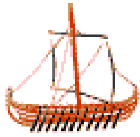


ODYSSEY **Ambiente de Reutilização de Software Baseado em Modelos de Domínio**

Financiamento CNPq

CAFÉ DA MANHÃ COM JAVA

Leonardo Gresta Paulino Murta
Cláudia Maria Lima Werner
Márcio de Oliveira Barros

novembro 1998

PUBLICAÇÕES TÉCNICAS 3/98
COPPE/UFRJ
Caixa Postal 68511 - CEP 21945-970 - Rio de Janeiro - RJ

Índice

1 - Introdução ... 1
- **Java: Como Surgiu? De Onde Veio? O que é? ... 1**
  - A sua História ... 1
  - As suas Características ... 1
- **Java x C++ ... 2**
  - Os 3 Alicerces ... 2
  - Main() e Environment ... 2
  - Ponteiros ... 3
  - Principais diferenças ... 3
- **JVM (Java Virtual Machine) ... 4**
- **Ambientes de Desenvolvimento ... 5**
- **Introdução ao JDK (Java Development Kit) ... 5**
  - Javac ... 5
  - Java ... 6
  - Jre ... 6
  - Jdb ... 7
  - Javah ... 7
  - Javap ... 7
  - JavaDoc ... 7
  - AppletViewer ... 8
- **Aplicação x Applet ... 8**

2 - Orientação a Objetos em Java ... 10
- **Classes ... 10**
  - Estrutura ... 10
  - Instanciação de classes ... 11
- **Interfaces ... 12**
- **Métodos ... 13**
  - Estrutura, Declaração e Utilização ... 13
  - Sobrecarga ... 13
- **Atributos ... 14**
- **Modificadores ... 14**
  - Visibilidade ... 14
  - Static ... 15
- **Pacotes ... 15**

3 - A Linguagem Java ... 16
- **Tipos Primitivos ... 16**
- **Arrays ... 17**
- **Operadores ... 18**
- **Repetição e Decisão ... 19**
  - If-else ... 20
  - Switch-case ... 20
  - For ... 21
  - While ... 22
  - Do-while ... 22
  - Try-catch-finally ... 22

Continue e break ... 23
4 - Pacotes Essenciais do JDK 1.1 ... 24
Applet ... 24
AWT ... 25
Button ... 25
Canvas ... 25
Checkbox ... 25
Dialog ... 25
Frame ... 26
Label ... 26
Layouts ... 26
List ... 26
Menu ... 27
Panel ... 27
TextArea ... 27
TextField ... 27
Lang ... 27
Math ... 27
String ... 28
Util ... 29
Hashtable ... 30
Vector ... 30
Stack ... 31
5 - Tópicos Básicos ... 32
Eventos ... 32
Exceções ... 32
Serialização ... 33
Threads ... 34
Comentários de Código ... 35
6 - Tópicos Avançados ... 37
IDL (Interface Definition Language) ... 37
Internationalization ... 37
JAR (Java Archive) ... 37
Java Beans ... 38
Java Media ... 38
JDBC (Java Database Connectivity) ... 38
Security ... 39
Servlets ... 39
RMI (Remote Method Invocation) ... 39
Bibliografia ... 40
Índice Remissivo ... 1

1 - Introdução

Java: Como Surgiu? De Onde Veio? O que é?

A sua História

Em torno de 1990, uma equipe da SUN Microsystems, liderada por James Gosling foi incumbida de desenvolver programas para controlar aparelhos eletrônicos. A linguagem inicialmente utilizada foi C++, mas, com o decorrer do tempo, essa equipe se deparou com várias dificuldades inerentes ao C++ (tais como herança múltipla e ponteiros). Para contornar essas dificuldades foi desenvolvida a linguagem Oak, que tinha a mesma estrutura do C++, só que sem os detalhes que o tornava complicado para controlar aparelhos eletrônicos.

A linguagem Oak foi utilizada na construção do sistema Star Seven (que possibilita o controle de vários aparelhos eletrônicos de uma casa através do toque em uma tela) e de um sistema de televisão interativa. Apesar de nenhum dos dois projetos terem se tornado produtos, a linguagem Oak, com o decorrer do tempo, pôde amadurecer. Tendo em vista que o nome Oak já havia sido reivindicado, a SUN passou a chamar a linguagem Oak de Java.

Em 1993, a Internet passou a suportar WWW, deixando de ser composta somente de texto e adotando um ambiente gráfico. Com isso, a SUN adaptou a já poderosa (independente de plataforma, segura e simples) linguagem Java para esse novo mundo, criando o recurso de Applet. Para mostrar que Java era uma linguagem robusta, a SUN implementou um browser, o HotJava.

Em meados de 1995 a linguagem Java foi lançada oficialmente, e passou a ser incorporada aos browsers da Netscape e da Microsoft.

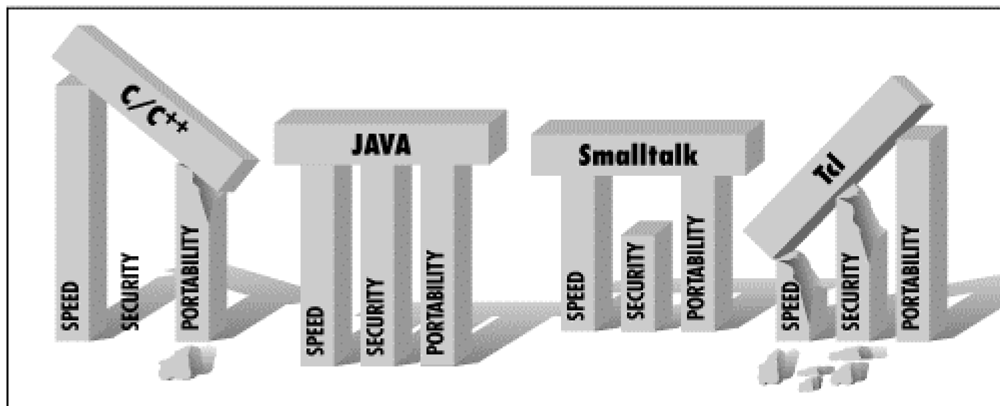
As suas Características

- **Simples:** Java eliminou as dificuldades herdadas do C++ e adicionou facilidades, tal como *Garbage Collector*;
- **Orientada a Objetos:** Java é pura (totalmente OO) e agrega um grande conjunto de classes em suas bibliotecas;
- **Distribuída:** Suporta aplicações em rede e objetos distribuídos;
- **Segura:** Verifica o *byte-code* antes da execução;
- **Robusta:** O interpretador permite tratamento de exceções e não permite que uma aplicação paralise o sistema (projetada para software confiável);
- **Plataforma Independente:** Java, por ser interpretada, pode rodar em qualquer plataforma (desde que esta tenha o interpretador);
- **Alta Performance:** Mais rápida que linguagens *script* (em browsers) e passível de compilação *just-in-time* (JIT);
- **Multi-tarefa:** Pode-se executar *threads* simultaneamente;

Java x C++

Os 3 Alicerces

Tradicionalmente, as linguagens de programação são avaliadas por três parâmetros: velocidade, segurança e portabilidade. O ideal é uma linguagem que tenha esses três alicerces alinhados, pois a deficiência em algum desses itens pode comprometer o sucesso do software que está sendo desenvolvido. Abaixo, é apresentado graficamente o peso de cada um dos três alicerces nas linguagens C++, Java, Smalltalk e Tcl:



Fonte: Niemeyer; Peck; 1997.

Main() e Environment

Tanto em Java quanto em C++ existe o método *main()*, que tem a seguinte sintaxe (em Java):

```
public static void main(String args[])
```

No C++, a passagem de parâmetros para o *main()* é através de *argc* e *argv*, o que se torna desnecessário em Java, pois é possível consultar o tamanho de um *array* pelo método *length*. Java, ao contrário de C++, não considera o nome da classe como argumento.

No exemplo abaixo, temos um programa que é idêntico ao comando *echo* do UNIX:

```
public class echo
{
  public static void main(String argv[])
  {
    for(int i=0; i < argv.length; i++)
      System.out.print(argv[i] + " ");
    System.out.print("\n");
    System.exit(0);
  }
}
```

Fonte: Flanagan; 1997.

Para terminar a execução de um programa deve ser chamado o método *exit()* da classe *System*, passando o código de erro como argumento: System.exit(cod_erro). Caso o programa não retorne nada, é necessária a presença do *void* antes do *main()*.

Como Java é independente de plataforma, não é possível o uso de *environment* para comunicação. A solução é o uso de propriedades passadas para o interpretador através do *flag* -D, como por exemplo:

```
java -Dmyapp.debug=true myapp
```

Para recuperar a propriedade passada, deve-se executar o método *getProperty()* da classe *System*. Por exemplo:

```
String homedir = System.getProperty("user.home");
String debug = System.getProperty("myapp.debug");
```

Ponteiros

Java, diferentemente de C++, não permite ao usuário manipular ponteiros ou endereços de memória. Não é possível fazer *cast* de objetos para *int* e nem aritmética de ponteiros.

Java trata seus tipos de dados primários por valor, e seus tipos não primários (objetos e *arrays*) por referência (ponteiros controlados pelo interpretador Java, e não pelo usuário). O valor padrão de qualquer dado não primário é *null* (palavra reservada que significa ausência de valor).

Principais diferenças

Por questões de segurança, simplificação, e por ser uma linguagem OO pura, Java não utiliza:

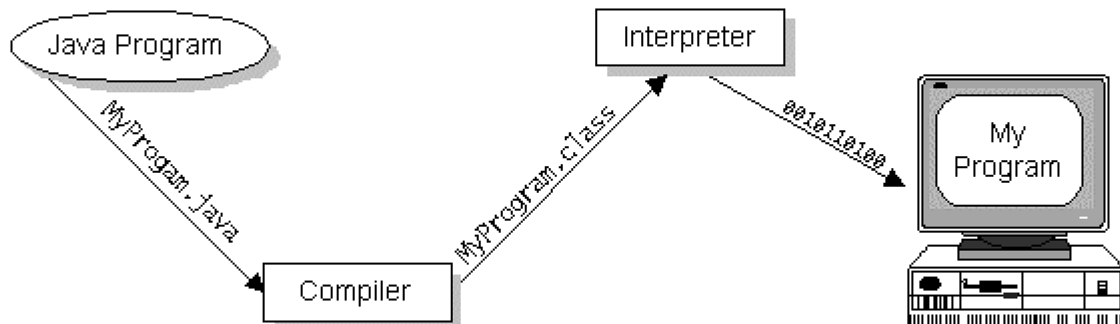
- *Structures* e *unions*;
- #define;
- Ponteiros;
- Herança múltipla;
- Funções;
- Goto;
- Sobrecarga de operador;
- Funções amigas.

Em compensação Java adicionou:

- *Garbage collection* automático;
- Acoplamento dinâmico;
- *Write once, run everywhere*.

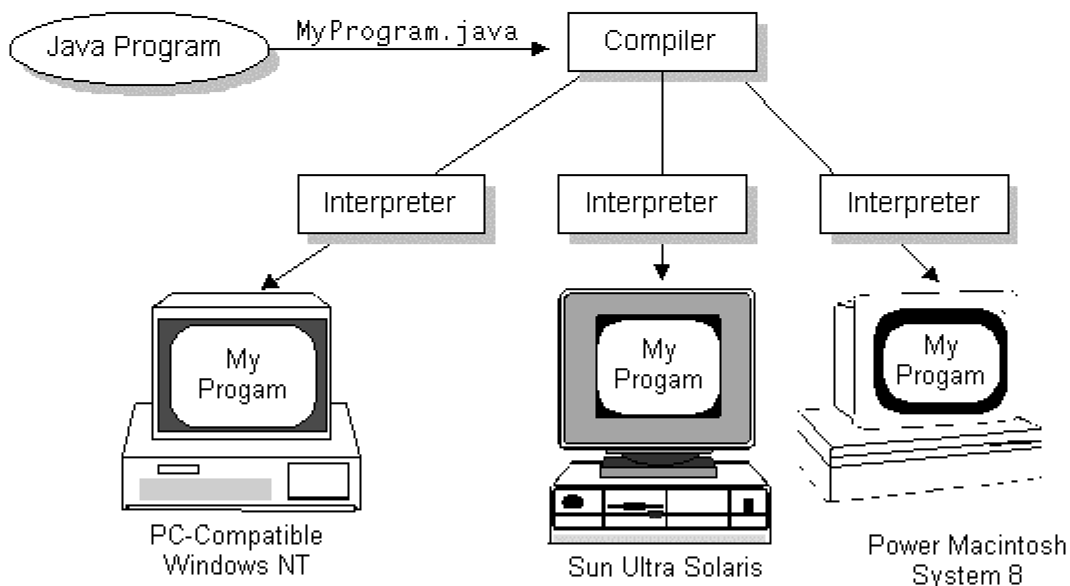
JVM (Java Virtual Machine)

Para que Java conseguisse atingir independência de plataforma, a SUN optou por uma arquitetura independente (máquina virtual), para a qual um código fonte Java é compilado. O código fonte (programa.java), após compilado, se transforma em *byte-code* (programa.class). Esse *byte-code* atualmente não pode ser executado em nenhuma arquitetura (a SUN está criando uma arquitetura que executará *byte-code*: *JavaChip*), mas pode ser interpretado em várias arquiteturas, desde que estas tenham um interpretador Java. Veja abaixo como ocorrem as mudanças <código fonte> → <*byte-code*> → <binário (plataforma dependente)>:



Fonte: Campione; Walrath; 1998.

O interpretador Java é uma implementação da máquina virtual Java, voltada para uma determinada plataforma. Um código fonte Java é compilado somente uma vez, mas será interpretado (se necessário por interpretadores implementados em plataformas diferentes) quantas vezes for necessária a sua execução.



Fonte: Campione; Walrath; 1998.

Http://www.javasoft.com/docs/white/platform/CreditsPage.doc.html

Ambientes de Desenvolvimento

Já existem, atualmente, no mercado, vários ambientes para desenvolvimento em Java. O ambiente que define os padrões é o JDK (atualmente na versão 1.1.6), fabricado pela SUN. A grande vantagem do JDK é ser *shareware*, mas ele não possui interface gráfica e nem apresenta suporte adequado à sua construção.

Outro ambiente bastante conhecido é o Jbuilder (atualmente na versão 2), fabricado pela Inprise. As suas grandes vantagens são a interface gráfica padrão Delphi/CBuilder, suporte à construção de interface gráfica e o seu depurador gráfico. Abaixo, segue uma tabela com os ambientes mais utilizados, fabricante, e onde encontrar informações úteis:

Ambiente	Fabricante	Referência
JDK	SUN	http://www.javasoft.com/products/jdk/1.1/index.html
Jbuilder	Inprise	http://www.inprise.com/jbuilder/
Visual Age	IBM	http://www7.software.ibm.com/vad.nsf
Visual J++	Microsoft	http://msdn.microsoft.com/visualj/
Visual Café	Symantec	http://www.symantec.com/domain/cafe/vc4java.html

Introdução ao JDK (Java Development Kit)

Nesta seção, apresentamos as ferramentas mais utilizadas no JDK 1.1 (versão Windows):

Javac

<table>
<tr><td>Descrição</td><td colspan="2">Compilador Java.</td></tr>
<tr><td>Função</td><td colspan="2">Transformar código fonte em byte-code.</td></tr>
<tr><td rowspan="9">Sintaxe</td><td colspan="2">javac [opções] códigofonte.java

opções mais utilizadas:</td></tr>
<tr><td>-classpath diretório</td><td>Diretório de classes utilizadas pelo programa.</td></tr>
<tr><td>-d diretório</td><td>Diretório destino para os byte-codes.</td></tr>
<tr><td>-g</td><td>Cria tabelas de debug.</td></tr>
<tr><td>-deprecation</td><td>Exibe a localização de métodos obsoletos.</td></tr>
<tr><td>-nowarn</td><td>Não exibe advertências.</td></tr>
<tr><td>-o</td><td>Tenta otimizar o byte-code a ser gerado.</td></tr>
<tr><td>-verbose</td><td>Exibe informações de status</td></tr>
<tr><td>-depend</td><td>Causa a recompilação das classes dependentes de códigofonte.java.</td></tr>
<tr><td>Exemplo</td><td colspan="2">javac Button.java</td></tr>
</table>

Http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javac.html

Java

Descrição	Interpretador Java.	
Função	Executar programas (byte-codes) escritos em Java.	
Sintaxe	**java** [*opções*] nomedaclasse <args> *opções* mais utilizadas:	
	-debug	Executa o debug.
	-classpath *diretório*	Diretório de classes utilizadas pelo programa.
	-mx*x*	Quantidade máxima de memória para a seção.
	-ms*x*	Quantidade de memória para a inicialização.
	-noasyncgc	Não utiliza coleta de lixo assíncrona.
	-noclassgc	Desliga a coleta de lixo automática para classes.
	-nojit	Desabilita a compilação just-in-time.
	-version	Exibe informações sobre versão.
	-help	Exibe informações sobre utilização.
	-ss*x*	Tamanho da pilha para código C.
	-oss*x*	Tamanho da pilha para código Java.
	-v, -verbose	Exibe informações de status.
	-verify	Verifica o código Java.
	-verifyremote	Verifica o código Java carregado de um classloader.
	-noverify	Não verifica o código Java.
	-verbosegc	Exibe informações de status do coletor de lixo.
	-D*propertyName=newValue*	Configura valores de propriedades.
Exemplo	java java.lang.String	

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/java.html

Jre

Descrição	Interpretador Java. O Jre é voltado para usuários finais, enquanto a ferramenta Java tem mais opções de desenvolvimento.
Função	Executar programas (byte-codes) escritos em Java.
Exemplo	**jre** java.lang.String

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/jre.html

Jdb

Descrição	Debug Java.
Função	Depurar programas (byte-codes) escritos em Java.
Exemplo	**jdb** sun.applet.AppletViewer

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/jdb.html

Javah

Descrição	Criador de cabeçalhos Java.
Função	Criar arquivos de cabeçalhos para uso com métodos nativos.
Exemplo	**javah** nomeClasse

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javah.html

Javap

Descrição	Desassemblador Java
Função	Transformar byte-code em código fonte Java.
Exemplo	**javap** nomeClasse

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javap.html

JavaDoc

Descrição	Documentador Java
Função	Gerar documentação HTML à partir de um código fonte Java, desde que este esteja no padrão JavaDoc (ver Comentários de Código na pág. 35).
Exemplo	**javadoc** –d C:\ws\html java.awt java.awt.event

Http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javadoc.html

AppletViewer

Descrição	Visualizador de Applets.
Função	Executar applets fora de um browser.
Exemplo	**appletviewer** página.html

http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/appletviewer.html

Para obter mais informações sobre as ferramentas descritas acima, ou sobre RMI (rmic, rmiregistry e serialver), Internacionalização (native2ascii), JAR (jar) ou Assinatura Digital (javakey), visite a seguinte página:

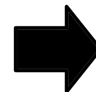
http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/index.html

Aplicação x Applet

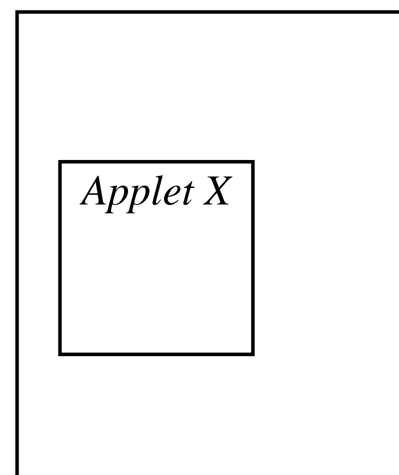
A princípio Java seria somente mais uma linguagem independente de plataforma e interpretada, mas, com o surgimento da WWW, a SUN adaptou Java para ser interpretada dentro de um browser, adicionando requisitos de segurança. Para isso foi criado o pacote Applet (ver Applet na pág. 24). Uma *tag* no HTML indica a presença de um *applet*, como ilustrado a seguir:

Página HTML

<HTML>
...
<BODY>
...
<APPLET CODE="X.class"
WIDTH=100 HEIGHT=100>
...
</APPLET>
...
</HTML>

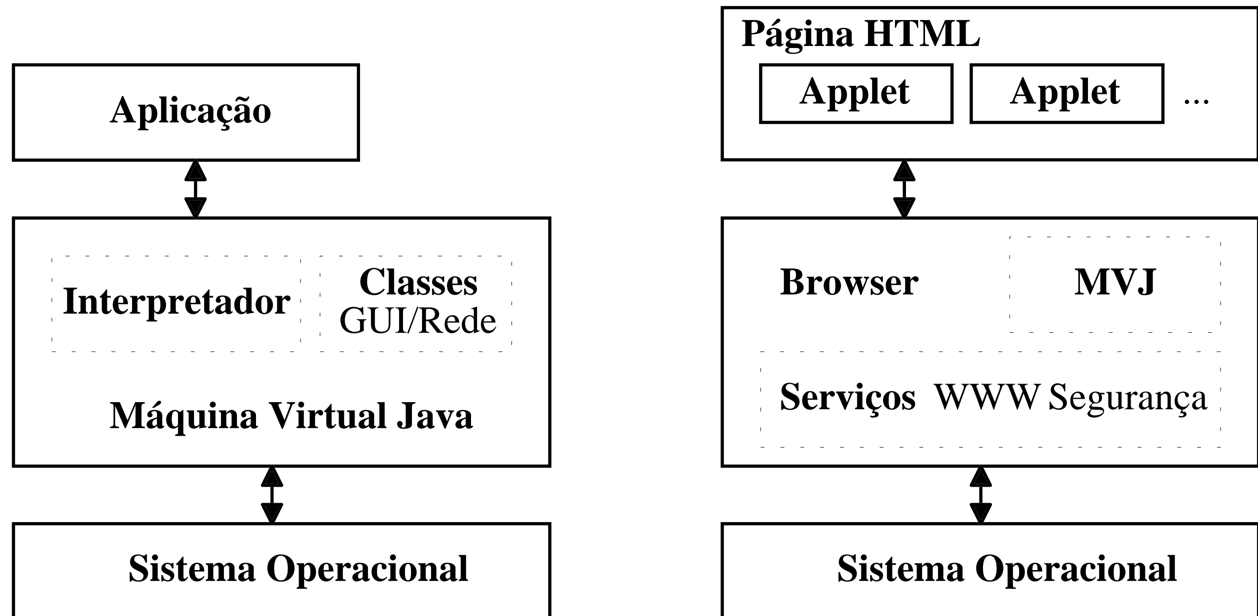


Browser



Fonte: Dias; 1998

Para que a execução desse *applet* se torne possível, o browser deve conter uma implementação da máquina virtual Java. À seguir, é ilustrada a execução de um aplicativo e de um *applet*:



Fonte: Dias; 1998

Os *applets* também podem ser executados pelo AppletViewer (ferramenta apresentada anteriormente).

2 - Orientação a Objetos em Java

Classes

Estrutura

As classes em Java tem a sua estrutura dividida em duas partes:

- Declaração da classe;
- Corpo da classe.

A declaração da classe contém toda a informação necessária para a sua identificação. Ela tem a seguinte estrutura:

```
modificadores class NomeDaClasse extends SuperClasse implements Interfaces
```

Onde *modificadores* podem ser:

- **abstract**: A classe não pode ser instanciada;
- **final**: A classe não pode ser herdada;
- **public**: A classe pode ser acessada por todos.

O corpo da classe é iniciado logo após a declaração da classe. Ele começa por *{* e termina por *}*. Sua estrutura é dividida em:

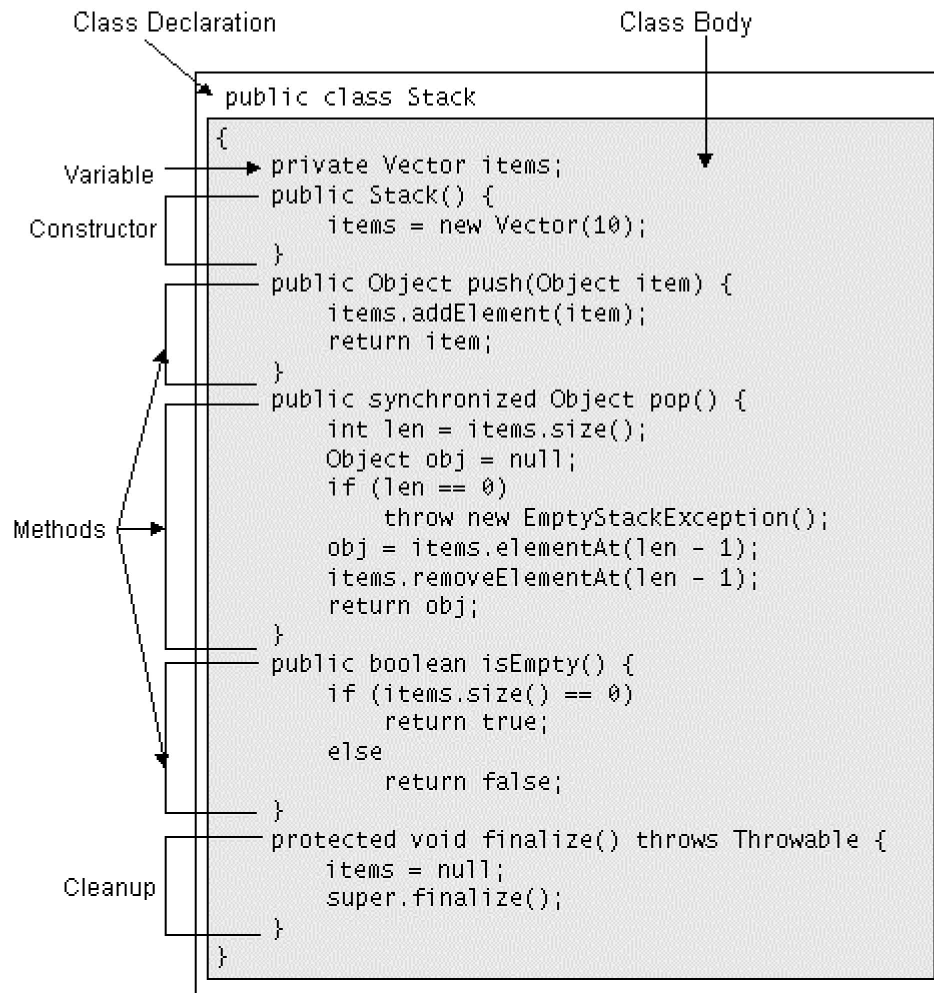
- Área de atributos (“variáveis”);
- Área de métodos (“funções”).

A área de métodos também pode ser semanticamente dividida entre construtores, métodos comuns e finalizador.

Os métodos construtores são os responsáveis pela instanciação da classe. Eles devem ter o mesmo nome da classe (sua assinatura é o seu nome e os seus parâmetros).

O método finalizador é chamado pelo sistema quando a última referência ao objeto é removida. A sua função é a finalização de arquivos, *sockets*, ...

Todas as classes herdam da classe *Object*. A seguinte figura descreve a estrutura de uma classe Java:



Fonte: Campione; Walrath; 1998.

Instanciação de classes

A instanciação de classes (construção de objetos) em Java é obtida da seguinte forma:

```
NomeDaClasse nomeDoObjeto;
NomeDoObjeto = new NomeDaClasse(parâmetros);
```

A primeira linha foi responsável pela atribuição do tipo *NomeDaClasse* à variável *nomeDoObjeto*. Na segunda linha um novo objeto foi construído e associado à variável *nomeDoObjeto*. Essas duas linhas são iguais à seguinte linha:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse(parâmetros);
```

Um objeto vai existir desde o momento da sua construção, e até que todas as suas referências sejam removidas (ou até que o programa termine). A remoção de uma referência a um objeto pode se dar explicitamente (nomeDoObjeto = null) ou implicitamente (fim do escopo de existência do objeto). O *garbage collector* se responsabiliza pela liberação da área de memória utilizada por um objeto quando este não tem mais referências.

Por convenção, os nomes de objetos começam por letras minúsculas e os nomes de classes por letras maiúsculas.

Interfaces

Tendo em vista que Java não utiliza herança múltipla, tornou-se necessária a existência de um mecanismo para prover polimorfismo para a classe que necessita de mais de uma herança. Esse mecanismo é a Interface.

Podemos ver Interfaces como uma espécie de contrato entre a classe que a implementa e o compilador Java. A Interface contém declarações (não implementadas) de métodos, e declarações de constantes. Quando uma classe tem na sua declaração o texto *implements NomeDaInterface*, ela está se comprometendo (assinando um contrato) de implementar todos os métodos dessa interface. Abaixo segue um exemplo de declaração de uma interface:

```
public interface Drawable
{
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}
```

Fonte: Flanagan; 1997.

Supondo que existam três classes (DrawableRectangle, DrawableCircle e DrawableSquare) que implementam a interface Drawable, então, utilizando o princípio de polimorfismo, podemos criar um *array* de *Drawable* e povoá-lo de objetos das classes *DrawableRectangle*, *DrawableCircle* e *DrawableSquare*. Tendo o *array* povoado, podemos entrar em um *loop* e chamar os métodos da interface Drawable para todos os seus elementos. O seguinte código exemplifica a execução do polimorfismo acima (apoiado por interface):

```
Drawable[] drawables = new Drawable[3]; // cria um array de Drawables.

// Cria alguns objetos de classes que implementam Drawable

DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3, 4.5);

// Insere (cria referência para) objetos em posições do array.

drawables[0] = dc;
drawables[1] = ds;
drawables[2] = dr;

// Exibe os objetos, chamando um método comum, declarado na interface.

for(int i = 0; i < drawables.length; i++)
{
  drawables[i].draw(draw_window); //assuma que draw_window já foi definida.
}
```

Fonte: Flanagan; 1997.

Métodos

Estrutura, Declaração e Utilização

Os métodos em Java tem a seguinte estrutura:

```
Modificadores TipoDeRetorno nomeDoMetodo(tipo1 nome1, tipo2 nome2,...)
{
      CORPO DO MÉTODO;
      ...
}
```

Onde *Modificadores* são os modificadores exibidos na seção “Modificadores”, *TipoDeRetorno* é o tipo do objeto retornado pelo método e *tipoN nomeN* são as declarações de objetos passados como parâmetro.

A chamada de um método é formada pelo nome da classe e pelo nome do método (com os seus devidos parâmetros), separados por um ponto, como exemplificado a seguir:

```
// Declaração de constantes

final int BRANCO = 255;
...
final int PRETO = 0;

// Declaração de variáveis

boolean mamifero = true;
int cor = PRETO;

// Instância um cachorro

Animal cachorro = new Animal(mamifero, cor);

cachorro.atribuiCor(BRANCO);
```

Sobrecarga

Sobrecarga (*Overloading*) de métodos é a possibilidade de declarar métodos com o mesmo nome, só que com passagem de parâmetros diferentes. A maior utilidade de sobrecarga de métodos é para a construção de métodos que fazem a mesma coisa com dados diferentes. Veja o exemplo a seguir:

```
...
public void atribuiCor(int cor)
{
  this.cor = cor;
}

public void atribuiCor(CorRGB cor)
{
  this.cor = RGB2Gray(cor);
}
```

Atributos

Os atributos em Java são declarados da seguinte forma:

```
Modificadores Tipo nomeDoAtributo;
```

Onde *Modificadores* são os modificadores exibidos na seção “Modificadores”, e *Tipo* é o tipo do atributo. Na declaração também pode ser definido o valor inicial do atributo. Veja o exemplo, a seguir:

```
final float aceleraçãoDaGravidade = 9.8F
```

O *9.8F* indica que *9.8* está sendo escrito como *float*.
Quando o atributo é declarado como *final*, ele não poderá ser modificado (é uma constante).

Modificadores

Visibilidade

Os modificadores de visibilidade indicam o nível de acesso ao método ou atributo. Na tabela a seguir, temos todos os níveis possíveis e suas visibilidades:

Modificador	**Visibilidade**
Private	Somente dentro da classe
Sem modificador	Em todas as classes do mesmo pacote
Protected	Em todas as classes do mesmo pacote e em subclasses
Public	Todas as classes

Fonte: Niemeyer; Peck; 1997

Static

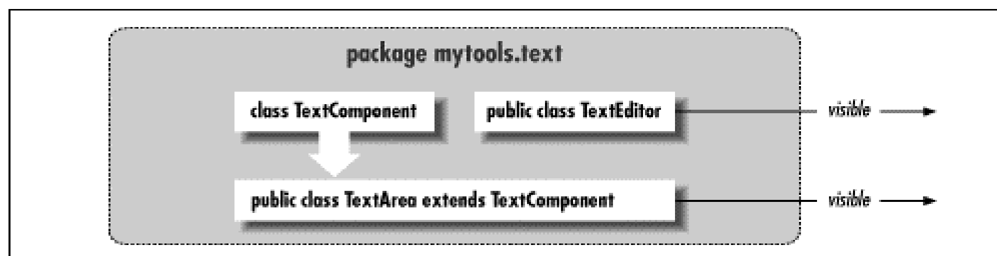
O modificador *static* indica que o método ou atributo é de classe (as instâncias criadas não terão esse método ou atributo), o que implica na não utilização da palavra reservada *this* (referencia ao próprio objeto) no interior desse tipo de método, pois este nunca vai fazer parte de nenhum objeto.

Métodos *static* só podem fazer referência a outros métodos ou atributos se estes forem também *static*. Veja, a seguir, um exemplo de método e atributo *static*:

```
Cor cor = new Cor(0,0,255);

int corGray = Cores.RGB2Gray(cor); // método de classe

if (corGray = Cores.corPadraoGray) // atributo de classe
{
  System.out.println("A cor padrão em tons de cinza é azul");
}
```

Pacotes

O uso de pacotes é a alternativa para agregar classes relacionadas. Existem modificadores (seção anterior) que permitem que certos métodos sejam vistos por outras classes, desde que estas estejam no mesmo pacote. No exemplo abaixo, a classe *TextComponent* só pode ser vista dentro do pacote *mytools.text*, pois ela foi declarada sem modificador.



Fonte: Niemeyer; Peck; 1997

Cada arquivo pode conter múltiplas classes, sendo que somente uma poderá ser *public* (e esta terá que ter o mesmo nome do arquivo). Cada arquivo pode pertencer somente a um pacote, mas um pacote pode conter vários arquivos. Para declarar o nome do pacote, deve-se utilizar a seguinte sintaxe na primeira linha de código do arquivo:

```
package MeuPacote;
```

Pacotes podem ser vistos como bibliotecas de classes. Para incluir classes de um pacotes em um arquivo, deve ser utilizada a seguinte sintaxe:

```
import MeuPacote.*;
```

Pacotes são vistos pelo sistema operacional como diretórios.

3 - A Linguagem Java

Tipos Primitivos

Os tipos primitivos em Java, diferentemente de objetos (que necessitam de construção e são tratados por referência), não necessitam de construção (instanciação através da palavra reservada *new*) e são tratados por valor. Esses tipos são:

Tipo	Tamanho/Formato	Descrição
(números inteiros)		
`Byte`	8-bit complemento a dois	Inteiro de tamanho de um byte
`Short`	16-bit complemento a dois	Inteiro curto
`Int`	32-bit complemento a dois	Inteiro
`Long`	64-bit complemento a dois	Inteiro longo
(números reais)		
`Float`	32-bit IEEE 754	Ponto flutuante
`Double`	64-bit IEEE 754	Ponto flutuante de precisão dupla
(outros tipos)		
`Char`	16-bit caracter Unicode	Um único caracter
`Boolean`	true ou false	Um valor booleano (`verdadeiro ou falso`)

Fonte: Campione; Walrath; 1998

Abaixo seguem trechos de código para auxiliar a compreensão da sintaxe de declaração de tipos primitivos:

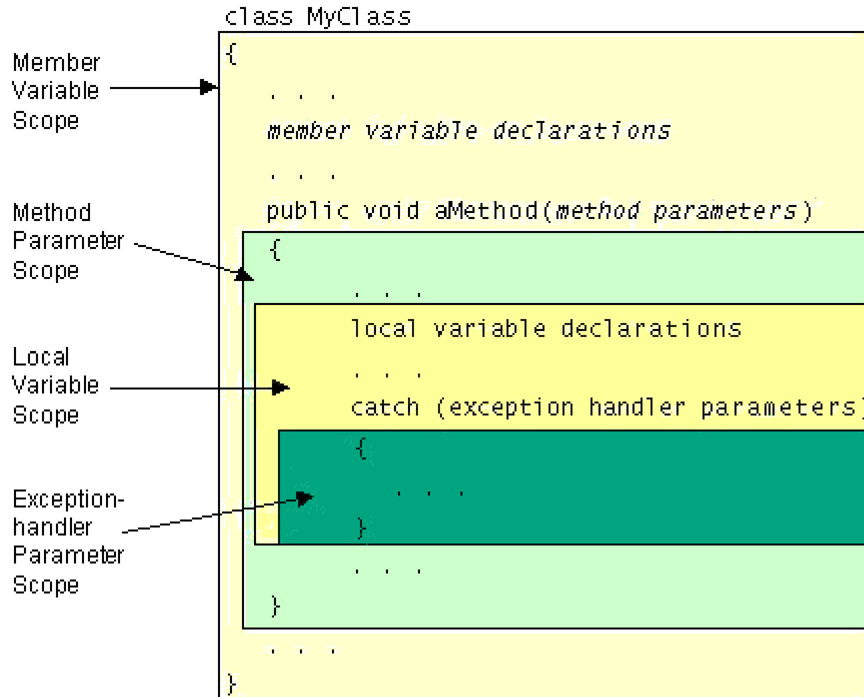
```
byte tamanhoDoPe = 38;
int numeroDaPorta = 4096;

numeroDaPorta += 40;    // passa a conter 4136

float nota = 7.25F;  // F indica que o numero é um float

final boolean VERDADE = true;  // Constante VERDADE com valor true
```

Apesar de String não ser um tipo primitivo, ele pode ser visto como um, pois pode ser construído diretamente, sem o uso de *new*, como está descrito no exemplo a seguir:

```
String texto = "Exemplo!!!";  // cria uma String com o valor "Exemplo!!!"
System.out.println(texto);  // mostra a String
```

Para obter mais informações sobre a classe String vá para a página 28.

O escopo de existência de uma variável depende do local de declaração da mesma. A figura abaixo ilustra essa relação:



Fonte: Campione; Walrath; 1998

Arrays

A declaração de um *array* de um determinado tipo é indicada pelo uso de *[]*. Cada *[]* indica uma dimensão do *array*. É necessária também a indicação do tamanho do *array* para a sua utilização. A sintaxe é a seguinte:

```
TipoOuClasse[] nomeDoArray = new TipoOuClasse[tamanhoDoArray]; // 1D

TipoOuClasse[] nomeDoArray;
nomeDoArray = new TipoOuClasse[tamanhoDoArray1]; // idem a forma anterior

TipoOuClasse[][] nomeDoArray = new TipoOuClasse[tam1][tam2]; // 2D
TipoOuClasse[][]...[] nomeDoArray = new TipoOuClasse[t1][t2]...[tn]; // nD
```

Para descobrir o tamanho de um *array* em tempo de execução, pode ser usado o método *length*, como descrito no exemplo a seguir:

```
Carro[] carros = new Carro[3]; // array de 3 carros
carros[0] = new Carro("BMW", 60000);
carros[1] = new Carro("Vectra", 40000);
carros[2] = new Carro("Gol", 20000);
int[] precos = new int[3];
for (int i = 0; i < carros.length; i ++)
{
  precos[i] = carros[i].getPreco(); // copia os precos
  System.out.println("Preço: " + precos[i]);  // mostra os precos na tela
}
```

Operadores

Operadores aritméticos:

Operador	Uso	Descrição
+	op1 + op2	Soma op1 com op2
-	op1 - op2	Subtrai op2 de op1
*	op1 * op2	Multiplica op1 por op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Calcula o resto da divisão (módulo) de op1 por op2
++	op++	Incrementa op de 1; utiliza o valor antes de incrementar
++	++op	Incrementa op de 1; utiliza o valor depois de incrementar
--	op--	Decrementa op de 1; utiliza o valor antes de incrementar
--	--op	Decrementa op de 1; utiliza o valor depois de incrementar

Fonte: Campione; Walrath; 1998

Operadores relacionais:

Operador	Uso	Retorna verdadeiro se
>	op1 > op2	op1 é maior que op2
>=	op1 >= op2	op1 é maior ou igual que op2
<	op1 < op2	op1 é menor que op2
<=	op1 <= op2	op1 é menor ou igual que op2
==	op1 == op2	op1 e op2 são iguais
!=	op1 != op2	op1 e op2 são diferentes

Fonte: Campione; Walrath; 1998

Operadores condicionais:

<table>
<tr><th>Operador</th><th>Uso</th><th>Retorna verdadeiro se</th></tr>
<tr><td>&&</td><td>op1 && op2</td><td>Ambos op1 e op2 são verdadeiros. Avalia a expressão op1 e condicionalmente avalia a expressão op2</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>Ou op1 ou op2 é verdadeiro. Avalia a expressão op1 e condicionalmente avalia a expressão op2</td></tr>
<tr><td>!</td><td>! op</td><td>op é falso</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>Ambos op1 e op2 são verdadeiros. Sempre avalia op1 e op2</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>Ou op1 ou op2 é verdadeiro. Sempre avalia op1 e op2</td></tr>
</table>

Fonte: Campione; Walrath; 1998

Operadores de bits:

Operador	Uso	Operação
`>>`	`op1 >> op2`	desloca os bits de `op1` para a direita de uma distância `op2`
`<<`	`op1 << op2`	desloca os bits de `op1` para a esquerda de uma distância `op2`
`>>>`	`op1 >>> op2`	desloca os bits de `op1` para a direita de uma distância `op2` (sem sinal)
`&`	`op1 & op2`	`e` de bits
|	`op1 | op2`	`ou` de bits
`^`	`op1 ^ op2`	`ou exclusivo` de bits
`~`	`~op2`	`complemento` de bits

Fonte: Campione; Walrath; 1998

Operadores de atribuição:

Operador	Uso	Equivalente a
`+=`	`op1 += op2`	`op1 = op1 + op2`
`-=`	`op1 -= op2`	`op1 = op1 - op2`
`*=`	`op1 *= op2`	`op1 = op1 * op2`
`/=`	`op1 /= op2`	`op1 = op1 / op2`
`%=`	`op1 %= op2`	`op1 = op1 % op2`
`&=`	`op1 &= op2`	`op1 = op1 & op2`
|=	`op1 |= op2`	`op1 = op1 | op2`
`^=`	`op1 ^= op2`	`op1 = op1 ^ op2`
`<<=`	`op1 <<= op2`	`op1 = op1 << op2`
`>>=`	`op1 >>= op2`	`op1 = op1 >> op2`
`>>>=`	`op1 >>>= op2`	`op1 = op1 >>> op2`

Fonte: Campione; Walrath; 1998

Repetição e Decisão

Os tipos de controle de fluxo de um programa em Java estão descritos a seguir:

Tipo	**Palavra chave**
Decisão	`if-else`, `switch-case`
Repetição	`for`, `while`, `do-while`
Exceção	`try-catch-finally`, `throw`
Outros	`break`, `continue`, `label:` , `return`

Fonte: Campione; Walrath; 1998

If-else

Sintaxe:

```
if (expressãoBooleana)
{
    // Código executado caso expressãoBooleana seja verdadeira
}
else
{
    // Código executado caso expressãoBooleana seja falsa
}
```

Switch-case

Sintaxe:

```
switch (expressão)
{
  case valor1:
    // Código executado caso expressão seja igual a valor1
  case valor2:
    // Código executado caso expressão seja igual a valor2
  ...
  case valorN:
    // Código executado caso expressão seja igual a valorN
  default:
    // Código executado caso expressão seja diferente de todos os valores
    // anteriores
}
```

Para que, após a execução de um código pertencente a um determinado *case*, o programa saia do *switch* (sem executar os *cases* consecutivos), é necessário o uso de *break*. Veja o exemplo a seguir:

```
int month;
int numDays;
. . .
switch (month)
{
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    numDays = 31;
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    numDays = 30;
    break;
  case 2:
    if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
      numDays = 29;
    else
      numDays = 28;
    break;
}
```

Fonte: Campione; Walrath; 1998

For

Sintaxe:

```
for (inicialização; expressãoBooleana; incremento)
{
  // Código executado enquanto expressãoBooleana for verdadeira
}
```

Por exemplo, para corrermos todos os elementos de um array, atribuindo um valor default igual a posição do elemento no array, podemos utilizar o seguinte código:

```
for (int i = 0; i < posicao.length; i++)
{
  posicao[i] = i;
}
```

While

Sintaxe:

```
while (expressãoBooleana)
{
  // Código executado enquanto expressãoBooleana for verdadeira
}
```

Do-while

Sintaxe:

```
do
{
  // Código executado enquanto expressãoBooleana for verdadeira
}
while (expressãoBooleana)
```

Try-catch-finally

Quando um método tem o poder de disparar uma exceção (isto é, possui *throw* no seu cabeçalho), ele deve ser utilizado dentro de um bloco *try-catch-finally* (onde o *finally* não é obrigatório). A sintaxe é a seguinte:

```
try
{
  // Código que contém o método que pode disparar exceções
}
catch (ClasseDeExcecao1 e)
{
  // Código que deve ser executado caso algum método contido no bloco try
  // dispare uma exceção do tipo ClasseDeExcecao1
}
catch (ClasseDeExcecao2 e)
{
  // Código que deve ser executado caso algum método contido no bloco try
  // dispare uma exceção do tipo ClasseDeExcecao2
}
...
catch (ClasseDeExcecaoN e)
{
  // Código que deve ser executado caso algum método contido no bloco try
  // dispare uma exceção do tipo ClasseDeExcecaoN
}
finally
{
  // Código que sempre deve ser executado, mesmo que tenha ocorrido uma
  // exceção no bloco try
}
```

Continue e break

A qualquer momento, dentro de um loop (for, while e do-while) pode ser feita uma reconsideração da condição de execução do loop. Para que isso ocorra é utilizada a chamada *continue*. Ao encontrar um *continue*, o programa é desviado para o início do loop, no caso de for e while, ou para o final do loop, no caso de do-while, para a reavaliação. Para a interrupção da execução de um loop, é utilizada a cláusula *break*. Tanto o *continue* quanto o *break* podem conter um parâmetro, que é o *label* indicativo de qual loop deve ser reavaliado ou terminado. Veja o exemplo abaixo:

```
public int indexOf(String str, int fromIndex)
{
  char[] v1 = value;
  char[] v2 = str.value;
  int max = offset + (count - str.count);
test:
  for (int i = offset + ((fromIndex < 0) ? 0 : fromIndex); i <= max ; i++)
  {
    int n = str.count;
    int j = i;
    int k = str.offset;
    while (n-- != 0)
      if (v1[j++] != v2[k++])
        continue test;
    return i - offset;
  }
  return -1;
}
```

Fonte: Campione; Walrath; 1998

4 - Pacotes Essenciais do JDK 1.1

http://java.sun.com/products/jdk/1.1/docs/api/packages.html

Applet

Para que uma aplicação Java rode em um browser (Internet) é necessário que ela herde de Applet (convencionalmente chamamos de *applet* toda aplicação que herda de Applet).

Quando um *applet* é criado, alguns métodos devem ser observados com atenção, pois são eles os responsáveis por responder a eventos do browser. Esses métodos são os seguintes:

init()	Chamado pelo browser quando os recursos necessários para a execução do applet são alocados (inicializa o applet)
start()	Chamado pelo browser quando o applet deve iniciar a execução (indica ganho de foco do applet)
stop()	Chamado pelo browser quando o applet deve parar a execução (indica perda de foco do applet)
destroy()	Chamado pelo browser quando este necessita desalocar recursos concedidos ao applet (finaliza o applet)
paint()	Chamado pelo browser quando este necessita redesenhar o applet na tela.

Depois de pronto, o *applet* deve ser referenciado por uma *tag* em um arquivo HTML, de acordo com a seguinte sintaxe:

```
< APPLET
    [CODEBASE = URL do applet (default = URL da Página)]
    CODE = Nome do arquivo do applet
    [ALT = Texto para browsers que não conseguem rodar applet]
    [NAME = Nome para o applet para comunicação com outros applets]
    WIDTH = Largura inicial do applet (em pixels)
    HEIGHT = Largura inicial do applet (em pixels)
    [ALIGN = Alinhamento do applet (tq o utilizado com imagens)]
    [VSPACE = Borda vertical do applet (tq a utilizado com imagens)]
    [HSPACE = Borda horizontal do applet (tq a utilizado com imagens)]
>
[< PARAM NAME = Nome do 1° parâmetro passado ao applet VALUE = Valor1 >]
[< PARAM NAME = Nome do 2° parâmetro passado ao applet VALUE = Valor2 >]
. . .
[Código HTML alternativo para browsers que não reconhecem applet]
</APPLET>
```

Fonte: Campione; Walrath; 1998

Para fazer a coleta dos parâmetros que podem ser passados dentro da *tag* HTML, é utilizado o método *getParameter(String name)*, onde *name* é o nome do parâmetro. O valor do parâmetro é retornado.

AWT

O AWT é o pacote gráfico Java, responsável por todos os componentes de interface para aplicativos e applets. Abaixo são descritos os componentes mais utilizados.

Button

Componente gráfico que representa um botão. Este botão emite o evento *ActionEvent* que indica a ocorrência de um clique.

Canvas

Área de desenho que pode tratar eventos gerados pelo usuário.

Checkbox

Caixa de marcação que pode ser agrupada em *CheckboxGroups*, onde somente uma das caixas do grupo pode estar selecionada em um instante de tempo, ou pode ser utilizada sem agrupamento, possibilitando a ocorrência de mais de uma caixa selecionada no mesmo instante de tempo.

Dialog

Janela que pode ser modal em relação a outra janela. A *Dialog* gera os seguintes eventos:

- WindowOpened;
- WindowClosing;
- WindowClosed;
- WindowActivated;
- WindowDeactivated.

Frame

Janela de alto nível que gera todos os eventos de janela possíveis em Java, que são:

- WindowOpened;
- WindowClosing;
- WindowClosed;
- WindowIconified;
- WindowDeiconified
- WindowActivated;
- WindowDeactivated.

A principal diferença entre *Dialog* e *Frame* é que *Frame* é uma janela completa, com todas as opções que o sistema oferece (ex.: minimizar) e *Dialog* é mais simples, e utilizada como caixa de diálogo com o usuário.

Label

Texto somente de leitura (pode ser alterado pelo programa via método *setText()*) com apenas uma linha.

Layouts

Forma de disposição de componentes em um *container* (componente que pode agregar outros componentes). Os layouts mais utilizados são os seguintes:

- **BorderLayout**: Coloca os componentes segundo um parâmetro, que pode ser Norte, Sul, Leste, Oeste e Centro.
- **FlowLayout**: Coloca os componentes sequencialmente na vertical. Quando não há mais espaço, vai para a próxima linha.
- **GridLayout**: Coloca os componentes numa disposição definida pela grade de N colunas e M linhas, onde N e M são parâmetros.
- **XYLayout**: Coloca os componentes em uma posição absoluta (x,y) do *container*.

Deve ser dada preferência a layouts que não trabalham com posições absolutas, pois estas podem ser afetadas devido a configurações e plataformas diferentes.

List

Lista com barra de rolagem que pode ser configurada para permitir a seleção de um ou de múltiplos elementos.

Menu

A estrutura de menu em Java é composta pela barra de menus de uma janela (classe *MenuBar*), onde cada janela tem somente uma barra de menus, que agrega elementos de menus (classe *Menu*). Cada *Menu*, por sua vez, agrega itens de menu (classe *MenuItem*), que podem ser itens comuns (que geram eventos de clique de mouse), outros elementos da classe *Menu* (que formaria um menu aninhado), ou elementos da classe *CheckboxMenuItem* (que são itens com caixas de checagem). Também existe um tipo de menu que pode ser dinamicamente apresentado em uma determinada posição de um componente (classe *PopupMenu*).

Panel

Painel é um *container* simples que recebe um layout próprio e agrega outros componentes, inclusive outros painéis. Ele pode ser utilizado para combinar os esquemas de *layout*, formando janelas com diagramação mais complexa.

TextArea

Área de múltiplas linhas que pode ser utilizada tanto para visualização quanto para edição de textos. Na sua configuração, pode ser determinado o número de linhas e colunas, além do tipo de barra de rolagem.

TextField

Campo de texto de uma linha que permite edição e gera dois eventos *ActionPerformed* a cada letra digitada. O primeiro evento se refere ao pressionar da tecla, e o segundo ao soltar da tecla.

Lang

Math

A classe *Math* fornece o ferramental básico para manipulações matemáticas. Essa classe é composta somente por atributos e métodos de classe (*static*), o que significa não ser necessária a instanciação de um objeto desta classe para a sua utilização.

A seguir, são detalhados os seus atributos e principais métodos, com uma descrição sucinta:

Atributo/Método	Descrição
E	base de logaritmos Neperianos
PI	razão da circunferência de um circulo pelo seu diâmetro
abs(N)	valor absoluto de N
acos(N)	arco coseno de N
asin(N)	arco seno de N
atan(N)	arco tangente de N
ceil(N) e cos(N)	coseno de N
exp(N)	*E* elevado a N
floor(N)	piso de N
log(N)	logaritmo Neperiano de N
max(N,M)	maior valor entre N e M
min(N,M)	menor valor entre N e M
pow(N,M)	N elevado a M
random()	número aleatório entre zero e um
rint(N) e round(N)	arredondamento de N
sin(N)	seno de N
sqrt(N)	raiz quadrada de N
tan(N)	tangente de N

String

Como já foi visto anteriormente, strings podem ser tratadas como tipos primitivos, pois sua instanciação pode ser feita diretamente, sem a utilização da palavra reservada *new*. Na realidade, *String* é uma classe, e quando uma instanciação direta é feita, internamente a Máquina Virtual Java converte o texto entre aspas para um objeto da classe string. Devido a isso, o seguinte trecho de código se torna possível:

```
System.out.println("teste".length()); //mostra o valor 5
```

A classe *String* fornece o apoio necessário para manipulação de strings com os seus métodos.

A instanciação de *String* é usualmente obtida diretamente, ou partindo de um array de caracteres. Os dois trechos de código abaixo são equivalentes, e exemplificam as duas formas citadas acima:

```
String texto = "teste";
```

```
char letras[] = {'t', 'e', 's', 't', 'e'};
String texto = new String(letras);
```

Tendo em vista que strings em Java são tratadas como objetos, conclui-se que o operador == não retorna se o valor das strings é igual, mas se as variáveis fazem referência para o mesmo objeto, como é demonstrado no trecho de código a seguir:

```
String s1,s2,s3;
s1 = "teste";
s2 = "teste";
s3 = s1;
if (s1 == s2)  // retorna falso
{
...
}
if (s1 == s3)  // retorna verdade
{
...
}
```

Para fazer comparação entre valores de strings, deve ser utilizado o método equals(String), da seguinte forma:

```
String s1,s2,s3;
s1 = "teste";
s2 = "teste";
if (s1.equals(s2))  // retorna verdade (identico a s2.equals(s1))
{
...
}
if ("teste".equals(s1))  // retorna verdade (identico a s1.equals("teste"))
{
...
}
```

Util

As três classes apresentadas nesta seção são as responsáveis por armazenamento de objetos em Java. A maior diferença entre *Hashtable*, *Vector* e *Stack* é a forma de armazenamento adotada. Em *Hashtable*, os objetos são armazenados segundo uma chave que os indexa em uma tabela *hash*. Em *Vector*, os objetos são armazenados sequencialmente. Já em *Stack*, os objetos são armazenados em uma estrutura de pilha (*last in first out*).

Hashtable

Permite o armazenamento de objetos segundo um índice, e tem como métodos principais os seguintes:

Método	**Descrição**
clear()	Esvazia a hashtable
contains(Objeto)	Verifica se o objeto esta na hashtable
containsKey(Chave)	Verifica se o objeto é chave da hashtable
elements()	Retorna os objetos que estão na hashtable
get(Chave)	Retorna o objeto mapeado por Chave
isEmpty()	Verifica se a hashtable está vazia
keys()	Retorna as chaves da hashtable
put(Chave, Objeto)	Mapeia o Objeto pela Chave na hashtable
rehash()	Reorganiza os objetos em uma hashtable maior
remove(Chave)	Remove a Chave e o objeto mapeado por ela da hashtable
size()	Retorna o número de elementos da hashtable

O seguinte trecho de código exemplifica a utilização de uma *hashtable*:

```
Hashtable numbers = new Hashtable();

numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

Integer n = (Integer)numbers.get("two");
if (n != null)
  System.out.println("two = " + n);
```

Fonte: Sun Microsystems

Vector

Armazena seqüencialmente objetos, tendo os seguintes métodos principais:

addElement(Objeto)	Adiciona Objeto no fim do Vector
contains(Objeto)	Verifica se o vector contem Objeto
elementAt(Posicao)	Retorna o elemento em Posição
elements()	Retorna os elementos do Vector
firstElement()	Retorna o primeiro elemento do Vector
insertElementAt(Obj, Pos)	Adiciona Obj na posição Pos
isEmpty()	Verifica se o Vector está vazio
lastElement()	Retorna o último elemento do Vector
removeAllElements()	Remove todos os elementos
removeElement(Objeto)	Remove a primeira ocorrência de Objeto no Vector
removeElementAt(Posicao)	Remove o elemento da posição Posição
size()	Retorna o número de elementos do Vector

O seguinte trecho de código exemplifica a sua utilização:

```
String one = "one";
String two = "two";
String three = "three";
things.addElement( one );
things.addElement( three );
things.insertElementAt( two, 1 );

String s1 = (String)things.firstElement();       // "one"
String s3 = (String)things.lastElement();        // "three"
String s2 = (String)things.elementAt(1);         // "two"
```

Fonte: Niemeyer; Peck; 1997.

Stack

Herda de *Vector*, mantém uma estrutura de pilha e tem como principais métodos os seguintes:

Método	**Descrição**
empty()	Verifica se a pilha está vazia
peek()	Olha o objeto que está no topo da pilha sem removê-lo
pop()	Retorna o objeto que está no topo da pilha removendo-o
push(Objeto)	Coloca Objeto no topo da pilha
search(Objeto)	Retorna a posição de Objeto na pilha

O seguinte código exemplifica a sua utilização:

```
Stack numbers = new Stack();

numbers.push("one");
numbers.push("two");
numbers.push("three");

String tres = (String)numbers.pop(); // “three”
String dois = (String)numbers.pop(); // “two”
String um = (String)numbers.pop();   // “one”
```

5 - Tópicos Básicos

Eventos

A forma utilizada pelo Java para fazer a comunicação entre usuário e objetos é através de eventos disparados à partir de ações na interface gráfica. Alguns objetos da interface delegam o tratamento do evento para outros objetos (que implementam interfaces *listener*). O modelo utilizado pelo AWT 1.1 para eventos é composto das seguintes etapas:

1. Construir classes que implementam alguma interface do tipo *listener*. Essas classes serão as responsáveis pelo tratamento do evento. Dependendo de qual interface *listener* for implementada, um determinado método será chamado quando um evento registrado ocorrer. Abaixo segue um exemplo de classe *listener*:

```
public class ClasseListener implements ActionListener
{
  public ClasseListener()
  {
    // Inicialização do objeto
  }

  public void actionPerformed(ActionEvent e) // método referente a listener
  {
    // Código que será executado quando um evento registrado ocorrer
  }
}
```

2. Registrar objetos como *listeners* de eventos gerados em componentes de interface, como descrito a seguir:

```
...
ClasseListener listener = new ClasseListener();
Button button = new Button();
button.addActionListener(listener);
...
```

Sempre que o botão *button* for clicado, o método *actionPerformed* do objeto *listener* será executado. Deve ser ressaltado que mais de um objeto do tipo *listener* pode ser registrado para tratar eventos de um componente de interface, e vice-versa.

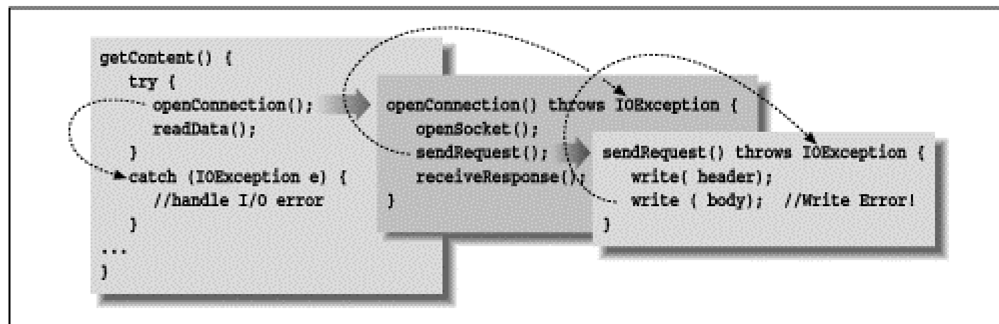
Exceções

Devido ao fato de Java ter sido projetado para ser interpretado em aparelhos eletrônicos, onde é necessário altos níveis de segurança, foi criado um modelo de tratamento

de exceções, que provê ao programador suporte ao tratamento de erros. Esse modelo permite dois tipos de ações:

- repasse: **throws**
- tratamento: **try/catch/finally**

Sempre que, dentro de um método, se torna necessário chamar algum método que pode disparar exceções, essa chamada tem que ser feita dentro de um bloco try/catch/finally (como descrito na página 22), ou o método chamador deve conter em seu cabeçalho uma indicação que ele pode lançar uma determinada exceção (throws). Desta forma, a exceção vai percorrendo os níveis de chamada de métodos (em sentido inverso às chamadas) até ser capturada por um bloco try/catch/finally, como exibido na figura a seguir:



Fonte: Niemeyer; Peck; 1997.

Serialização

Para que se torne possível guardar em disco uma situação de memória (objetos instanciados na memória), Java utiliza a serialização. Serialização consiste no armazenamento de um objeto (raiz de persistência) e de todos os objetos que são referenciados por este (recursivamente). Desta forma, a raiz de persistência é a raiz de uma árvore onde todos os objetos e ponteiros são armazenados. Abaixo, segue um exemplo de armazenamento de uma hashtable (e, consequentemente, todos os objetos contidos nela):

```
Hashtable objeto = new Hashtable();
...
try
{
  FileOutputStream fileOut = new FileOutputStream("arquivo.xxx");
  ObjectOutputStream out = new ObjectOutputStream(fileOut);
  out.writeObject(objeto);
}
catch (Exception e)
{
  System.out.println(e);
}
```

Para recuperar o objeto armazenado no exemplo anterior, é utilizado o seguinte código:

```
Hashtable objeto;
try
{
  FileInputStream fileIn = new FileInputStream("arquivo.xxx");
  ObjectInputStream in = new ObjectInputStream(fileIn);
  objeto = (Hashtable)in.readObject();
}
catch (Exception e) { System.out.println(e); }
```

Threads

Para se fazer uso de *threads,* em Java, é necessário a implementação da interface *runnable* e, consequentemente, do método *run()*. Ao criar uma *thread* e chamar o seu método *start()*, o método *run()* do objeto que se deseja fazer multitarefa será executado. Não ocorrerá pausa para execução do método *start()*, o que indica que o computador estará fazendo duas (ou mais) tarefas ao mesmo tempo. O seguinte código exemplifica 3 *threads* contando, ao mesmo tempo, de 0 até 100:

```
public class Conta
{
  public static void main(String argv[])
  {
    Contador c1 = new Contador("Contador1");
    Contador c2 = new Contador("Contador2");
    Contador c3 = new Contador("Contador3");
    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);
    Thread t3 = new Thread(c3);
    t1.start();
    t2.start();
    t3.start();
  }
}

class Contador implements Runnable
{
  private String nome;

  public Contador(String nome)
  {
    this.nome = nome;
  }

  public void run()
  {
    for (int i = 0; i <= 100; i++)
      System.out.println(nome + ": " + i);
  }
}
```

Comentários de Código

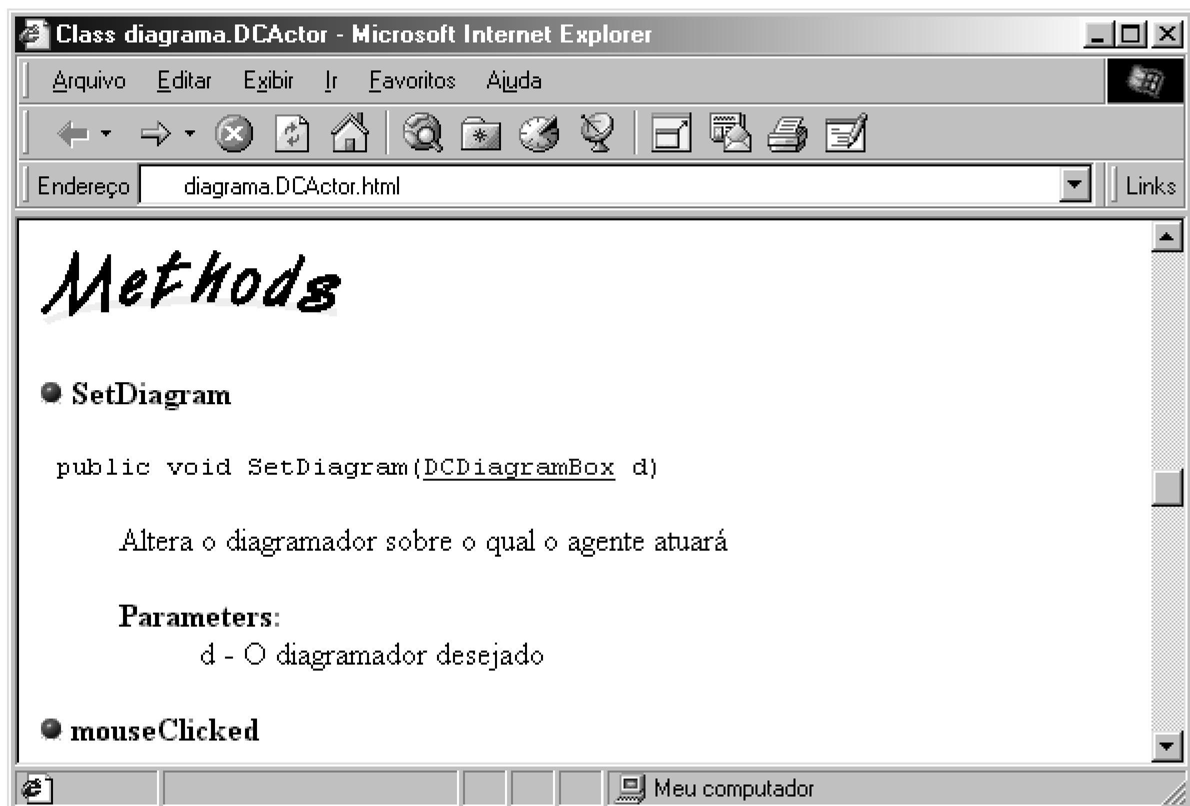
Java suporta comentários estilo C, começando por /* e terminando por */, e também aceita comentários estilo C++, começando por // e durando até a próxima linha.

Um estilo especial de comentário suportado por Java é o JavaDoc. Ele começa por /**, tem um * em cada linha seguinte e termina por */. O padrão JavaDoc tem como objetivo descrever cada método e atributo de uma classe. Essa descrição se dá através de *tags* que começam por @. Uma classe comentada (documentada) através de JavaDoc pode ser submetida a um interpretador JavaDoc, que irá gerar um HTML como documentação. Abaixo segue um exemplo de comentário JavaDoc em um método:

```
/**
 * Altera o diagramador sobre o qual o agente atuará
 *
 * @param d  O diagramador desejado
 */

public void SetDiagram(DCDiagramBox d)
{
  Diagram = d;
}
```

Fonte: Projeto Odyssey – COPPE/UFRJ

Após passar essa classe pelo interpretador JavaDoc, o seguinte HTML será gerado:



Janela de um browser exibindo pedaço do HTML que contém o método acima.

As tags JavaDoc mais utilizadas são:

TAG	DESCRIÇÃO
@author	Se o autor não é conhecido, use “unascribed” como argumento. *Ex.: @author Leonardo Gresta Paulino Murta*
@version	Convenção de string SCCS: "%I%, %G%", que é convertido para "1.39, 02/28/97". *Ex.: @version %I%, %G%*
@param	Recebe como argumentos o nome da variável (sem o tipo) e uma descrição sucinta. *Ex.: @param img a imagem a ser desenhada*
@return	Deve ser omitido em métodos void. Recebe como argumento a descrição do valor de retorno. *Ex.: @return a nova imagem, depois de desenhada*
@deprecated	Indica a partir de qual versão o método está obsoleto e qual método deve ser usado em seu lugar. Deve ser seguido do link para o novo método. *Ex. (JDK1.1):* *@deprecated A partir do JDK 1.1, trocado por setBounds* *@see #setBounds(int,int,int,int)* *Ex. (JDK1.2):* *@deprecated A partir do JDK 1.1, trocado por {@link #setBounds(int,int,int,int)}*
@exception (@throws, sinônimo adicionado no Javadoc 1.2)	Descreve as exceções que o método pode gerar. *Ex.: @exception IOException se alguma exceção de entrada ou saída ocorrer*

http://java.sun.com/products/jdk/javadoc/index.html

6 - Tópicos Avançados

IDL (Interface Definition Language)

Suporte para utilização de CORBA (Common Object Request Broker Architecture) com Java. Inclui um Java ORB que utiliza comunicação IIOP (Internet Inter-ORB Protocol).

http://www.javasoft.com/products/jdk/idl/index.html

Internationalization

Permite que programadores escrevam programas independentes da língua e cultura do usuário. São conhecidos como Programas Globais.

http://java.sun.com/products/jdk/1.1/docs/guide/intl/index.html

JAR (Java Archive)

Java Archive possibilita a união de vários arquivos (classes compiladas, sons, imagens) em um só, aumentando a velocidade de *download* de uma *applet* (o JAR é compactado).

http://java.sun.com/products/jdk/1.1/docs/guide/jar/index.html

Java Beans

Forma o modelo de reutilização de componentes de software (beans) em Java, permitindo a venda de pedaços de software que podem ser integrados para a construção de um programa.

http://www.javasoft.com/beans/index.html

Java Media

Conjunto de classes que permite utilização de multimídia (som, gráficos 3D, vídeos, telefonia,...) em Java.

http://www.javasoft.com/products/java-media/index.html

JDBC (Java Database Connectivity)

Interface SQL padrão para acesso a banco de dados variados. Vem com uma ponte para acesso ODBC (Open DataBase Connectivity – interface de programação para acessar bancos de dados).

http://www.javasoft.com/products/jdbc/index.html

Security

Suporte para utilização de níveis baixos e altos de segurança em aplicações e applets Java.

http://www.javasoft.com/security/index.html

Servlets

Permite criação de aplicações que serão executadas no servidor e se comunicarão com o usuário por meio de uma interface de formulário HTML (tal como CGIs).

http://www.javasoft.com/products/servlet/index.html

RMI (Remote Method Invocation)

Possibilita a chamada a métodos de objetos remotos por outra máquina virtual Java. Os objetos (chamador e chamado) podem estar em diferentes máquinas.

http://www.javasoft.com/products/jdk/rmi/index.html

Bibliografia

Campione, M.; Walrath, K.; 1998; “The Java Tutorial : Object-Oriented Programming for the Internet”, 2ª edição, Addison-Wesley Pub Co.

Dias, M.; 1998; “Java: muito além de uma linguagem de programação...”, transparências, DCC-IM/UFRJ.

Flanagan, D.; 1997; “Java in a Nutshell”, 2ª edição, O’Reilly.

Grand, M.; 1997; “Java Language Reference”, 2ª edição, O’Reilly.

Grand, M.; Knudsen, J.; 1997; “Java Fundamental Classes Reference”, 1ª edição, O’Reilly.

Niemeyer, P.; Peck, J.; 1997; “Exploring Java”, 2ª edição, O’Reilly.

Sun Microsystems; “JavaTM Platform 1.1.5 Core API Specification”, JDK documentation.

Zukowski, J.; 1997; “Java AWT Reference”, 1ª edição, O’Reilly.

Índice Remissivo

A

Ambientes de Desenvolvimento 5
Aplicação 8
Applet 1, 8, 24
AppletViewer 7, 8, 9
Arrays 17
Atributos 14
AWT 25, 32, 40

B

Button 5, 25, 32

C

C++ 1, 2, 3, 35
Canvas 25
Características 1
Checkbox 25
Classes 10, 40
Comentários de Código 7, 35
Continue e break 23

D

Decisão 19
Dialog 25, 26
Do-while 22

E

Environment 2
Estrutura 10, 13
Eventos 32
Exceções 32

F

For 21
Frame 26

H

Hashtable 29, 30, 33, 34
História 1

I

If-else 20
Instanciação de classes 11
Interfaces 10, 12
Internationalization 37

J

Java 6
Java Beans 38
Java Media 38
Javac 5
JavaDoc 7, 35, 36
Javah 7
Javap 7
Jdb 7
Jre 6

L

Label 26
Lang 27
Layouts 26
List 26

M

Main() 2
Math 27
Menu 27
Métodos 13, 15
Modificadores 13, 14

O

Operadores 18, 19

P

Pacotes 15, 24
Panel 27
Ponteiros 3

R

Repetição 19
RMI (Remote Method Invocation) 39

S

Security 39
Serialização 33
Servlets 39
Sobrecarga 3, 13
Stack 29, 31
Static 15
String 2, 3, 6, 16, 23, 25, 28, 29, 31, 34
Switch-case 20

T

TextArea 27
TextField 27
Threads 34
Tipos Primitivos 16
Try-catch-finally 22

U

Util 29

V

Vector 29, 30, 31
Visibilidade 14

W

While 22