);
        bar.add (menuFile);
        frame.setMenuBar (bar);
        frame.setSize (200, 100);
        frame.setVisible (true);
    }
}

```



```

    }
}

class Handler implements ItemListener {
    public void itemStateChanged (ItemEvent e) {
        String nomeItem = e.getItem ().toString();
        String estado = "desabilitado";
        if (e.getStateChange ()==ItemEvent.SELECTED) estado = "habilitado";
        System.out.println (nomeItem + " foi "+estado);
    }
}

```

## 8.22 Imprimindo um Frame

Java permite a impressão de componentes de interface. Quando uma operação de impressão é iniciada é aberta a caixa de diálogo da impressão, o que permite escolher opções de papel, qualidade e outros.

Para isso, é preciso criar um objeto *Graphics* que se “conecta” a impressora escolhida pelo usuário.

```

Frame frame = new Frame ("Teste de Impressão");
Toolkit toolkit = f.getToolkit();
PrintJob job = t.getPrintJob (frame, "Nome da Impressão", null);
Graphics g = job.getGraphics();

```

Após, usamos o método `printAll()` para imprimir o Frame e todos os componentes contidos nele.

```
f.printAll( );
```

Para enviar a página para a impressora, usamos o método `dispose()`:

```
g.dispose( );
```

Após isso, chamamos o método `end()` do objeto `PrintJob` para iniciar a impressão:

```
job.end( );
```

O programa abaixo desenha retângulos de cores diferentes em um Canvas e permite que o usuário imprima o Frame e os objetos desenhados.

```

import java.awt.*;
import java.lang.Math.*;
import java.awt.event.*;
import java.awt.Color.*;
import java.applet.Applet;
public class Impressao {
    public static void main (String args []) {
        Frame frame = new Frame ("Squares");
        Squares c = new Squares (frame);
        Button b1 = new Button ("Imprimir");
        b1.addActionListener (c);
        frame.add (BorderLayout.SOUTH, b1);
        frame.add (BorderLayout.CENTER, c);
        frame.setSize (300, 300);
        frame.setVisible (true);
    }
}

class Squares extends Canvas implements MouseListener, MouseMotionListener,
ActionListener {
    int xi=1, yi=1, xf=10, yf=10;
    int posicao;
    Frame f;
    public Squares (Frame f) {

```

```

        this.f = f;
        addMouseListener(this);
        addMouseMotionListener(this);
        setBackground (Color.yellow);
    }
    public void paint (Graphics g) {
        switch(posicao) {
            case 0: g.setColor(Color.blue);
                    break;
            case 1: g.setColor(Color.magenta);
                    break;
            case 2: g.setColor(Color.cyan);
                    break;
            case 3: g.setColor(Color.green);
                    break;
            default: g.setColor(Color.red);
                    break;
        }
        posicao = (posicao<5) ? posicao+1 : 0;
        g.fillRect(xi, yi, Math.abs(xf-xi), Math.abs(yf-yi));
    }
    public void update (Graphics g) {
        paint (g);
    }
    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) {
        xi = e.getX();
        yi = e.getY();
    }
    public void mouseReleased(MouseEvent e) {
        xf = e.getX();
        yf = e.getY();
        repaint();
    }
    public void mouseClicked(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void actionPerformed (ActionEvent e) {
        PrintJob pjob = f.getToolkit().getPrintJob(f, "Impressão Squares - Java",
null);
        if (pjob != null) {
            Graphics pg = pjob.getGraphics();
            if (pg != null) {
                f.printAll(pg);
                pg.dispose();
            }
            pjob.end();
        }
    }
}

```

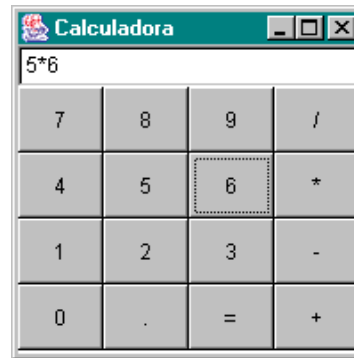
## 8.23 Exercícios

### Exercício 1 - Calculadora

Crie uma calculadora com a aparência abaixo.

Quando você clica no botão de igual(=), ela apaga o que estava no campo TextField e mostra o resultado.

**Dica:** Use o gerenciador de layout GridLayout.



## 9 Applets

Um applet é um código Java que roda em um browser. Ele se difere de uma aplicação *stand-alone* na maneira que é executado. Uma aplicação é inicializada quando sua função `main ()` é chamada. O ciclo de vida de um applet é um pouco mais complexo e será explicado neste capítulo.

Quando um usuário acessa uma página HTML que contém um applet, o código do applet é todo copiado para a máquina do usuário para, após isso, ser executado localmente.

### 9.1 Restrições de Segurança de um Applet

Como os applets são programas que podem ser carregados de páginas na Internet para ser executado localmente, é necessário proteger o usuário de programas maliciosos que tentam, por exemplo, ler um número de cartão de crédito do usuário ou uma senha para enviá-la pela Internet. Uma maneira de Java prevenir esses acessos ilegais é fornecendo um gerenciador de segurança (classe `SecurityManager`). Isso é possível porque o código Java é interpretado pela Máquina Virtual Java e não diretamente executado pela CPU. Este modelo é chamado de modelo de segurança "sandbox". Quando uma das restrições de segurança do Java é violada, o gerenciador de segurança do applet gera uma exceção do tipo *SecurityException*.

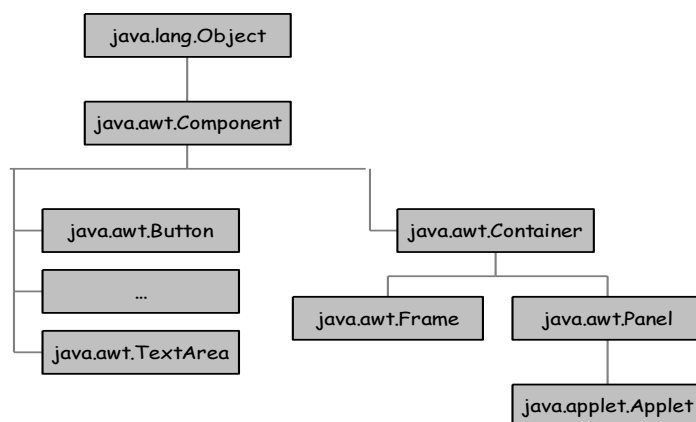
O nível de controle de segurança é implementado no browser. A maioria dos browsers (inclusive Netscape) previne as seguintes ações:

- Executar um outro programa;
- Leitura ou escrita de arquivos na máquina local;
- Chamadas a métodos nativos;
- Abrir um socket a qualquer máquina que não a host que forneceu o applet.

Dialogs e Frames quando criados através de um applet, exibirão em seus respectivos frames uma mensagem ao rodapé do mesmo: "*UNTRUSTED JAVA APPLET*" ou "*UNAUTHENTICATED JAVA APPLET*". Com isso, pode advertir o usuário de programas ilegais que tentam ler informações confidenciais.

### 9.2 Hierarquia de Java

A classe `Applet` pertence ao pacote *Java.applet* e é uma subclasse de um `Container`. Abaixo, encontra-se a hierarquia da classe `Applet`.



Como um `Applet` é um `Panel`, o seu gerenciador de layout default é o `FlowLayout`. Além disso, assim como no `Panel`, é possível inserir componentes nele.

## 9.3 Carregando um Applet

Como um applet é executado em um browser, ele não é inicializado diretamente por uma linha de comando. Para executar um applet deve-se criar um arquivo HTML especificando o applet que deve ser carregado.

Existe uma tag *applet* para especificar chamada de um applet:

```
<applet
code="appletFile.class"
width=largura em pixels da janela do applet
height=altura em pixels da janela do applet
[codebase=URL]
[archive=listaDeArquivos]
[alt=textoAlternativo]
[align=alinhamento]
[vspace=pixels] [hspace=pixels]
>
[<param name=atributo1 value=valor1>]
[<param name=atributo2 value=valor2>]
...
[TextoHTML alternativo]
</applet>
```

Onde:

- **code:** Nome do arquivo .class que é subclasse da classe Applet.
- **codebase:** URL do diretório que contém o applet. Este atributo não precisa ser especificado se a URL do applet é a mesma do documento HTML.
- **archive:** Contém a lista de classes (.class) e outros recursos (por ex., figura, som) separados por “;”. Ao invés disso, é possível conter o nome do arquivo compactado que contém todos os fontes necessários. A vantagem do arquivo compactado é que ao invés de ele criar um socket para cada arquivo, ele traz todos em único socket. Java suporta os tipos compactados “.zip” e “.jar” (gerado pelo JDK). Para compactar um arquivo “.jar” use o comando: “jar cvf minhasClasses.jar \* .class”
- **width e height:** especifica a altura e largura da área de exibição do applet. Mesmo que seja definido outro tamanho o método setSize, este é o usado para exibição do applet no browser.
- **alt:** Este texto é exibido se o browser entende a tag *applet*, mas não pode executar applets.
- **name:** Nome do applet. Usado para que o applet possa comunicar com outros applets.
- **align:** Este atributo especifica o alinhamento do applet na página HTML. Pode ser: left, right, top, bottom, texttop, middle, absmiddle, baseline e absbottom.
- **vspace e hspace:** Estes parâmetros especificam o número de pixels acima e abaixo do applet (vspace) e em cada lado do applet (hspace).
- **<param name = value= >:** Especifica parâmetros que serão lidos no applet pelo método *getParameter()* no método *init()* do applet.

## 9.4 O Ciclo de Vida de um Applet

Em uma aplicação stand-alone, a JVM começa a execução pelo método *main()*. Em um applet, porém, a execução é iniciada pelo método construtor e depois passa para alguns métodos definidos. São eles:

- **init():** Após a execução do construtor, o browser chama o método *init()*. Este método é chamado apenas na primeira vez que o applet é iniciado. Geralmente, este método é usado para ler parâmetros da página HTML (*getParameter()*), adicionar componentes de interface (GUI) e definir um novo gerenciador de Layout.
- **start():** Após a execução do método *init()* é chamado o método *start()*. Este método também é chamado toda vez que o applet torna-se visível.
- **stop():** Este método é chamado toda vez que o applet deixar de ser visível, por exemplo, quando é minimizado ou quando outra página é chamada. O método *stop()* também é chamado imediatamente antes do método *destroy()* descrito a seguir.
- **destroy():** Este método é chamado quando o applet deverá terminar a sua execução, liberando todos os recursos alocados, por exemplo, quando o browser finaliza a sua execução.



- **paint (Graphics g):** Este método é executado toda vez que o browser enviar uma mensagem de pintura para o applet. Este método é chamado na inicialização do applet e toda vez que o browser for ativado.
- **repaint():** Este método é chamado pelo programador quando ele deseja repintar a tela. Este método chama o método update ().
- **update(Graphics g):** Limpa a área de fundo e após chama o método paint (Graphics g);

## 9.5 O primeiro exemplo de um applet

O nosso primeiro applet desenha figuras geométricas com cores diferentes nas posições clicadas pelo usuário. Note que o método update( ) foi sobrescrito para que não apagasse a tela antes de chamar o método paint( ).

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class MeuApplet1 extends Applet implements MouseListener {
    int mouseX=25;
    int mouseY=25;
    int cont=0;
    public void init () {
        addMouseListener (this);
    }
    public void paint (Graphics g) {
        switch (cont) {
            case 1: g.setColor (Color.yellow);
                    g.drawRect (mouseX, mouseY, 40, 30);
                    break;
            case 2: g.setColor (Color.red);
                    g.drawOval (mouseX, mouseY, 40, 30);
                    break;
            case 3: g.setColor (Color.green);
                    g.drawLine (0, 0, mouseX, mouseY);
                    break;
            case 4: g.setColor (Color.cyan);
                    g.fillOval (mouseX, mouseY, 40, 30);
                    break;
            case 5: g.setColor (Color.gray);
                    g.drawRoundRect (mouseX, mouseY, 40, 30, 8, 8);
                    break;
            case 6: g.setColor (Color.magenta);
                    g.fillRect (mouseX, mouseY, 40, 30);
                    break;
            case 7: g.setColor (Color.black);
                    g.drawArc (mouseX, mouseY, 40, 30, 120, 300);
                    break;
            case 8: g.setColor (Color.orange);
                    g.fillArc (mouseX, mouseY, 40, 30, 120, 300);
                    break;
            case 9: g.setColor (Color.blue);
                    g.drawString ("HelloWorld!", mouseX, mouseY);
                    cont=0;
        }
    }
    public void mousePressed (MouseEvent e) {
        mouseX = e.getX();
        mouseY = e.getY();
        repaint ();
        cont++;
    }
    public void update (Graphics g) {
        paint (g);
    }
    public void mouseClicked (MouseEvent e) { }
```

```
public void mouseEntered (MouseEvent e) { }  
public void mouseExited (MouseEvent e) { }  
public void mouseReleased (MouseEvent e) { }  
}
```

Mas, para que possamos executar um applet no browser é preciso escrever uma página HTML que chame esse applet.

```
<HTML>  
  <HEAD>  
    <TITLE> Testando um Applet </TITLE>  
  </HEAD>  
  <BODY>  
    <APPLET CODE=MeuApplet.class WIDTH = 300 HEIGHT = 300 >  
  </APPLET>  
  </BODY>  
</HTML>
```

Tendo criado a página HTML, podemos abri-la em um browser para ver o applet rodando.

## 9.6 Criando programas Java aplicativos e applets

Muitas vezes, queremos que nossa aplicação execute tanto como um applet, em um browser, ou como aplicação stand-alone. Podemos fazer isso, colocando um método main() dentro do código do applet. Como um applet, é subclasse de um Panel é necessário criar um Frame para inserir o applet.

O código acima poderia ser escrito da seguinte maneira para executar como aplicação:

```
import java.awt.*;  
import java.awt.event.*;  
import java.applet.Applet;  
public class MeuApplet extends Applet implements MouseListener, WindowListener {  
    int mouseX=25;  
    int mouseY=25;  
    int cont=0;  
    public static void main (String args []) {  
        Frame f = new Frame ("Aplicação");  
        MeuApplet applet = new MeuApplet ();  
        f.add (applet);  
        f.addWindowListener (applet);  
        f.setSize (400, 400);  
        f.setVisible (true);  
        applet.init();  
        applet.start();  
    }  
    public void init () {  
        addMouseListener (this);  
    }  
    public void paint (Graphics g) {  
        switch (cont) {  
            case 1: g.setColor (Color.yellow);  
                    g.drawRect (mouseX, mouseY, 40, 30);  
                    break;  
            case 2: g.setColor (Color.red);  
                    g.drawOval (mouseX, mouseY, 40, 30);  
                    break;  
            case 3: g.setColor (Color.green);  
                    g.drawLine (0, 0, mouseX, mouseY);  
                    break;  
            case 4: g.setColor (Color.cyan);  
                    g.fillOval (mouseX, mouseY, 40, 30);  
                    break;  
        }  
    }  
}
```

```

        case 5: g.setColor (Color.gray);
                g.drawRoundRect (mouseX, mouseY, 40, 30, 8, 8);
                break;
        case 6: g.setColor (Color.magenta);
                g.fillRect (mouseX, mouseY, 40, 30);
                break;
        case 7: g.setColor (Color.black);
                g.drawArc (mouseX, mouseY, 40, 30, 120, 300);
                break;
        case 8: g.setColor (Color.orange);
                g.fillArc (mouseX, mouseY, 40, 30, 120, 300);
                break;
        case 9: g.setColor (Color.blue);
                g.drawString ("HelloWorld!", mouseX, mouseY);
                cont=0;
    }
}
public void mousePressed (MouseEvent e) {
    mouseX = e.getX();
    mouseY = e.getY();
    repaint ();
    cont++;
}
public void update (Graphics g) {
    paint (g);
}
public void mouseClicked (MouseEvent e) { }
public void mouseEntered (MouseEvent e) { }
public void mouseExited (MouseEvent e) { }
public void mouseReleased (MouseEvent e) { }
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
public void windowOpened(WindowEvent e) { }
public void windowIconified(WindowEvent e) { }
public void windowDeiconified(WindowEvent e) { }
public void windowClosed(WindowEvent e) { }
public void windowActivated(WindowEvent e) { }
public void windowDeactivated(WindowEvent e) { }
}

```

## 9.7 Exibindo Som e Imagem em um Applet

Para tocar um som ou exibir uma imagem em um applet é necessário aprendermos primeiramente para que servem alguns métodos da classe Applet:

- **getDocumentBase():** retorna um objeto do tipo URL que contém a URL do documento HTML.
- **getCodeBase():** retorna um objeto do tipo URL que contém a URL do código Java ".class". Se o arquivo ".class" encontra-se no mesmo diretório que a página HTML, estes dois métodos retornarão o mesmo valor.
- **getImage (URL base, String target):** retorna um objeto do tipo Image que contém a imagem buscada na URL especificada.
- **getAudioClip (URL base, String target):** retorna um objeto do tipo AudioClip que contém o som de nome target localizado na URL target. Java aceita arquivos de áudio terminação ".au".
- **getParameter (String nomeAtributo):** obtém o valor de um atributo especificado na página HTML que chama o applet.

Além destes é importante ressaltar os métodos da classe AudioClip que permitem tocar e terminar um som:

- **loop():** inicia um som e o reinicia automaticamente ao terminar;
- **start():** toca um som uma única vez;
- **stop():** finaliza a execução de um som;

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

public class AppletSomImagem extends Applet {
    Image img1;
    AudioClip audioClip;
    public void init() {
        String strSound, strImg1, strImg2;

        strImg1 = getParameter("imagem1");
        img1 = getImage(getDocumentBase(), strImg1);

        strSound = getParameter("som");
        audioClip = getAudioClip(getDocumentBase(), strSound);
    }
    public void paint(Graphics g) {
        g.drawImage(img1, 0, 0, this);
    }
    public void start( ) {
        audioClip.play();
    }
    public void stop() {
        audioClip.stop();
    }
}
```

A página HTML que chama o applet:

```
<HTML>
<HEAD>
    <TITLE> Testando Recursos em um Applet </TITLE>
</HEAD>
<BODY>
    <APPLET CODE = "AppletSomImagem.class" WIDTH = 300 HEIGHT = 200>
        <PARAM NAME = imagem1 VALUE="Orb.gif">
        <PARAM NAME = som VALUE="Som.au">
    </APPLET>
</BODY>
</HTML>
```

## 10 Interface Gráfica – Biblioteca Swing

Quando o Java 1.0 foi introduzido, ele continha um conjunto de classes chamado Abstract Window Toolkit (AWT) para programação de interfaces gráficas. A maneira como a biblioteca AWT trabalhava com os elementos gráficos era delegando sua criação e comportamento ao toolkit GUI nativo de cada plataforma (Windows, Solaris, etc). Por isso, os componentes de interface gráfica tinham a aparência (look-and-feel) dos componentes da plataforma.

Java criou um conjunto de bibliotecas para definir elementos gráficos de interface (botões, menus e outros) que possuem a mesma aparência e comportamento, independente da plataforma onde são executados. Essa biblioteca é chamada de SWING. Além disso, o Swing tem um conjunto maior de elementos gráficos de interface.

Assim, como a interface gráfica pode ter a mesma aparência em diferentes plataformas, o Swing permite que o programador defina o "look and feel" desejado, ou seja, aquele em que o usuário já está mais acostumado. A Sun desenvolveu um "look and feel" chamado Metal que é o "look and feel" default das aplicações gráficas em Swing. Além desse, existem o "Windows" (aparência da GUI do Windows) e Motif (aparência da GUI do Solaris). O Swing faz parte do Java Foundation Classes (JFC), uma biblioteca mais vasta que inclui também uma API 2D entre outras.

Para utilizar os componentes do pacote Swing é necessário importar o pacote `javax.swing.*` ; .

### 10.1 Determinando o Look-and-Feel de uma Interface Gráfica em Java

O programa Java irá primeiramente verificar se foi definido algum Look and Feel. Caso positivo, ele utiliza este, senão ele utiliza o Look and Feel default Java que é o "Metal".

Para determinar o Look and Feel utilize o método `UIManager.setLookAndFeel()`:

```
UIManager.setLookAndFeel (UIManager.getCrossPlatformLookAndFeelClassName());
```

No exemplo acima, o Look and Feel utilizado foi o default Java Metal. Nesse caso era desnecessário determinar o Look and Feel, pois o "Metal" já é utilizado por default.

É possível ainda determinar que a interface gráfica Java tenha o Look and Feel do ambiente em que está rodando. Por exemplo, para ela ter a aparência Windows se estiver executando no Windows, Motif se estiver em um Solaris.

```
UIManager.setLookAndFeel (UIManager.getSystemLookAndFeelClassName());
```

Para utilizar um determinado Look and Feel, devemos fornecer o nome da classe.

Para determinar um Look and Feel "Metal":

```
UIManager.setLookAndFeel ("javax.swing.plaf.metal.MetalLookAndFeel");
```

Para determinar um Look and Feel "Windows":

```
UIManager.setLookAndFeel ("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

Para determinar um Look and Feel Motif:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

Para determinar um Look and Feel "MAC" (é válido apenas se a aplicação Java estiver executando em uma plataforma MacOS):

```
UIManager.setLookAndFeel("javax.swing.plaf.mac.MacLookAndFeel");
```



LookAndFeel

Metal

Windows

Motif

O exemplo a seguir mostra uma aplicação Java utilizando o Look and Feel da plataforma:

```
import javax.swing.*;
public class LookAndFeel {
    public static void main (String args[]) {
        try {
            /* Seta para Look and Feel da plataforma (por ex. Windows ) */
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }
        catch (Exception e) { }
        JFrame f = new JFrame ( );
        JButton button = new JButton("Hello, world");
        f.getContentPane().add (button);
        f.pack();
        f.show ();
    }
}
```

Para modificar o Look And Feel de um frame de um programa já inicializado é necessário também chamar o método `SwingUtilities.updateComponentTreeUI` para atualização dos componentes. O código fica assim:

```
UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
SwingUtilities.updateComponentTreeUI (f.getContentPane());
```

Geralmente, os componentes da biblioteca Swing possuem um componente equivalente na biblioteca AWT com um nome similar acrescido da letra J no início da palavra. Assim, temos o componente Swing JFrame para o Frame do AWT, JButton para Button, etc.

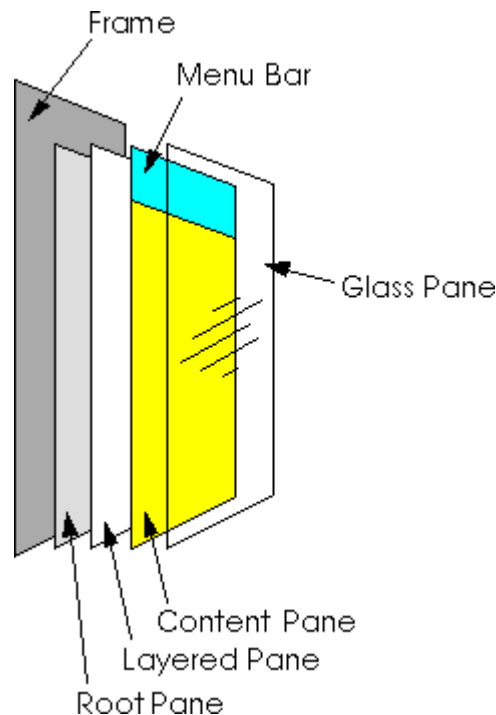
## 10.2 JFrame

Assim, como na biblioteca AWT existe a janela Frame, no Swing há o JFrame. Frames são containers e, por isso, podem conter outros componentes de interface como botões, labels, etc.

A estrutura de um Frame é bastante complexa como pode ser observado na Figura 3.

Existem 4 camadas em um JFrame: root pane, layered pane, glass pane, que não interessam ao programador e que são necessários para organizar a barra de menu, e o content pane para implementar o Look and Feel. É o content pane que é usado pelo programador para adicionar os componentes ao Frame. Por exemplo:

```
Container contentPane = frame.getContentPane ( );
JPanel p = new JPanel ( );
contentPane.add (p);
```



**Figura 3: Estrutura Interna de um Frame.**

Os que estavam familiarizados com a biblioteca AWT devem ter notado que ao contrário do que fazíamos, adicionar o componente diretamente no frame, agora adicionamos o componente na camada content pane do JFrame.

AWT	SWING
<pre>Frame frame = new Frame ( );</pre>	<pre>JFrame frame = new JFrame ( );</pre>
<pre>Panel p = new Panel ( );</pre>	<pre>Container contentPane = frame.getContentPane ( );</pre>
<pre>frame.add (p);</pre>	<pre>JPanel p = new JPanel ( );</pre>
	<pre>contentPane.add (p);</pre>

Pode-se desenhar uma mensagem diretamente em um JFrame, mas essa não é uma boa prática de programação. O ideal é colocar a mensagem dentro de um Panel e adicionar este componente ao Frame.

No Swing não existe um componente especial para desenho como o Canvas na biblioteca AWT. Para desenho no Swing deve-se usar o JPanel. Além disso, o método que deve ser sobrescrito é o



```
public void paintComponent (Graphics g){
    super.paintComponent (g);
    ... // código restante
}
```

ao invés do método paint sobrescrito no AWT.

### 10.2.1 Capturando os eventos de fechar janela no JFrame

No Frame do AWT quando clicávamos no botão de fechar não acontecia nada até que tratássemos o evento de fechar a janela. No Swing, por default, a janela é escondida quando o usuário clicar no botão de fechar.

O programador pode determinar outras opções para quando o usuário clicar no botão de fechar. Essas opções devem ser determinadas pelo método `setDefaultCloseOperation` do frame. Os argumentos para este método podem ser:

- `DO_NOTHING_ON_CLOSE` – não faz nada quando o usuário clicar no botão de fechar. Neste caso, o programa poderia usar um window listener que execute uma outra ação em seu método `windowClosing`.
- `HIDE_ON_CLOSE` (o default) – Esconde o frame. Remove-o da tela.
- `DISPOSE_ON_CLOSE` – Faz a mesma coisa que o método `dispose()`. Remove o frame da tela e libera os recursos usados.

Observe que a janela do frame será escondida, mas a aplicação Java não é finalizada e continuará executando até a chamada do método `System.out.exit(0)` ou até a finalização do programa pelo sistema operacional.

## 10.3 Capturando eventos nos componentes Swing

Os eventos em componentes Swing são capturados pela mesma biblioteca de eventos do AWT (O Modelo de Delegação de Eventos), vistos no capítulo de Interface Gráfica AWT desta apostila. Por isso, devemos sempre tomar o cuidado de importar o pacote `java.awt.event.*`.

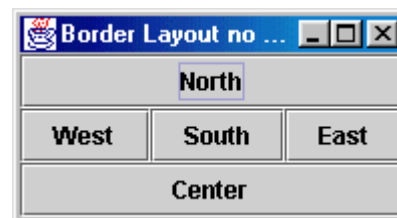
A partir desta seção serão exibidos os componentes Swing mais utilizados na criação de interfaces gráficas, bem como o tratamento de eventos destes componentes.

## 10.4 Determinando o Layout Manager do Frame

É possível também determinar um layout para um frame (ver seção de Gerenciadores de Layout desta apostila). Esse layout deve ser aplicado ao content pane, assim como os componentes (botões, labels, etc) também devem ser adicionados ao content pane do JFrame. No exemplo abaixo, estamos aplicando um `BorderLayout` ao Frame.

O **BorderLayout** é o layout default para content pane, ou seja, para frames, applets e dialogs. O **FlowLayout** é o layout default para panels em Swing.

```
import java.awt.Container;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;
class GerenciadorBorderLayout {
    public static void main (String args[]) {
        JFrame f = new JFrame ("Border Layout no Swing");
        BorderLayout layout = new BorderLayout ( );
        Container c = f.getContentPane();
        c.setLayout (layout);
        JButton b1 = new JButton ("North");
        JButton b2 = new JButton ("Center");
        JButton b3 = new JButton ("South");
        JButton b4 = new JButton ("West");
        JButton b5 = new JButton ("East");
        c.add (BorderLayout.NORTH, b1);
        c.add (b2, BorderLayout.SOUTH);
        c.add (b3, BorderLayout.CENTER);
        c.add (b4, BorderLayout.WEST);
        c.add (b5, BorderLayout.EAST);
        f.addWindowListener(new WindowAdapter() {
```





```

        public void windowClosing(WindowEvent e)
    {
        {System.exit(0);} }
        f.pack();
        f.show();
    }
}

```

## 10.5 JPanel

No JPanel adicionamos os componentes diretamente nele utilizando o método `add()`, assim como para o Panel (AWT). Da mesma maneira, determinamos o layout de um panel, chamando o método `setLayout` do objeto do tipo JPanel. Veja o código abaixo:

```

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

class JPanelDemo {
    public static void main (String args[]) {
        JFrame f = new JFrame ();
        JPanel panel = new JPanel ();
        FlowLayout layout = new FlowLayout (FlowLayout.LEFT);
        panel.setLayout (layout);
        JButton b1 = new JButton ("um");
        JButton b2 = new JButton ("dois");
        JButton b3 = new JButton ("três");
        JButton b4 = new JButton ("quatro");
        panel.add (b1);
        panel.add (b2);
        panel.add (b3);
        panel.add (b4);
        f.getContentPane().add (panel);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
        });
        f.pack();
        f.show();
    }
}

```

## 10.6 JButton

Os botões são implementados em Swing através da classe JButton. O JButton é subclasse da classe AbstractButton, por isso, muitas das suas funcionalidades podem ser encontradas na documentação da classe pai, tais como, definição de um texto e de uma imagem a ser exibida pelo botão.

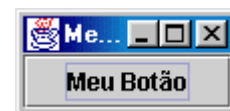
### 10.6.1 Um exemplo simples de botão:

```

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

class JButtonDemo {
    public static void main (String args []) {
        JFrame f = new JFrame ("Meu Frame");
        JButton button = new JButton ("Meu Botão");
        button.addActionListener (new TratadorDeBotao());
        Container c = f.getContentPane ();
        c.add (button);
        f.pack();
        f.show ();
    }
}

```

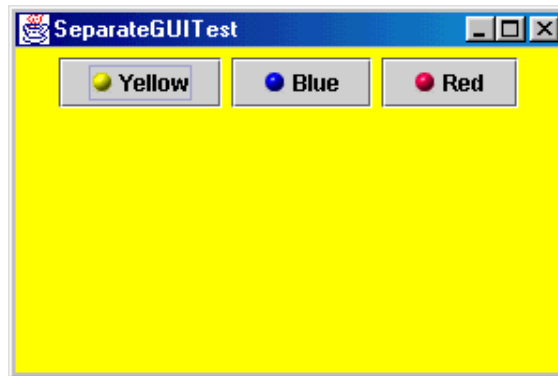


```

    }
}
class TratadorDeBotao implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String str = e.getActionCommand();
        if (str.equals ("Meu Botão")) System.out.println ("Botão
pressionado.");
    }
}
}

```

### 10.6.2 Desenhando botões com Imagens:



```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class SeparateGUIFrame extends JFrame implements ActionListener {
    JPanel panel;
    public SeparateGUIFrame() {
        setTitle("SeparateGUI Test");
        setSize(300, 200);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0); } } );
        panel = new JPanel();

        JButton blueAction = new JButton("Blue", new ImageIcon("blue-ball.gif"));
        JButton yellowAction = new JButton("Yellow", new ImageIcon("yellow-
ball.gif"));
        JButton redAction = new JButton("Red", new ImageIcon("red-ball.gif"));

        blueAction.addActionListener (this);
        yellowAction.addActionListener (this);
        redAction.addActionListener (this);

        panel.add(yellowAction);
        panel.add(blueAction);
        panel.add(redAction);

        Container contentPane = getContentPane();
        contentPane.add(panel);
    }
    public void actionPerformed(ActionEvent evt) {
        Color c;
        String s = evt.getActionCommand ();
        if (s.equals ("Red")) c = Color.red;
        else if (s.equals ("Blue")) c = Color.blue;
        else c = Color.yellow;

        panel.setBackground(c);
        panel.repaint();
    }
}

```

```

    }
}
public class ButtonComImagem {
    public static void main(String[] args) {
        JFrame frame = new SeparateGUIFrame();
        frame.show();
    }
}

```

## 10.7 JLabel

Para a criação de labels no Swing temos o JLabel. O JLabel permite a apresentação de conteúdo puramente textual, assim como o Label do AWT, e também de figuras. Veja o exemplo a seguir retirado de (Campione, 2000):

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JLabelDemo extends JPanel {
    JLabel label1, label2, label3;
    public JLabelDemo() {
        ImageIcon icon = new ImageIcon("middle.gif", "a pretty but meaningless
splat");
        setLayout(new GridLayout(3,1));
        label1 = new JLabel("Image and Text", icon, JLabel.CENTER);
        //Set the position of the text, relative to the icon:
        label1.setVerticalTextPosition(JLabel.BOTTOM);
        label1.setHorizontalTextPosition(JLabel.CENTER);
        label1.setOpaque (true); // para o fundo não ficar transparente
        label1.setBackground (Color.white);
        label2 = new JLabel("Text-Only Label");
        label2.setOpaque (true);
        label2.setBackground (Color.yellow);
        label3 = new JLabel(icon);
        label3.setOpaque (true);
        label3.setBackground (Color.blue);
        add(label1);
        add(label2);
        add(label3);
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("LabelDemo");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e)
            {System.exit(0);});
        frame.getContentPane().add(new JLabelDemo(), BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
    }
}

```



## 10.8 JTextField e JTextArea

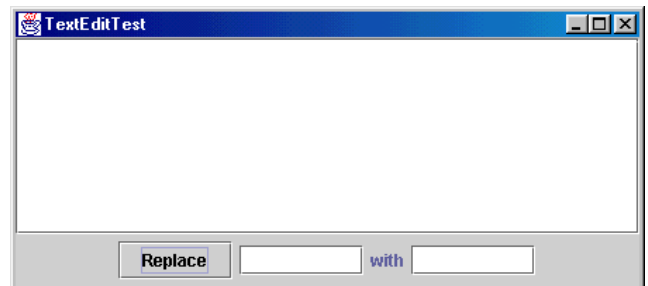
O JTextField é uma área de edição de texto de uma única linha. Este componente equivale ao TextField do AWT. Quando o usuário entrar todo o texto no JTextField, ele pode clicar no botão return que acionará um evento do tipo ActionEvent (ver TextField no AWT).

Quando desejamos editar um texto de múltiplas linhas, devemos usar um JTextArea (equivalente ao TextArea da biblioteca AWT). Como no JTextArea o return insere uma nova linha, é usual utilizar um JButton para capturar eventos associados a um JTextArea.

O exemplo abaixo mostra uma aplicação retirada de (Horstman & Cornell, 2000) que mostra como fazer a substituição de um texto especificado dentro de um JTextArea.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class TextEditFrame extends JFrame
{
    public TextEditFrame()
    {
        setTitle("TextEditTest");
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        Container contentPane = getContentPane();
        JPanel panel = new JPanel();
        JButton replaceButton = new JButton("Replace");
        panel.add(replaceButton);
        replaceButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                String f = from.getText();
                int n = textArea.getText().indexOf(f);
                // substitui o texto especificado, da posição inicial até a
                posição final
                if (n >= 0 && f.length() > 0)
                    textArea.replaceRange(to.getText(), n,
                                            n + f.length());
            }
        });
        from = new JTextField(8);
        panel.add(from);
        panel.add(new JLabel("with"));
        to = new JTextField(8);
        panel.add(to);
        textArea = new JTextArea(8, 40);
        scrollPane = new JScrollPane(textArea);
        contentPane.add(panel, "South");
        contentPane.add(scrollPane, "Center");
        pack();
    }
    private JScrollPane scrollPane;
    private JTextArea textArea;
    private JTextField from, to;
}

public class TextDemo {
    public static void main(String[] args)
    {
        JFrame f = new TextEditFrame();
        f.show();
    }
}
```



Para implementar uma área de obtenção de password (senha) existe a classe JPasswordField que substitui os caracteres digitados pelo usuário por \* na caixa de texto.

## 10.9 JCheckBox

O JCheckBox funciona de maneira análoga ao CheckBox do AWT, ou seja, permite a escolha de um ou mais itens a partir de um conjunto de elementos exibidos. Veja o exemplo abaixo que foi retirado de (Campione, 2000).

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

```

public class CheckBoxDemo extends JPanel {
    JCheckBox chinButton;
    JCheckBox glassesButton;
    JCheckBox hairButton;
    JCheckBox teethButton;
    /*
     * Four accessory choices provide for 16 different
     * combinations. The image for each combination is
     * contained in a separate image file whose name indicates
     * the accessories. The filenames are "geek-XXXX.gif"
     * where XXXX can be one of the following 16 choices.
     * The "choices" StringBuffer contains the string that
     * indicates the current selection and is used to generate
     * the file name of the image to display.
     ----                // zero accessories
     c---                // one accessory
     -g--
     --h-
     ---t
     cg--                // two accessories
     c-h-
     c--t
     -gh-
     -g-t
     --ht
     -ght                // three accessories
     c-ht
     cg-t
     cgh-
     cght                // all accessories
     */
    StringBuffer choices;
    JLabel pictureLabel;
    public CheckBoxDemo() {
        // Create the check boxes
        chinButton = new JCheckBox("Chin");
        chinButton.setMnemonic('c');
        chinButton.setSelected(true);
        glassesButton = new JCheckBox("Glasses");
        glassesButton.setMnemonic('g');
        glassesButton.setSelected(true);
        hairButton = new JCheckBox("Hair");
        hairButton.setMnemonic('h');
        hairButton.setSelected(true);
        teethButton = new JCheckBox("Teeth");
        teethButton.setMnemonic('t');
        teethButton.setSelected(true);
        // Register a listener for the check boxes.
        CheckBoxListener myListener = new CheckBoxListener();
        chinButton.addItemListener(myListener);
        glassesButton.addItemListener(myListener);
        hairButton.addItemListener(myListener);
        teethButton.addItemListener(myListener);
        // Indicates what's on the geek.
        choices = new StringBuffer("cght");
        // Set up the picture label
        pictureLabel = new JLabel(new ImageIcon(
            "geek/geek-"
            + choices.toString()
            + ".gif"));
        pictureLabel.setToolTipText(choices.toString());
        // Put the check boxes in a column in a panel
        JPanel checkPanel = new JPanel();
        checkPanel.setLayout(new GridLayout(0, 1));
        checkPanel.add(chinButton);
    }
}

```



```

        checkPanel.add(glassesButton);
        checkPanel.add(hairButton);
        checkPanel.add(teethButton);
        setLayout(new BorderLayout());
        add(checkPanel, BorderLayout.WEST);
        add(pictureLabel, BorderLayout.CENTER);
        setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
    }
    /** Listens to the check boxes. */
    class CheckBoxListener implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            int index = 0;
            char c = '-';
            Object source = e.getItemSelectable();
            if (source == chinButton) {
                index = 0;
                c = 'c';
            } else if (source == glassesButton) {
                index = 1;
                c = 'g';
            } else if (source == hairButton) {
                index = 2;
                c = 'h';
            } else if (source == teethButton) {
                index = 3;
                c = 't';
            }
            if (e.getStateChange() == ItemEvent.DESELECTED)
                c = '-';
            choices.setCharAt(index, c);
            pictureLabel.setIcon(new ImageIcon("geek/geek-"+ choices.toString()+
".gif"));
            pictureLabel.setToolTipText(choices.toString());
        }
    }
    public static void main(String s[]) {
        JFrame frame = new JFrame("CheckBoxDemo");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
        });
        frame.setContentPane(new CheckBoxDemo());
        frame.pack();
        frame.setVisible(true);
    }
}

```

## 10.10 JRadioButton

No Swing, existe a classe `JRadioButton` para implementação de botões radio. O usuário pode escolher apenas uma das opções de `JRadioButton` num dado instante, enquanto que o `JCheckBox` permite múltipla escolha, se assim definido pelo programador.

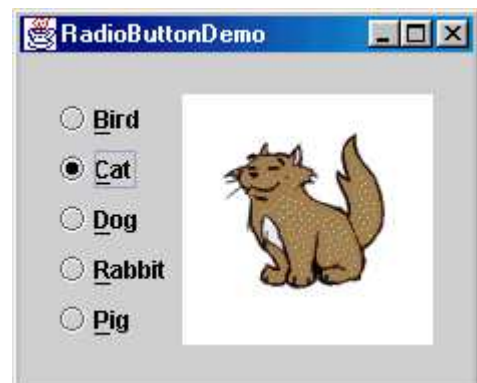
Como o `JRadioButton` é subclasse de `AbstractButton`, ele pode possuir todas as funcionalidades que um botão possui. Por exemplo, um `JRadioButton` pode ser formado por uma imagem e um texto.

O exemplo abaixo foi adaptado de (Campione, 2000).

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class RadioButtonDemo extends JPanel {
    static JFrame frame;
    static String birdString = "Bird";

```



```

static String catString = "Cat";
static String dogString = "Dog";
static String rabbitString = "Rabbit";
static String pigString = "Pig";
JLabel picture;
public RadioBottonDemo() {
    JRadioButton birdButton = new JRadioButton(birdString);
    birdButton.setMnemonic('b');
    birdButton.setSelected(true);
    JRadioButton catButton = new JRadioButton(catString);
    catButton.setMnemonic('c');
    JRadioButton dogButton = new JRadioButton(dogString);
    dogButton.setMnemonic('d');
    JRadioButton rabbitButton = new JRadioButton(rabbitString);
    rabbitButton.setMnemonic('r');
    JRadioButton pigButton = new JRadioButton(pigString);
    pigButton.setMnemonic('p');
    // Group the radio buttons.
    ButtonGroup group = new ButtonGroup();
    group.add(birdButton);
    group.add(catButton);
    group.add(dogButton);
    group.add(rabbitButton);
    group.add(pigButton);
    // Register a listener for the radio buttons.
    RadioListener myListener = new RadioListener();
    birdButton.addActionListener(myListener);
    catButton.addActionListener(myListener);
    dogButton.addActionListener(myListener);
    rabbitButton.addActionListener(myListener);
    pigButton.addActionListener(myListener);
    // Set up the picture label
    picture = new JLabel(new ImageIcon(birdString + ".gif"));
    // Put the radio buttons in a column in a panel
    JPanel radioPanel = new JPanel();
    radioPanel.setLayout(new GridLayout(5, 1));
    radioPanel.add(birdButton);
    radioPanel.add(catButton);
    radioPanel.add(dogButton);
    radioPanel.add(rabbitButton);
    radioPanel.add(pigButton);
    setLayout(new BorderLayout());
    add(radioPanel, BorderLayout.WEST);
    add(picture, BorderLayout.CENTER);
    // deixa espaco em branco de 20 pixels em torno do panel
    setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
}

/** Listens to the radio buttons. */
class RadioListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        picture.setIcon(new ImageIcon(e.getActionCommand() + ".gif"));
    }
}

public static void main(String s[]) {
    frame = new JFrame("RadioBottonDemo");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e)
        {System.exit(0);});
    frame.getContentPane().add(new RadioBottonDemo(), BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
}

```

## 10.11 JComboBox

O exercício anterior poderia ser feito utilizando um JComboBox ao invés de RadioButton. Um JComboBox exibe uma lista de elementos a qual só pode ser selecionado um único item.

O JComboBox é a implementação no Swing da classe Choice do AWT. O exemplo abaixo foi retirado de (Campione, 2000).

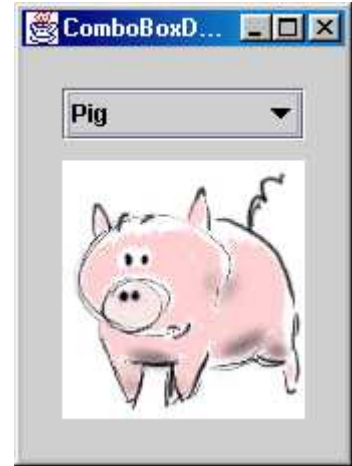
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBoxDemo extends JPanel {
    JLabel picture;

    public ComboBoxDemo() {
        String[] petStrings = { "Bird", "Cat", "Dog",
                                "Rabbit", "Pig" };

        JComboBox petList = new JComboBox(petStrings);
        petList.setSelectedIndex(4);
        petList.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JComboBox cb = (JComboBox)e.getSource();
                String petName = (String)cb.getSelectedItem();
                picture.setIcon(new ImageIcon(petName + ".gif"));
            }
        });
        picture = new JLabel(new ImageIcon(petStrings[petList.getSelectedIndex()] +
        ".gif"));
        picture.setBorder(BorderFactory.createEmptyBorder(10,0,0,0));
        setLayout(new BorderLayout());
        add(petList, BorderLayout.NORTH);
        add(picture, BorderLayout.SOUTH);
        setBorder(BorderFactory.createEmptyBorder(20,20,20,20));
    }

    public static void main(String s[])
    {
        JFrame frame = new JFrame("ComboBoxDemo");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
        });
        frame.getContentPane().add(new ComboBoxDemo(), BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
    }
}
```



## 10.12 JList

Um JList permite a seleção de múltiplos elementos a partir de uma lista exibida. É mais útil utilizar lista do que Checkox quando houver muitos elementos a serem exibidos que ocupem muito espaço na tela, pois ela permite a utilização de scroll. Uma lista não possui um scroll, é necessário inserir a lista em um scroll pane. Na biblioteca AWT a lista é implementada pela classe List e no Swing pela classe JList.

Uma lista pode ser construída a partir de um array de strings. Por exemplo:

```
String itens [] = { "lápis", "caderno", "caneta", "livro", "aluno"};
JList list = new JList (itens);
```

Para inserir a lista em um scroll pane, use:






```
JScrollPane scroll = new JScrollPane (list);
```

Por default, uma lista pode exibir 8 elementos. Para modificar esse valor (por ex. para 10) use o método `setVisibleRowCount(10)`.

Por default, o usuário pode selecionar mais de um elemento de uma lista. Para selecionar mais de um item, use a tecla CTRL enquanto clicar em cada item. Para selecionar uma lista contínua de elementos, clique no primeiro item, tecle SHIFT e clique no último item.

Para restringir a seleção que o usuário pode fazer, use o método `setSelectionMode()`. Por exemplo:

Seleção de um único item por vez	<code>list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);</code>	
Seleção de um item ou um intervalo contínuo de itens	<code>list.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);</code>	
Qualquer combinação de seleção é permitida. Este é o valor default.	<code>list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);</code>	

Para pegar eventos em uma lista (seleção de elementos pelo usuário) é necessário implementar o listener de eventos de listas `ListSelectionListener` que deve ter o método `public void valueChanged(ListSelectionEvent evt)`.

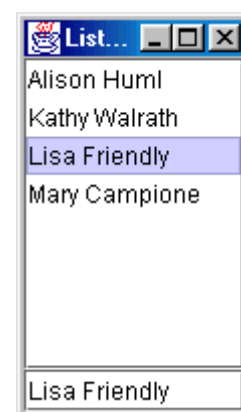
```
class ListaListener implements ListSelectionListener
```

O exemplo abaixo exibe um `JList` criado a partir de uma lista de strings. Os itens selecionados na lista são exibidos na área de texto abaixo do frame.

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

public class ListaSimples extends JFrame implements ListSelectionListener {
    private JList list;
    private JTextField employeeName;

    public ListaSimples() {
        super("Lista Simples");
        String [] str = {"Alison Huml", "Kathy Walrath",
                        "Lisa Friendly", "Mary Campione"};
        list = new JList (str);
        list.setSelectionMode
            (ListSelectionModel.SINGLE_SELECTION);
        list.setSelectedIndex(0);
        list.addListSelectionListener(this);
        JScrollPane listScrollPane = new JScrollPane(list);
        employeeName = new JTextField(10);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        contentPane.add(listScrollPane, BorderLayout.CENTER);
        contentPane.add(employeeName, BorderLayout.SOUTH);
    }
}
```



```

public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting() == false) {
        if (list.getSelectedIndex() == -1)
            employeeName.setText("");
        else {
            String name = list.getSelectedValue().toString();
            employeeName.setText(name);
        }
    }
}

public static void main(String s[]) {
    JFrame frame = new ListaSimples();
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    });
    frame.pack();
    frame.setVisible(true);
}
}

```

Observe que usamos o método **e.getValueIsAdjusting()** do evento que retorna um valor booleano. Toda vez que o usuário pressiona o botão do mouse apontando para um item da lista é notificado um evento e é chamado o método `valueChanged()`, mesmo que o usuário não tenha soltado o botão pressionado do mouse. O método **e.getValueIsAdjusting()** retorna `false` se foi realizada uma seleção pelo usuário com o mouse, ou seja, o usuário pressionou e soltou o botão do mouse; ou `true` se a seleção ainda está se modificando, ou seja, o usuário está apontando com o mouse um item com o botão pressionado, mas ainda não soltou o botão.

Uma vez que o evento tenha sido notificado é necessário obter quais itens foram selecionados. O método **list.getSelectedValue()** retorna o elemento selecionado. Se a lista permite múltipla seleção deve-se usar o método **list.getSelectedValues()**.

### 10.12.1 ListModel

Você deve ter observado que os métodos para adicionar ou retirar elementos em uma lista não estão contidos na classe `JList`. Isso porque as listas usam o padrão model-view-controller para separar a aparência visual dos dados na lista. O `JList` é responsável pela aparência visual, enquanto que para adicionarmos dados (objetos) em uma lista precisamos de um `ListModel`.

Um `ListModel` é uma interface que deve ser implementada. Para simplesmente selecionar e extrair elementos de uma lista existe a classe `DefaultListModel`.

Para adicionarmos elementos em uma lista usamos uma instância de `DefaultListModel`. Essa instância é criada e passada ao método construtor da `JList`.

```

listModel = new DefaultListModel();
listModel.addElement("Alison Huml");
listModel.addElement("Kathy Walrath");
listModel.addElement("Lisa Friendly");
listModel.addElement("Mary Campione");
list = new JList(listModel);

```

Para removermos um elemento da lista usamos o método:

```
listModel.remove(int index);
```

Para inserirmos um item por outro, podemos usar o método:

```
listModel.insertElementAt(Object novoItem, int posicao);
```

O próximo exemplo, retirado de (Campione, 2000), permite a seleção e duplicação dos elementos apresentados em uma lista.

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

public class ListDemo extends JFrame
    implements ListSelectionListener {
    private JList list;
    private DefaultListModel listModel;

    private static final String hireString = "Hire";
    private static final String fireString = "Fire";
    private JButton fireButton;
    private JTextField employeeName;

    public ListDemo() {
        super("ListDemo");
        listModel = new DefaultListModel();
        listModel.addElement("Alison Huml");
        listModel.addElement("Kathy Walrath");
        listModel.addElement("Lisa Friendly");
        listModel.addElement("Mary Campione");

        //Create the list and put it in a scroll pane
        list = new JList(listModel);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.setSelectedIndex(0);
        list.addListSelectionListener(this);
        JScrollPane listScrollPane = new JScrollPane(list);

        JButton hireButton = new JButton(hireString);
        hireButton.setActionCommand(hireString);
        hireButton.addActionListener(new HireListener());

        fireButton = new JButton(fireString);
        fireButton.setActionCommand(fireString);
        fireButton.addActionListener(new FireListener());

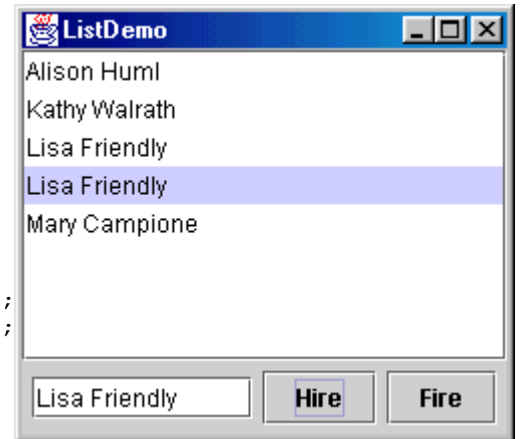
        employeeName = new JTextField(10);
        employeeName.addActionListener(new HireListener());
        String name = listModel.getElementAt(list.getSelectedIndex()).toString();
        employeeName.setText(name);

        //Create a panel that uses FlowLayout (the default).
        JPanel buttonPane = new JPanel();
        buttonPane.add(employeeName);
        buttonPane.add(hireButton);
        buttonPane.add(fireButton);

        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        contentPane.add(listScrollPane, BorderLayout.CENTER);
        contentPane.add(buttonPane, BorderLayout.SOUTH);
    }

    class FireListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {

            //This method can be called only if
            //there's a valid selection
            //so go ahead and remove whatever's selected.
            int index = list.getSelectedIndex();
```



```

        listModel.remove(index);

        int size = listModel.getSize();

        //Nobody's left, disable firing
        if (size == 0) {
            fireButton.setEnabled(false);

            //Adjust the selection
        } else {
            if (index == listModel.getSize())//removed item in last position
                index--;
            list.setSelectedIndex(index);    //otherwise select same index
        }
    }
}

//This listener is shared by the text field and the hire button
class HireListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {

        //User didn't type in a name...
        if (employeeName.getText().equals("")) {
            Toolkit.getDefaultToolkit().beep();
            return;
        }

        int index = list.getSelectedIndex();
        int size = listModel.getSize();

        //If no selection or if item in last position is selected,
        //add the new hire to end of list, and select new hire
        if (index == -1 || (index+1 == size)) {
            listModel.addElement(employeeName.getText());
            list.setSelectedIndex(size);

            //Otherwise insert the new hire after the current selection,
            //and select new hire
        } else {
            listModel.insertElementAt(employeeName.getText(), index+1);
            list.setSelectedIndex(index+1);
        }
    }
}

public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting() == false) {

        //No selection, disable fire button
        if (list.getSelectedIndex() == -1) {
            fireButton.setEnabled(false);
            employeeName.setText("");

            //Selection, update text field
        } else {
            fireButton.setEnabled(true);
            String name = list.getSelectedValue().toString();
            employeeName.setText(name);
        }
    }
}

public static void main(String s[]) {
    JFrame frame = new ListDemo();
}

```

```

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
        });

        frame.pack();
        frame.setVisible(true);
    }
}

```

### 10.12.2 ListCellRendering

Uma JList usa um objeto chamado cell renderer para exibir cada um de seus componentes. O cell renderer default sabe exibir strings e ícones. Porém, se desejarmos colocar outro tipo de objeto em uma lista ou se desejarmos modificar o modo como o cell renderer exibe strings e ícones, temos que implementar um cell renderer específico. Para implementar um cell renderer deve-se:

- Escrever uma classe que implemente a interface ListCellRenderer.
- Criar uma instância de sua classe e chamar o método setCellRenderer passando a instância como argumento.

A classe que implementa o cell renderer deve implementar o método getListCellRendererComponent. Esse método retorna um componente o qual o método paintComponent do objeto é usado para exibi-lo na lista. Por isso, além do cell renderer implementar o ListCellRenderer, é mais simples se ele for subclasse de algum outro componente que se deseja exibir na lista. No exemplo abaixo, queremos mostrar uma lista com strings e ícones. Faremos isso, adicionando esses componentes a um JLabel que será exibido na lista.

```

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

```

```

public class CustomListDemo extends JPanel {
    ImageIcon images[];

    public CustomListDemo() {
        String[] names = { "Blue", "Red", "Yellow"};
        ImageIcon [] images = new ImageIcon [names.length];
        images [0] = new ImageIcon("blue-ball.gif");
        images[0].setDescription("Blue");
        images [1] = new ImageIcon("red-ball.gif");
        images[1].setDescription("Red");
        images [2] = new ImageIcon("yellow-ball.gif");
        images[2].setDescription("Yellow");

        JList lista = new JList(images);
        ListRenderer renderer= new ListRenderer();
        lista.setCellRenderer(renderer);

        lista.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        lista.setSelectedIndex(0);
        lista.setVisibleRowCount (3);
        JScrollPane listScrollPane = new JScrollPane(lista);

        setLayout(new BorderLayout());
        add (listScrollPane, BorderLayout.NORTH);
    }

    class ListRenderer extends JLabel implements ListCellRenderer {
        public ListRenderer() {
            setOpaque(true);
        }
        public Component getListCellRendererComponent(
            JList list,
            Object value,

```



```

        int index,
        boolean isSelected,
        boolean cellHasFocus) {
    if (isSelected) {
        setBackground(list.getSelectionBackground());
        setForeground(list.getSelectionForeground());
    } else {
        setBackground(list.getBackground());
        setForeground(list.getForeground());
    }

    ImageIcon icon = (ImageIcon)value;
    setText(icon.getDescription());
    setIcon(icon);
    return this;
}

public static void main(String s[]) {
    JFrame frame = new JFrame("CustomComboBoxDemo");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);} });

    frame.getContentPane().add(new CustomListDemo(), BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
}

```

## 10.13 Caixas de Diálogo (JDialog e JOptionPane)





Dialog Box, ou seja, caixas de diálogo, são usadas para advertência e para fornecer e obter informações do usuário. As dialog boxes podem ser modais, escravas do frame (só é possível voltar ao frame se a dialog box for fechada), ou não modais.

No Swing as dialog boxes são criadas a partir da classe `JOptionPane` ou da classe `JDialog`. Use `JOptionPane` quando desejar criar caixas de diálogos mais simples e `JDialog` quando desejar uma caixa de diálogo mais sofisticada com todas as opções de um frame.

### 10.13.1 JOptionPane

Usando `JOptionPane` é possível criar diferentes tipos de diálogos. `JOptionPane` fornece suporte para diálogos fornecendo ícones, permitindo determinar títulos e textos e personalizando o botão. Ainda é possível determinar quais componentes irá conter e em que área da tela irá aparecer.

O `JOptionPane` oferece as seguintes opções de ícone:

interrogação	informação	advertência	erro
			

Para a criação de forma mais simples de diálogos pode-se usar os métodos `showXXXDialog` da classe `JOptionPane`. Os métodos mais usados são `showMessageDialog` (dialog com um único botão) e `showOptionDialog` (pode conter uma variedade de botões personalizados, texto e outros componentes).

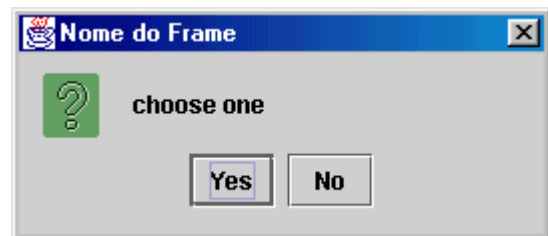
Como opção de métodos para criar dialog box temos:

- **showMessageDialog:** Exibe uma dialog box modal com um único botão OK. É possível especificar a mensagem, o ícone e o título da dialog box.

- **showOptionDialog:** Cria uma dialog box com o texto, ícone, botões e título especificados. Com esse método é possível especificar os textos dos botões e outros tipos de personalização.

O programa a seguir cria uma dialog box com os botões de opção YES e NO.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class JOptionPaneTest {
    public static void main (String args []) {
        JFrame f = new JFrame ("Meu Frame");
        JButton b = new JButton ("Botão");
        f.getContentPane().setLayout (new FlowLayout ());
        f.getContentPane().add (b);
        Handler handler = new Handler (f);
        b.addActionListener (handler);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        f.setSize (200, 100);
        f.show ();
    }
}
class Handler implements ActionListener {
    JFrame f;
    public Handler (JFrame f) {
        this.f = f;
    }
    public void actionPerformed (ActionEvent e) {
        int ret = JOptionPane.showConfirmDialog(f,
            "choose one", "Nome do Frame",
            JOptionPane.YES_NO_OPTION);
        System.out.println ("Opcao escolhida = " + ret);
    }
}
```



## 10.14 JDialog

Para criar um JDialog deve-se criar uma classe que seja subclasse de JDialog. É o mesmo processo que derivar uma classe de um JFrame. Você deve:

- No construtor da classe chamar o construtor da classe pai JDialog. É preciso fornecer a instância do frame, o título da caixa de diálogo e um flag booleano para indicar se é modal ou não.
- Adicione os componentes desejados ao dialog box.
- Adicione os handler de eventos.
- Determine o tamanho da dialog box.

O programa abaixo, retirado de (Horstman & Cornell, 2000), exibe uma caixa de diálogo JDialog quando o usuário escolhe a opção About do menu.

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

class DialogFrame extends JFrame
    implements ActionListener
{
    public DialogFrame()
    {
        setTitle("DialogTest");
        setSize(300, 300);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}
```

```

    }
} );

JMenuBar mbar = new JMenuBar();
setJMenuBar(mbar);
JMenu fileMenu = new JMenu("File");
mbar.add(fileMenu);
aboutItem = new JMenuItem("About");
aboutItem.addActionListener(this);
fileMenu.add(aboutItem);
exitItem = new JMenuItem("Exit");
exitItem.addActionListener(this);
fileMenu.add(exitItem);
}

public void actionPerformed(ActionEvent evt)
{
    Object source = evt.getSource();
    if(source == aboutItem)
    {
        if (dialog == null) // first time
            dialog = new AboutDialog(this);
        dialog.show();
    }
    else if(source == exitItem)
    {
        System.exit(0);
    }
}

private AboutDialog dialog;
private JMenuItem aboutItem;
private JMenuItem exitItem;
}

class AboutDialog extends JDialog
{
    public AboutDialog(JFrame parent)
    {
        super(parent, "About DialogTest", true);

        Box b = Box.createVerticalBox();
        b.add(Box.createGlue());
        b.add(new JLabel("Core Java"));
        b.add(new JLabel("By Cay Horstmann and Gary Cornell"));
        b.add(Box.createGlue());
        getContentPane().add(b, "Center");

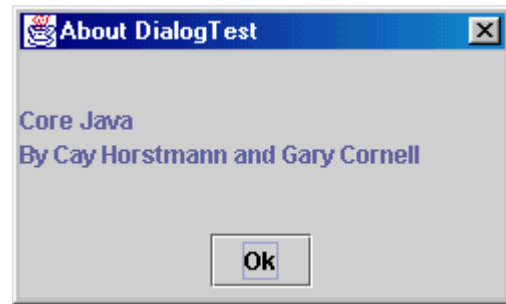
        JPanel p2 = new JPanel();
        JButton ok = new JButton("Ok");
        p2.add(ok);
        getContentPane().add(p2, "South");

        ok.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent evt)
            {
                setVisible(false);
            }
        });

        setSize(250, 150);
    }
}

public class DialogTest {
    public static void main(String[] args)
    {
        JFrame f = new DialogFrame();
        f.show();
    }
}

```





## 10.15 JFileChooser

O JFileChooser fornece uma janela para navegação pelo sistema de arquivos. Ele permite escolher um arquivo para abrir ou especificar o diretório onde um arquivo será salvo. JFileChooser são sempre dialog box modais. Esta classe equivale ao FileDialog da biblioteca AWT.

O programa abaixo, retirado de (Campione, 2000), exibe uma caixa de diálogo JFileChooser.

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

public class FileChooserDemo extends JFrame {
    static private final String newline = System.getProperty("line.separator");

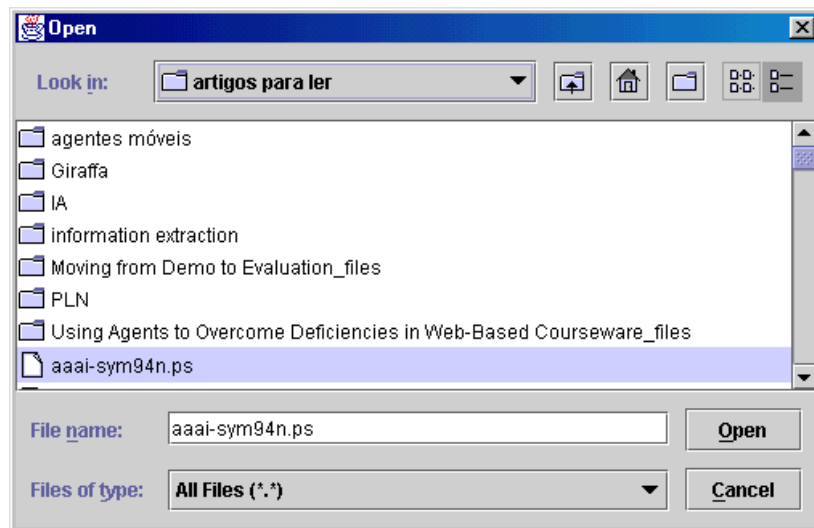
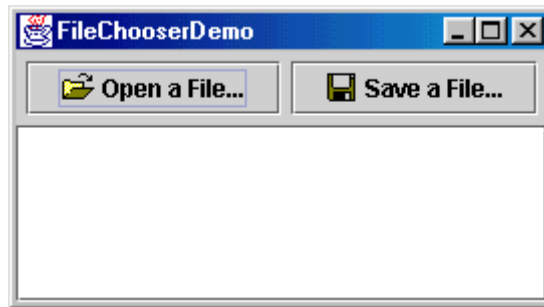
    public FileChooserDemo() {
        super("FileChooserDemo");
        final JTextArea log = new JTextArea(5,20);
        log.setMargin(new Insets(5,5,5,5));
        JScrollPane logScrollPane = new JScrollPane(log);
        //Create a file chooser
        final JFileChooser fc = new JFileChooser();
        //Create the open button
        ImageIcon openIcon = new ImageIcon("open.gif");
        JButton openButton = new JButton("Open a File...", openIcon);
        openButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int returnVal = fc.showOpenDialog(FileChooserDemo.this);
                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    File file = fc.getSelectedFile();
                    //this is where a real application would open the file.
                    log.append("Opening: " + file.getName() + "." + newline);
                } else {
                    log.append("Open command cancelled by user." + newline);
                }
            }
        });
        //Create the save button
        ImageIcon saveIcon = new ImageIcon("save.gif");
        JButton saveButton = new JButton("Save a File...", saveIcon);
        saveButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int returnVal = fc.showSaveDialog(FileChooserDemo.this);
                // opcao retornada pelo usuario -> escolheu == APPROVE_OPTION
                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    File file = fc.getSelectedFile();
                    //this is where a real application would save the file.
                    log.append("Saving: " + file.getName() + "." + newline);
                } else {
                    log.append("Save command cancelled by user." + newline);
                }
            }
        });
        //For layout purposes, put the buttons in a separate panel
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(openButton);
        buttonPanel.add(saveButton);
        //Add the buttons and the log to the frame
        Container contentPane = getContentPane();
        contentPane.add(buttonPanel, BorderLayout.NORTH);
        contentPane.add(logScrollPane, BorderLayout.CENTER);
    }

    public static void main(String s[]) {
        JFrame frame = new FileChooserDemo();
    }
}
```

```

frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {System.exit(0);}
});
frame.pack();
frame.setVisible(true);
}
}

```



## 10.16 Menu

O menu em Swing funciona de maneira análoga ao menu da biblioteca AWT (ver seção menu do capítulo de Interface Gráfica AWT desta apostila).

Primeiramente, é necessário criar uma barra de menu.

```
JMenuBar menuBar = new JMenuBar ( );
```

Para adicionar a barra de menu no topo do frame, use o método `setJMenuBar`:

```
frame.setJMenuBar (menuBar);
```

Para cada menu, você deve criar um objeto de menu:

```
JMenu editMenu = new JMenu ("Edit");
```

Após, deve-se adicionar itens de menu, separadores e submenus ao objeto menu:

```
JMenuItem pasteItem = new JMenuItem ("Paste");
editMenu.add (pasteItem);
```

```
editMenu.addSeparator ( );
JMenu optionsMenu = new JMenu ("Options"); // a submenu
editMenu.add (optionMenu);
```

Por último, temos que adicionar os menus a barra de menu:

```
menuBar.addMenu (editMenu);
```

O exemplo abaixo, retirado de (Horstman & Cornell, 2000), exibe uma barra de menu com menus e submenus.

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

public class MenuTest extends JFrame implements ActionListener, MenuListener
{
    public MenuTest()
    {
        setTitle("MenuTest");
        setSize(400, 300);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });

        JMenuBar mbar = new JMenuBar();
        setJMenuBar(mbar);
        // demonstrate enabled/disabled items
        JMenu fileMenu = new JMenu("File");
        fileMenu.addMenuListener(this);
        // demonstrate accelerators
        JMenuItem openItem = new JMenuItem("Open");
        openItem.setAccelerator
            (KeyStroke.getKeyStroke(KeyEvent.VK_O, InputEvent.CTRL_MASK));
        saveItem = new JMenuItem("Save");
        saveItem.setAccelerator
            (KeyStroke.getKeyStroke(KeyEvent.VK_S, InputEvent.CTRL_MASK));
        saveAsItem = new JMenuItem("Save As");
        mbar.add(makeMenu(fileMenu,
            new Object[] {"New", openItem, null, saveItem, saveAsItem, null, "Exit" },
            this));
        // demonstrate check box and radio button menus
        readonlyItem = new JCheckBoxMenuItem("Read-only");
        ButtonGroup group = new ButtonGroup();
        JRadioButtonMenuItem insertItem
            = new JRadioButtonMenuItem("Insert");
        insertItem.setSelected(true);
        JRadioButtonMenuItem overtypeItem
            = new JRadioButtonMenuItem("Overtyping");
        group.add(insertItem);
        group.add(overtypingItem);
        // demonstrate icons and nested menus
        mbar.add(makeMenu("Edit",
            new Object[]
            {
                new JMenuItem("Cut",
                    new ImageIcon("cut.gif")),
                new JMenuItem("Copy",
                    new ImageIcon("copy.gif")),
                new JMenuItem("Paste",
                    new ImageIcon("paste.gif")),
                null,
                makeMenu("Options",
                    new Object[]
                    {
                        readonlyItem,
                        null,
                        insertItem,
                        overtypingItem
                    }
                )
            }
        ));
    }

    public void actionPerformed(ActionEvent e)
    {
        // ...
    }

    public void menuSelected(MenuEvent e)
    {
        // ...
    }

    public void menuWillBecomeVisible(MenuEvent e)
    {
        // ...
    }

    public void menuWillBecomeDisabled(MenuEvent e)
    {
        // ...
    }

    public void menuIsDisabled(MenuEvent e)
    {
        // ...
    }
}
```

```

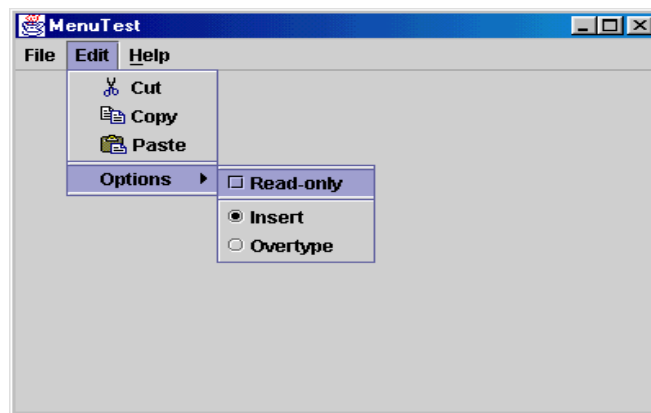
        },
        this)
    },
    this));
// demonstrate mnemonics
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
mbar.add(makeMenu(helpMenu,
    new Object[]
    { new JMenuItem("Index", 'I'),
      new JMenuItem("About", 'A')
    },
    this));
// demonstrate pop-ups
popup = makePopupMenu(new Object[] { "Cut", "Copy", "Paste" }, this);
getContentPane().addMouseListener(new MouseAdapter()
{ public void mouseReleased(MouseEvent evt)
  { if (evt.isPopupTrigger())
    popup.show(evt.getComponent(),
              evt.getX(), evt.getY()); } });
}
public void actionPerformed(ActionEvent evt)
{ String arg = evt.getActionCommand();
  System.out.println(arg);
  if(arg.equals("Exit"))
    System.exit(0);
}
public void menuSelected(MenuEvent evt)
{ saveItem.setEnabled(!readonlyItem.isSelected());
  saveAsItem.setEnabled(!readonlyItem.isSelected());
}
public void menuDeselected(MenuEvent evt) { }
public void menuCanceled(MenuEvent evt) { }
public static JMenu makeMenu(Object parent,
    Object[] items, Object target)
{ JMenu m = null;
  if (parent instanceof JMenu)
    m = (JMenu)parent;
  else if (parent instanceof String)
    m = new JMenu((String)parent);
  else
    return null;
  for (int i = 0; i < items.length; i++)
  { if (items[i] == null)
    m.addSeparator();
    else
      m.add(makeMenuItem(items[i], target));
  }
  return m;
}
public static JMenuItem makeMenuItem(Object item,
    Object target)
{ JMenuItem r = null;
  if (item instanceof String)
    r = new JMenuItem((String)item);
  else if (item instanceof JMenuItem)
    r = (JMenuItem)item;
  else return null;
  if (target instanceof ActionListener)
    r.addActionListener((ActionListener)target);
  return r;
}
public static JPopupMenu makePopupMenu
    (Object[] items, Object target)
{ JPopupMenu m = new JPopupMenu();

```

```

    for (int i = 0; i < items.length; i++)
    { if (items[i] == null)
      m.addSeparator();
      else
        m.add(makeMenuItem(items[i], target));
    }
    return m;
}
public static void main(String[] args)
{ Frame f = new MenuTest();
  f.show();
}
private JMenuItem saveItem;
private JMenuItem saveAsItem;
private JCheckBoxMenuItem readonlyItem;
private JPopupMenu popup;
}

```



## 10.17 JApplet

Se o seu applet irá conter elementos de interface gráfica do pacote swing, ele deverá ser um JApplet, caso contrário não funcionará corretamente. Um JApplet é subclasse da classe Applet. O ciclo de vida de um JApplet é análogo a de um Applet, por isso, não será revisado neste capítulo. Para informações básicas sobre o funcionamento de um applet, veja a seção de Applet desta apostila.

Em applets swing, temos que adicionar os componentes de interface gráfica dentro do content pane do applet, assim como fazemos em um JFrame.

Veja abaixo um exemplo simples de um applet swing, retirado de (Campione, 2000), que exibe uma mensagem em um label no browser.

```

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

public class HelloSwingApplet extends JApplet {
    public void init() {
        JLabel label = new JLabel(
            "You are successfully running a Swing applet!");
        label.setHorizontalAlignment(JLabel.CENTER);
        label.setBorder(BorderFactory.createMatteBorder(1,1,2,2,Color.black));
        getContentPane().add(label, BorderLayout.CENTER);
    }
}

```

Se o seu browser não suporta o JDK1.2 ou superior você pode instalar o Java Plug-in. O Java Plug-in permite selecionar a versão de JDK desejada. Porém, para rodar um applet através do Java Plug-in é necessária que a página HTML contenha mais do que uma simples tag <applet>, como vista no capítulo de Applet. Para gerar esta página crie uma página HTML com a tag <applet> e execute Java Plug-in HTML converter<sup>1</sup> para gerar a nova página HTML a partir da anterior. Uma maneira mais simples é usar o appletviewer para visualizar os applets. A página HTML para exibir este applet em um browser com Java Plug-in fica:

```
<html>
<head>
<title>Run HelloSwingApplet</title>
</head>
<body BGCOLOR="#ffffff">
<h2>
    Run HelloSwingApplet
</h2>
<p>
<blockquote>
<!--"CONVERTED_APPLET"-->
<!-- CONVERTER VERSION 1.1 -->
<SCRIPT LANGUAGE="JavaScript"><!--
    var _info = navigator.userAgent; var _ns = false;
    var _ie = (_info.indexOf("MSIE") > 0 && _info.indexOf("Win") > 0 &&
_info.indexOf("Windows 3.1") < 0);
//--></SCRIPT>
<COMMENT><SCRIPT LANGUAGE="JavaScript1.1"><!--
    var _ns = (navigator.appName.indexOf("Netscape") >= 0 && ((_info.indexOf("Win")
> 0 && _info.indexOf("Win16") < 0 &&
java.lang.System.getProperty("os.version").indexOf("3.5") < 0) ||
(_info.indexOf("Sun") > 0) || (_info.indexOf("Linux") > 0)));
//--></SCRIPT></COMMENT>

<SCRIPT LANGUAGE="JavaScript"><!--
    if (_ie == true) document.writeln('<OBJECT classid="clsid:8AD9C840-044E-11D1-
B3E9-00805F499D93" WIDTH = 400 HEIGHT = 50
codebase="http://java.sun.com/products/plugin/1.1.2/jinstall-112-
win32.cab#Version=1,1,2,0"><NOEMBED><XMP>');
    else if (_ns == true) document.writeln('<EMBED type="application/x-java-
applet;version=1.1.2" java_CODE = "HelloSwingApplet.class" java_CODEBASE = "example-
swing" java_ARCHIVE = "applets.jar" WIDTH = 400 HEIGHT = 50
pluginspage="http://java.sun.com/products/plugin/1.1.2/plugin-
install.html"><NOEMBED><XMP>');
//--></SCRIPT>
<APPLET CODE = "HelloSwingApplet.class" CODEBASE = "example-swing" ARCHIVE =
"applets.jar" WIDTH = 400 HEIGHT = 50 ></XMP>
<PARAM NAME = CODE VALUE = "HelloSwingApplet.class" >
<PARAM NAME = CODEBASE VALUE = "example-swing" >
<PARAM NAME = ARCHIVE VALUE = "applets.jar" >

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1.2">

</APPLET>

</NOEMBED></EMBED></OBJECT>
<!--
<APPLET CODE = "HelloSwingApplet.class" CODEBASE = "example-swing" ARCHIVE =
"applets.jar" WIDTH = 400 HEIGHT = 50 >
```

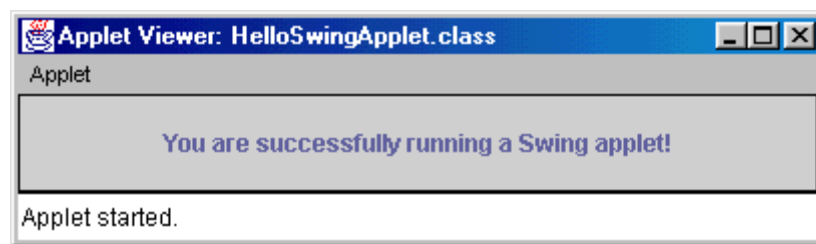
---

<sup>1</sup> Veja como obter o Java Plug-in HTML Converter na URL <http://java.sun.com/products/plugin>.

```

</APPLET>
-->
<!--"END_CONVERTED_APPLET"-->
<blockquote><hr><strong>Note:</strong> Because the preceding applet uses the Swing
1.1 API, it requires Java Plug-in 1.1.2 or Java Plug-in 1.2. It won't work with Java
Plug-in 1.1.1 or uncustomized 1.1 browsers. For more information about running
applets in the tutorial, refer to <a href="../../../information/examples.html">About
Our Examples</a>. That page includes a section about <a
href="../../../information/examples.html#plugin">Downloading Java Plug-
in</a>.<hr></blockquote>
</blockquote>
<p>
<hr size=4>
<p>
</body>
</html>

```



### 10.17.1 Applet Context

Um applet executa dentro de um browser ou dentro de um appletviewer. Um applet pode pedir para um browser fazer coisas como tocar um música, mostrar uma página web, etc. Cabe ao browser decidir fazer.

A comunicação com o browser é feita através da chamada do método `getAppletContext`. Este método retorna um objeto que implementa uma interface do tipo `AppletContext`. Por exemplo, podemos usar este objeto para mostrar uma mensagem na linha de status do browser através do método:

```
showStatus ("Loading data... please wait!");
```

Ele também pode ser usado para mostrar uma página web diferente. Por exemplo:

```
URL u = new URL ("http://java.sun.com");
getAppletContext( ).showDocument (u);
```

Este comando irá exibir a página HTML na mesma janela de browser que a página corrente, substituindo o applet. Para abrir o documento em uma nova janela, use:

```
getAppletContext( ).showDocument (u, "_blank");
```

O programa abaixo, retirado de (Horstman & Cornell, 2000), exibe um applet que implementa um bookmark.

```

import java.awt.*;
import java.applet.*;
import java.util.*;
import java.net.*;
import javax.swing.*;
import javax.swing.event.*;

public class Bookmark extends JApplet
    implements ListSelectionListener
{
    public void init()

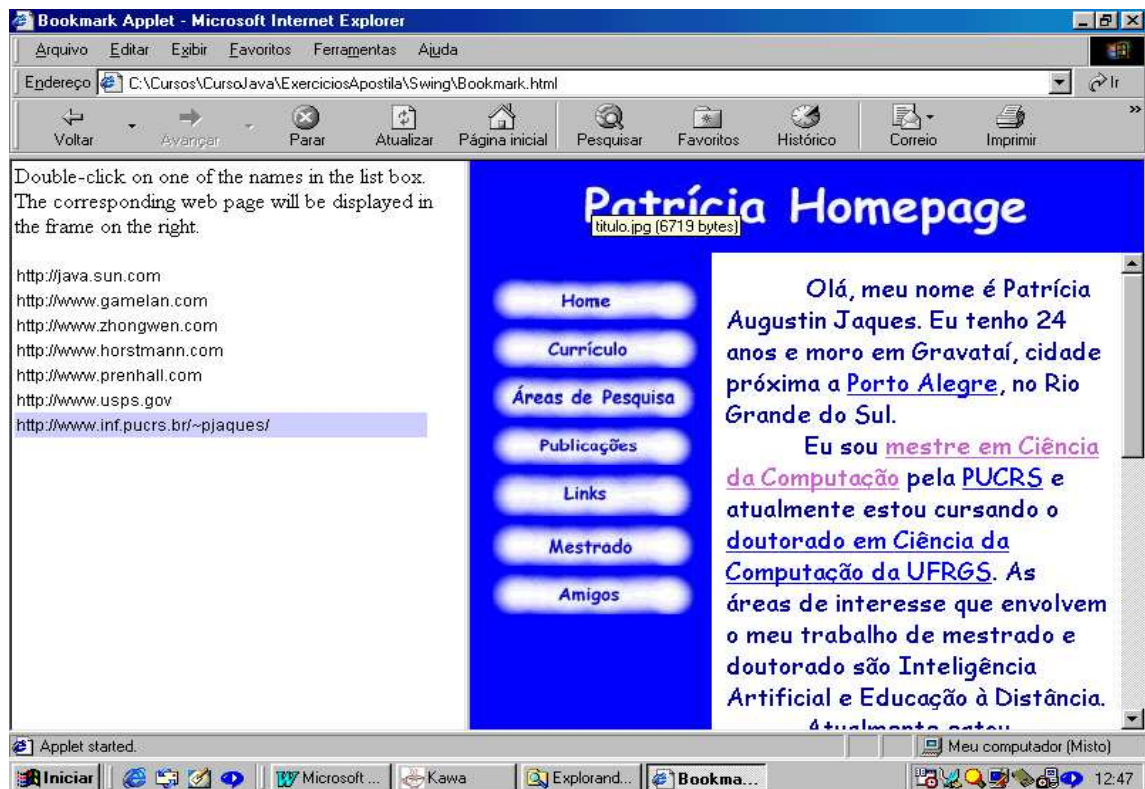
```

```

{
    int i = 1;
    String s;
    Vector v = new Vector();
    while ((s = getParameter("link_" + i)) != null)
    {
        v.add(s);
        i++;
    }
    JList links = new JList(v);
    Container contentPane = getContentPane();
    contentPane.add(links);
    links.addListSelectionListener(this);
}

public void valueChanged(ListSelectionEvent evt)
{
    if (evt.getValueIsAdjusting()) return;
    JList source = (JList)evt.getSource();
    String arg = (String)source.getSelectedValue();
    try
    {
        AppletContext context = getAppletContext();
        URL u = new URL(arg);
        context.showDocument(u, "right");
    } catch (Exception e)
    {
        showStatus("Error " + e);
    }
}
}

```



Página Bookmark.html (Principal)

```

<HTML>
<HEAD>
<TITLE>Bookmark Applet</TITLE>
</HEAD>
<FRAMESET COLS="320,*">
<FRAME NAME="left" SRC="Left.html"
    MARGINHEIGHT=2 MARGINWIDTH=2

```



```
    SCROLLING = "no" NORESIZE>
<FRAME NAME="right" SRC="Right.html"
    MARGINHEIGHT=2 MARGINWIDTH=2
    SCROLLING = "yes" NORESIZE>
</FRAMESET>
</HTML>
```

#### Página Left.html

```
<HTML>
<TITLE>A Bookmark Applet</TITLE>
<BODY>
Double-click on one of the names in the list box.
The corresponding web page
will be displayed in the frame on the right.
<APPLET CODE = "Bookmark.class" WIDTH = 290 HEIGHT = 300 >
<PARAM NAME = link_1 VALUE = "http://java.sun.com">
<PARAM NAME = link_2 VALUE = "http://www.gamelan.com">
<PARAM NAME = link_3 VALUE = "http://www.zhongwen.com">
<PARAM NAME = link_4 VALUE = "http://www.horstmann.com">
<PARAM NAME = link_5 VALUE = "http://www.prenhall.com">
<PARAM NAME = link_6 VALUE = "http://www.usps.gov">
<PARAM NAME = link_7 VALUE = "http://www.inf.pucrs.br/~pjaques/">
</APPLET>
</BODY>
</HTML>
```

#### Página Right.html

```
<HTML>
<TITLE>
Web pages will be displayed here.
</TITLE>
<BODY>
Double-click on one of the names in the list box to the left.
The web page will be displayed here.
</BODY>
</HTML>
```

## 11 Endereços Úteis na Internet

<http://java.sun.com> - Kit para desenvolvimento de aplicativos Java montado pelos criadores da linguagem. Sob este endereço você pode obter o compilador e outras ferramentas de desenvolvimento de aplicações Java para a sua plataforma de programação.

<http://java.sun.com/docs/books/tutorial/index.html> - Tutorial da linguagem Java, implementado pela Sun.

<http://java.sun.com/docs/books/vmspec/index.html> - Informações sobre a Java Virtual Machine.

<http://java.sun.com/docs/index.html> - Links para documentações da Plataforma Java.

<http://java.sun.com/products/jdbc/index.html> - Informações sobre JDBC.

<http://industry.java.sun.com/products/jdbc/drivers> - Lista de drivers JDBC disponíveis.

<http://www.gamelan.com> - Contem vários aplicativos Java e recursos para programadores.

<http://www.javaworld.com> - Revista eletrônica da linguagem java.

<http://www.javasoft.com/applets> - Links para vários *applets*, divididos por categorias: com jogos, animação, etc.

## 12 Referências Bibliográficas

- (Albuquerque, 1999) Albuquerque, Fernando. **Programação Orientada a Objetos usando Java e UML**. Brasília: MSD, 1999.
- (Booch, 1994) Booch, Grady. **Object Oriented Design with Applications**. Redwood City: The Benjamin/Cummings Publishing Company, 1994.
- (Campione, 2000) Campione Mary, Walrath Kathy. **The Java Tutorial, Object-Oriented Programming for the Internet**. <http://www.aw.com/cp/javaseries.html>.
- (Chan et al., 1999) Chan, Mark; Griffith, Steven & Iasi, Anthony. **Java 1001 Dicas de Programação**. São Paulo: Makron Books, 1999.
- (Coad & Yourdon, 1993) Coad, Peter & Yourdon, Edward. **Projeto Baseado em Objetos**. Rio de Janeiro: Campus, 1993.
- (Deitel, 2001) Deitel, Harvey M. Java como Programar. Porto Alegre: Bookman, 2001.
- (Flanagan, 1999) Flanagan, David. **Java in a Nutshell – A Desktop Quick Reference** (Java 2SDK 1.2 e 1.3). 3<sup>rd</sup> Edition. Sebastopol: O'Reilly, 1999.
- (Hamilton et al., 1997) Hamilton, Graham; Cattell, Rick & Fischer, Maydene. **JDBC Database Access with Java**. Massachusetts: Addison Wesley, 1997.
- (Harold, 2000) Harold, Elliotte Rusty. **Brewing Java: A Tutorial**. <http://sunsite.unc.edu/javafaq/javatutorial.html>.
- (Harold, 1997) Harold, Elliotte Rusty. **Java Network Programming**. Sebastopol: O'Reilly, 1997.
- (Horstman & Cornell, 2000) Horstmann, Cay & Cornell, Gary. **Core Java 2 - Volume 1 – Fundamentos**. Makron Books.
- (Horstman & Cornell, 2000) Horstmann, Cay & Cornell, Gary. **Core Java 2 - Volume 2 – Avançados**. Makron Books.
- (Hunter & Crawford, 1998) Hunter, Jason & Crawford, William. **Java Servlet Programming**. Sebastopol: O'Reilly, 1998. <http://www.ora.com/catalog/jservlet/>.
- (Koffman & Wolz, 1999) Koffman, Elliot & Wolz, Ursula. **Problems Solving with Java**. USA: Addison-Wesley, 1999.
- (Koosis & Koosis, 1999) Koosis, Donald & Koosis, David. **Programação com Java – Série para Dummies**. Rio de Janeiro: Campus, 1999.
- (Lalani et al., 1997) Lalani, Suleiman San; Jamsa, Kris. **Java Biblioteca do Programador**. São Paulo: Makron Books, 1997.
- (Lemay & Perkins, 1996) Lemay, Laura & Perkins Charles L. **Teach Yourself JAVA in 21 days**. Samsnet, 1996.
- (Lindholm et al., 1997) Lindholm, Tim; Yellin, Frank; Friendly, Lisa. **The Java Virtual Machine Specification**. Reading: Addison-Wesley, 1997. 475 p.
- (Morgan, 1998) Morgan, Mike. **Using Java 1.2**. Indianapolis: QUE, 1998.
- (Naughton, 1996) Naughton, Patrick. **Dominando o Java**. São Paulo: Makron Books, 1996.
- (Oaks & Wong, 1997) Oaks, Scott & Wong, Henry. **Java Threads**. Cambridge: O'Reilly, 1997.
- (Reese, 2000) Reese, George. **Database Programming with JDBC and Java**. <http://www.ora.com/catalog/javadata/>
- (Ritchey, 1996) Ritchey, Tim. **Programando com Java! Beta 2.0**. Rio de Janeiro: Campus, 1996.
- (Santos, 2003) Santos, Rafael. **Introdução à Programação Orientada a Objetos Usando Java**. São Paulo: SBC, Campus, 2003.
- (Sun, 1997) SUN Microsystems. **Java Programming SL-276 Student Guide**. June 1997.

(Venners, 2000) Venners, Bill. **Inside the Java Virtual Machine**. McGraw-Hill, 2000. Livro pode ser parcialmente consultado em: <http://www.artima.com/insidejvm/blurb.html>.