UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA ESPECIAL DE TREINAMENTO
CIÊNCIAS DA COMPUTAÇÃO

Introdução à Linguagem Java

Lucas Wanner

Versão 1.1
Florianópolis, Outubro de 2002

.

Versões atualizadas deste documento devem estar disponíveis no endereço `http://monica.inf.ufsc.br`. Caso não seja possível obter o documento neste sítio, envie um e-mail para pet@inf.ufsc.br ou para o autor.

Este documento foi construído e formatado com LaTeX 2_ε

.

Sumário

Prefácio

Esta apostila foi concebida originalmente em Setembro de 2001 como material de apoio aos Cursos de Introdução à Linguagem Java ministrados pelos bolsistas do Programa Especial de Treinamento - Ciências da Computação da Universidade Federal de Santa Catarina (PET/CCO). O material e o curso destinam-se a principiantes em Java, que (preferencialmente) já tenham algum conhecimento em alguma outra linguagem de programação, especialmente nas orientadas a objetos, como C++, Object Pascal (Delphi), Smalltalk, Eiffel, etc.

Capítulo 1 traz uma breve introdução à Orientação a Objetos, com enfoque em Java. Embora bastante simples e incompleta, esta introdução deve cobrir todos os tópicos necessários à compreensão deste material.

Capítulo 2 apresenta a sintaxe de Java, com tópicos relacionados à variáveis, controle de fluxo, operadores, etc.

Capítulo 3 mostra a tradução dos conceitos da programação orientada a objetos para código em Java

Capítulo 4 apresenta tópicos relacionados a aplicações em Java, como pacotes, documentação, entrada e saída, etc.

Apêndice A traz instruções para geração de código, compilação e execução de um primeiro programa em Java

Apêndice B traz um glossário de termos usados

Embora destinado especialmente aos alunos do curso, este material deve servir também ao autodidata com bons conhecimentos em linguagens orientadas a objetos.

Na versão 1.1 foram corrigidos pequenos erros existentes na versão inicial, mas outros podem eventualmente aparecer. Não deixe de enviar indicações de erros ou outras sugestões ao autor.

Lucas Wanner
lucas@inf.ufsc.br

Programa Especial De Treinamento - PET/CCO
pet@inf.ufsc.br

Capítulo 1

Introdução à Programação Orientada a Objetos

Todas as linguagens de programação são abstrações. A complexidade da resolução de um determinado problema está diretamente ligada à qualidade e ao tipo de abstração que é feita dele. A linguagem Assembly é uma pequena abstração da máquina para qual ela foi criada. Muitas linguagens chamadas "estruturadas" ou "procedurais" (como C, BASIC e Fortran) são abstrações da linguagem Assembly. Estas linguagens foram um grande progresso em relação a linguagem Assembly, mas sua abstração inicial ainda requer que o programador pense em termos da estrutura do computador ao invés da estrutura do problema a ser resolvido. O efeito colateral são programas difíceis de escrever e caros para se manter.

A alternativa é modelar o problema que se está tentando resolver de acordo com um paradigma diferente do da máquina. Linguagens como LISP e APL escolheram visões particulares do mundo ("Todos os problemas são listas" e "Todos os problemas são algoritmos", respectivamente). PROLOG classifica todos os problemas como cadeias de decisões. Cada uma destas visões é uma boa solução para o tipo específico de problemas para o qual elas foram desenvolvidas, mas quando se sai deste domínio, elas em geral tornam-se inadequadas.

O paradigma de orientação a objetos vai um passo além, dando ao programador ferramentas para representar elementos do "espaço do problema". Esta representação é geral o suficiente para que o programador não fique restrito a um tipo específico de problema. Os elementos do espaço do problema são chamados "objetos". A idéia é que o programa possa adaptar-se à linguagem do problema, assim quando se lê o código descrevendo a solução (ou seja, o "espaço da solução"), também se está lendo a descrição do problema em si (ou o "espaço do problema"). Assim, a Programação Orientada a Objetos (POO) permite que o problema seja descrito nos termos do proprio problema, ao invés de em termos do computador onde será executada a sua solução.

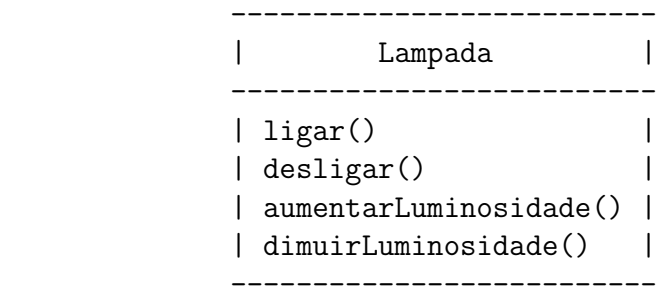
1.1 Classes e Objetos

O conceito de "classes" ou "tipos" [1] de objetos vem de Platão e Aristóteles. Platão acreditava que todos os objetos imperfeitos do "mundo dos sentidos" derivam de um conceito perfeito do "mundo das idéias". Este "conceito perfeito" funciona como uma forma de coelhos de chocolate. Todos os coelhos produzidos com a forma são do mesmo tipo, mas diferem levemente entre si. Um é um pouco menor, outro tem uma orelha quebrada, assim por diante. Aristóteles, tomando uma via mais prática, percebeu as semelhanças entre espécies de animais, falando da "classe dos peixes, a classe dos pássaros, . . . ".

A idéia de que todos os objetos, embora únicos, são parte de uma classe de objetos que tem características e comportamentos em comum foi utilizada diretamente na primeira linguagem orientada a objetos, Simula-67, com sua palavra chave fundamental "class" que introduz um novo tipo em um programa.

Objetos são idênticos exceto pelo seu estado durante a execução do programa. Criar tipos abstratos de dados (classes) é um conceito fundamental na programação orientada a objetos. Tipos abstratos de dados funcionam como os tipos primitivos (inteiro, real, char, etc.): pode-se criar variáveis de um tipo (chamadas objetos ou instâncias) e manipular estas variáveis (ou seja, mandar mensagens ou pedidos, manda-se uma mensagem ao objeto e ele entende o que fazer com ela). Os membros (elementos) de cada classe têm algo em comum: todas as pessoas têm nome, todos os livros têm um número de páginas, etc. Ao mesmo tempo, cada membro tem seu próprio estado, cada pessoa tem um nome diferente, cada livro tem um número diferente de páginas. Assim, pessoas, livros, etc., podem ser representados com uma entidade única no programa. Esta entidade é o objeto, e cada objeto pertence a uma classe que define seus atributos (características) e métodos (comportamento).

Quando se definem novos tipos, deve haver uma maneira de fazer um pedido a um objeto deste tipo para que o mesmo faça algo, como completar uma transação, desenhar algo na tela, ou ligar um interruptor. Cada objeto só pode satisfazer certos pedidos. Os pedidos que podem ser feitos a um objeto são determinados pela sua *interface*. Um exemplo simples pode ser a representação de uma lâmpada:




```
1 Lampada lamp = new Lampada();
2 lamp.ligar();
```

A parte superior do diagrama representa o nome da classe, e a inferior, sua interface. A interface estabelece que pedidos podem ser feitos a um deter-

[1] Este curso considera "classe" e "tipo" como sinônimos

minado objeto. Entretanto, deve haver um código que satisfaça este pedido. Isto, junto com os dados ocultos, ou protegidos, constitui a implementação. Do ponto de vista estrutural, um tipo tem uma função associada a cada pedido possível, e quando se faz um pedido a um objeto, sua respectiva função é chamada. Este processo é sumarizado dizendo-se que se "manda uma mensagem" (faz um pedido) a um objeto, e o objeto sabe o que fazer com esta mensagem (executa o código da função).

Aqui, o nome da classe é Lampada, o nome do objeto em particular é lamp, e os pedidos que se pode fazer são: acender a lâmpada, apagá-la, aumentar a luminosidade e diminuir a luminosidade. Cria-se um objeto definindo uma "referência" (lamp) para aquele objeto e chamando um pedido de um novo objeto daquele tipo. Para enviar uma mensagem ao objeto, escreve-se o nome (referência) do objeto e conecta o mesmo ao pedido com um ponto.

1.2 Encapsulamento

Torna-se importante neste ponto fazer uma distinção entre *criadores de classes* e *usuários de classes*. O objetivo do usuário é conseguir uma série de classes para o uso no desenvolvimento rápido de aplicações. O objetivo do criador de classes é desenvolver uma série de classes que expõe somente o necessário ao usuário. Esta prática é útil pois, se o usuário não pode ver, também não pode mudar. Isto significa que o criador pode modificar a parte oculta do código sem se preocupar com o impacto que isto terá sobre todos os clientes (desde, é claro, que mantenha a funcionalidade). A parte oculta geralmente representa o interior dos objetos, que poderia ser facilmente corrompido por um usuário mal informado, sendo assim, o encapsulamento reduz os bugs no programa.

Em qualquer relação é importante haver fronteiras que sejam respeitadas por todas as partes envolvidas. Quando o programador cria uma biblioteca[2], ele estabelece uma relação com o usuário, que também é um programador, mas um que está criando uma aplicação usando a biblioteca do primeiro, possivelmente para construir uma biblioteca maior.

Se todos os membros de uma classe estivessem visíveis a todos, o usuário poderia fazer tudo com aquela classe, e não há maneira de estabelecer regras. Assim, a principal razão para o controle de acesso é para manter o cliente longe das partes que são necessárias para o funcionamento interno da classe, mas não são parte da interface que o usuário utiliza para resolver seus problemas. Isto é também um serviço aos usuários, já que eles podem ver facilmente o que é importante para eles e ignorar o restante.

Outra razão para o controle de acesso é permitir ao criador da biblioteca mudar o funcionamento interno das classes sem se preocupar com o efeito para o usuário. Um programador pode desenvolver uma classe que implementa determinada solução, e depois modificá-la para fazê-la mais rápida. Se a interface e a implementação estão separadas e protegidas, isso pode ser conseguido facilmente.

[2]Uma biblioteca é um conjunto de classes

Java utiliza três palavras-chave para delimitar os acessos em uma classe: **public**, **private** e **protected**. **public** significa que as definições estão disponíveis para todos. **protected** determina que ninguém pode acessar aquelas definições, a não ser o criador da classe, dentro de funções daquela classe. **private** não permite acesso exterior, e funciona como protected, exceto que as especializações da classe herdam os membros "protected", mas não os "private". Se não for definido um delimitador, Java especifica o modo "friendly", que significa que, para as classes do mesmo pacote os membros aparecerão como "public", mas para fora do pacote, serão "private".

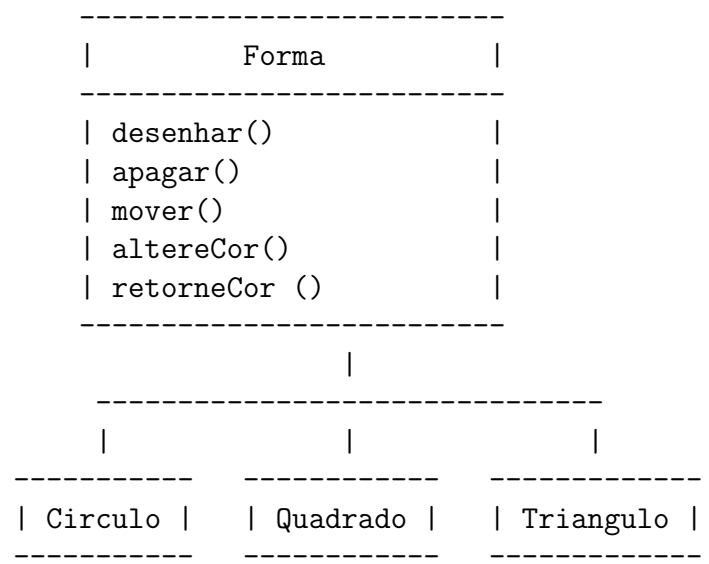
1.3 Agregação

Quando uma classe foi criada, ela deve representar uma parte utilizável do código. A maneira mais simples de reutilizar uma classe é usar um objeto daquela classe diretamente, mas também se pode colocar um objeto daquela classe dentro de uma nova classe. A nova classe será composta por qualquer número e tipo de outros objetos, em qualquer combinação necessária para atingir a funcionalidade desejada. Este conceito é chamado agregação (ou composição). A relação de agregração é chamada de "tem um", como em "um carro *tem um* motor".

1.4 Herança

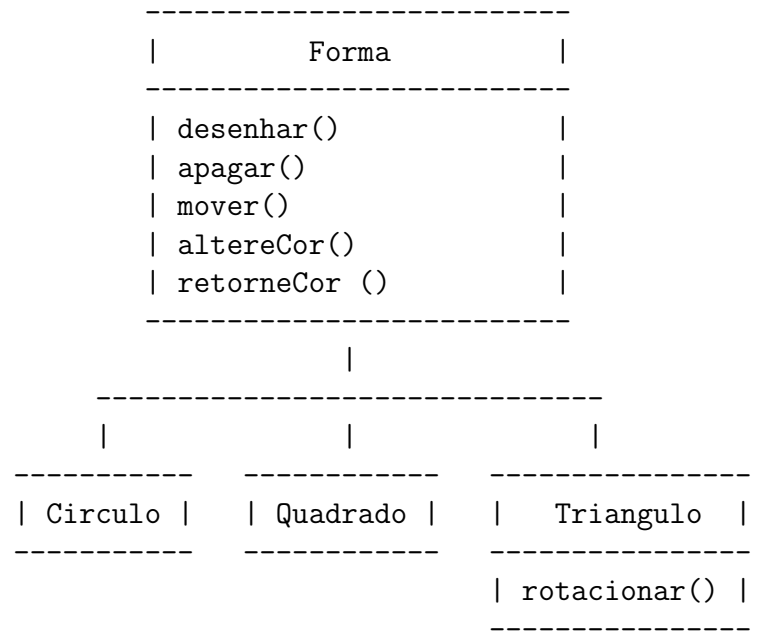
Uma classe faz mais do que descrever o funcionamento de um conjunto de objetos; ela também tem relações com outras classes. Duas classes podem ter características e comportamento em comum, mas uma pode lidar com mais mensagens (ou lidar com elas de forma diferente). A herança expressa esta similaridade entre os tipos usando o conceito de tipos base e tipos derivados. Um tipo base contém todas as características e comportamentos que são compartilhados entre os tipos derivados dele. Um exemplo clássico é o da "forma", que poderia ser usado em um sistema de CAD[3]. O tipo base (ou superclasse) é forma. Cada forma tem tamanho, cor, posição e assim por diante. Cada forma pode ser desenhada, apagada, movida, colorida, etc. Outros tipos específicos podem ser derivados (especializados) desta classe: circulo, quadrado, triangulo, e assim por diante, cada um dos quais pode ter comportamentos e características adicionais. Algumas formas podem ser rotacionadas, por exemplo. Alguns comportamentos podem ser diferentes, como para calcular a área de uma forma.

[3]Computer-Aided Design, Design Auxiliado por Computador



Quando se especializa uma classe existente, cria-se uma nova classe. Esta nova classe contém não só os membros (atributos) da classe base, mas também sua interface. Ou seja, todas as mensagens que podem ser mandadas para a classe superior também podem ser enviadas para a classe derivada. A classe derivada é do mesmo tipo da classe base.

Como tanto a classe base como a derivada tem a mesma interface, deve haver alguma implementação para executar quando um objeto recebe uma mensagem. Se simplesmente deriva-se uma classe e não se faz mais nada, os métodos da classe base são derivados também. Isto significa que os objetos da classe base e derivada não só tem o mesmo tipo, mas também o mesmo comportamento, o que não tem muita utilidade prática. Há duas maneiras de diferenciar a classe derivada da classe base. A primeira, é adicionar novas funções à classe especializada.



Embora o conceito de herança implicar às vezes (especialmente em Java, onde a palavra-chave que indica herança é **extends**) que serão adicionadas novas funções à interface, isto nem sempre é verdade. A segunda maneira

de diferenciar a classe derivada é mudar o comportamento de uma função existente na classe base. Esta operação é chamada *sobrescrever* (overriding). Para sobrescrever uma função, simplesmente cria-se uma nova definição para a função na classe derivada.

A relação de especialização é chamada de "é um", como em "um círculo *é uma* forma".

1.5 Classes Abstratas e Interfaces

Muitas vezes pode ser desejável que uma superclasse apresente somente a interface para suas classes derivadas. Ou seja, ninguém poderá criar um objeto daquela superclasse, ela só apresenta os métodos que podem ser tratados por suas subclasses. Isto é conseguido usando a palavra-chave **abstract**. Se alguém tentar criar um objeto de uma classe abstrata, o compilador acusará um erro. A palavra **abstract** também pode ser usada para descrever um método da interface da superclasse que ainda não foi implementado pela subclasse.

A palavra-chave **interface** leva o conceito de classe abstrata um passo adiante, impedindo que qualquer função da classe seja definida. Esta é uma ferramenta poderosa e muito utilizada, já que permite a separação perfeita entre a interface e a implementação. Além disso, é possível combinar várias interfaces ao mesmo tempo, sendo que a herança de múltiplas classes ou múltiplas classes abstratas não é possível.

Capítulo 2

O Básico de Java

A linguagem de programação Java era originalmente chamada Oak e foi desenvolvida inicialmente por James Gosling para ser usada em aparelhos eletrônicos. Depois de alguns anos de pesquisa a linguagem foi redirecionada para a Internet, ganhou um novo nome e evoluiu até o ponto onde hoje se encontra[1].

Apesar de ter realmente um forte apelo em relação à Internet, a linguagem e a tecnologia Java não se restringem aos famosos "applets". Java é uma linguagem orientada a objetos independente de plataforma, robusta, segura, *multithreaded*, distribuída e, apesar disto tudo, bastante simples. Há edições de Java especiais para dispositivos eletrônicos como celulares e cartões eletrônicos (Java Micro Edition) e também para aplicações de grande porte (Java 2 Enterprise Edition). Este curso tratará da plataforma Java 2 Standart Edition (J2SE)[2], que é ideal para aplicações via Internet e convencionais de médio porte.

2.1 Compilação e Execução

Cada máquina, em conjunto com o sistema operacional, tem uma maneira diferente de fazer a execução de seus programas. Sendo assim, os compiladores de linguagens como C++ e Object Pascal geram código executável específico para a máquina e o sistema operacional no qual estão sendo usados. É por este motivo que os arquivos executáveis de um sistema Windows (.exe) não rodam, por exemplo, em um sistema UNIX. Em teoria, para gerar um executável para outro sistema bastaria recompilar o código usando um compilador específico para o sistema em questão, mas isto nem sempre é possível quando são usadas livrarias específicas para um sistema, como as para interfaces gráficas.

O compilador Java não gera um executável específico para um sistema, mas sim um código intermediário (bytecode), que é interpretado pela Máquina Virtual Java (JVM). Um programa Java compilado (arquivo cuja extensão é .class) pode ser executado no ambiente Windows da Microsoft pois existe

[1] JDK 1.4.1, na data da última revisão deste curso. Os conceitos e exemplos deste curso devem ser compatíveis com todas as implementações de Java a partir da versão 1.2

[2] A plataforma Java tem dois componentes: A Máquina Virtual Java (Java VM) e a Interface para Programação de Aplicações (Application Programming Interface - Java API)

uma JVM que traduz as instruções em bytecode para o Windows. A mesma lógica aplica-se para outros sistemas operacionais (Linux, AIX, Solaris, Mac OS, etc.).

A partir do conceito de JVM, elimina-se o problema de se ter uma versão de um determinado software para cada um dos sistemas operacionais existentes no mercado ou dentro de uma corporação. Isto, naturalmente, reduz os custos de produção e manutenção de software.

Para um exemplo simples de compilação e execução de um programa em Java, consulte o Apêndice A

2.2 Anatomia de uma Aplicação Java

Vamos analisar a seguinte aplicação Java (as instruções para compilação e execução estão no Apêndice A):

```
1  /**
2   * A classe OlaJava implementa uma aplicação que
3   * simplesmente mostra "Ola Java !" na tela de console.
4   */
5
6  public class OlaJava
7  {
8      public static void main(String[ ] args)
9      {
10         System.out.println("Ola Java !");
11     }
12 }
```

As quatro primeiras linhas são de comentário. Java suporta três tipos de comentário:

```
/* este é um comentário multilinha */

// este é um comentário de uma linha

/**
 * Comentário para o Javadoc (Documentação)
 */
```

A linha 6 contém a declaração de classe. Neste caso, a classe é chamada "HelloJava"e tem acesso público. A maneira mais simples de declarar uma classe é:

```
1  class NomeDaClasse {
2          // aqui vai o código
3  }
```

A chave na linha 7 inicia um bloco de código (neste caso, da própria classe). Em Java, bem como em C, os blocos de código iniciam com `{` e terminam com `}`. O `{ ... }` do Java é equivalente ao `begin ... end` do Pascal.

A linha 8 inicia a definição do método **main**. Toda aplicação Java deve ter um método cuja assinatura é assim

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

Nesta assinatura:

public indica que o método main pode ser chamado por qualquer objeto.

static indica que o método main pode ser chamado mesmo que nenhuma instância da classe tenha sido criada.

void indica que o método main não retorna nenhum valor.

Os argumentos de um método em Java sempre vão na assinatura após o nome do método, dentro de parênteses. A declaração do argumento é sempre do tipo `(TipoDoArgumento nomeDoArgumento)`. Múltiplos argumentos são separados por vírgula, como em `(Tipo1 arg1, Tipo2 arg2)`. Métodos sem argumento devem ter na assinatura um `()` após o nome. No caso específico do método main, o argumento é um array de Strings, que pode ser utilizado para execução com opções de inicialização.

O método main em Java é similar à função main de C e C++. Quando o interpretador Java executa uma aplicação, ele inicia chamando o método main (principal) da classe da aplicação. Só a classe da aplicação deve ter um método main. Se você tentar invocar o interpretador Java em uma classe que não tem um método main ele acusa um erro.

A linha 9 inicia o bloco de código do método main. A linha 10 usa a variável `out` classe `System`, do API do Java, para mostrar uma mensagem na tela. Esta variável também é um objeto, do tipo `PrintStream`. A classe `PrintStream` possui um método estático chamado `println`, que mostra um string na tela de console. Note que nenhum objeto precisou ser criado nesta chamada, isto porque tanto a variável `out` quanto o método `println` são estáticos (static).

As linhas 11 e 12 fecham os blocos de código do método main e da classe, respectivamente.

2.3 Variáveis

2.3.1 Declaração de Variáveis

Uma variável é um dado referenciado por um identificador. Cada variável tem um tipo, como int ou Object, e um escopo[3]. O identificador é o nome da variável. Identificadores funcionam de maneira similar aos ponteiros do C. Quando chamados, eles referenciam o local da memória onde o objeto denotado por ele está localizado[4]. A seguir estão exemplos de declaração de variáveis em Java.

```
1 int i;
2 Object umObjeto;
3 String umTexto;
4 char caracter;
5 float real1,real2;
```

[3]Escopo é o alcance da variável

[4]Isto e válido para as variáveis de objetos. As variáveis de tipos primitivos guardam valores, não referências

A declaração sempre segue o esquema `TipoDaVariavel nomeDaVariavel`. É possível fazer a declaração de múltiplas variáveis, como na linha 5, onde são declaradas duas variáveis do tipo float(ponto flutuante de precisão simples).

2.3.2 Inicialização de Variáveis de Objetos

Em Java, as variáveis podem ser inicializadas tanto no momento da declaração quanto em algum ponto após a declaração. A primeira opção é mais recomendável, pois dá ao programador maior controle sobre a variável. No código a seguir estão exemplos de inicialização de variáveis de objetos, que são inicializadas por um **construtor**.

```
1  // declaração e inicialização da variável
2  NomeDaClasse nomeDoObjeto = new NomeDaClasse();
3
4
5  // declaração da variável
6  Object umObjeto;
7  .
8  .
9  // inicialização da variável
10 umObjeto = new Object();
11
12
13 // declaração e inialização da variável sem parâmetros
14 String umTexto = new String();
15
16
17 // declaração e inialização da variável com parâmetros
18 String outroTexto = new String("Olá, Enfermeira!");
```

O construtor é chamado através do comando `new NomeDaClasse`. Como pode-se ver pelas linhas 14 e 18, uma classe pode ter mais de um construtor, com diferentes parâmetros. O construtor da classe String usado na linha 14 não tem parâmetros, e cria um string em branco. Já o da linha 18 toma como parâmetro uma seqüência de caracteres, que se torna o conteúdo referenciado por aquela variável. O construtor, em última instância é um método, lembre-se que mesmo métodos que não tem argumentos devem ser chamados com um `()` após seu nome, como em `new String()`.

Caso Especial

A classe String tem um construtor especial, que não precisa ser chamado com a palavra-chave `new`. O código a seguir demonstra uma chamada do construtor especial da classe String.

```
1  // a chamada
2  String nomeDoLivro = "Cem Anos de Solidão";
3
4  // tem o mesmo efeito de
5  String nomeDoLivro = new String("Cem Anos de Solidão");
6
7  // Este é um caso especial, não vale para outras classes.
```

Os Arrays em Java também têm um construtor especial, que será explicado na Seção 2.6.

2.3.3 Inicialização de Variáveis de Tipos Primitivos

Tipos primitivos são tipos especiais, como número inteiros e em ponto flutuante, caracter e booleano. Estes tipos não são objetos[5], portanto não são inicializados através de construtores. O código a seguir mostra exemplos de inicialização de tipos primitivos.

```
1  // declaração e inicialização da variável
2  int   umInteiro        = 5000;
3  long  umInteiroGrande  = 100000000000000;
4  short umInteiroPequeno = 5;
5  float umReal           = 5.5f; /* o f depois do número
6                                    indica precisão simples */
7  double outroReal       = 4.23423423;
8  char umCaracter        = 'a';
9  boolean flag           = false;
10
11
12 // declaração da variável
13 int i;
14 .
15 .
16 // inicialização da variável
17 i = 10;
```

2.3.4 Destruição de Objetos

Os objetos criados em Java não precisam ser destruídos, um objeto será eliminado da memória automaticamente pelo "garbage collector" (coletor de lixo) quando não houver mais referências para ele.

2.3.5 Escopo

O escopo de uma variável é a região de um programa na qual a variável pode ser referenciada simplesmente por seu nome. Além disto, o escopo também determina quando o sistema cria e destrói as variáveis. Escopo não é o mesmo que visibilidade, a visibilidade é acertada com um modificador de acesso (public, protected, private).

O escopo de uma variável é determinado pelos blocos de código `{ ... }`. A variável estará acessível somente dentro do bloco de código no qual ela foi declarada ou dentro de blocos que estão no bloco no qual ela foi declarada. O código a seguir mostra alguns exemplos.

```
1  public class whatever
2  {
3    /* esta variável estará acessível
4      em todos os pontos desta classe */
5    protected int i = 5;
6
7    public void doSomething()
8    {
9      // esta variável estará acessível
10     // somente neste método.
```

[5]Ainda assim, cada um destes tipos tem classes que os representam como objetos, chamadas "wrapers"

```
11      char umaLetra = 'a';
12      if (true)
13      {
14        // umaLetra pode ser acessado aqui
15        umaLetra = 'b';
16        /* umNumero é acessível somente
17        dentro do if */
18        int umNumero = 5;
19      }
20      umNumero = umNumero + 1; //isto não funciona.
21    }
22
23  }
```

2.3.6 Variáveis Finais

A palavra-chave **final**, quando precede a declaração de uma variável, impede que a variável tenha seu conteúdo alterado.

```
1  // esta variável não varia nunca !
2  final int variavelFinal = 2;
```

2.3.7 Exercícios

Corrija, compile e execute o código a seguir.

```
1  public class Variaveis {
2
3    protected int = 3;
4
5    protected Object umObjeto = "Isto é um Objeto";
6
7    protected char c = "Lucas"
8
9    public static void main(String[ ] args) {
10     metodo1();
11     metodo2();
12     int tempoDeJogo = 50;
13   }
14
15   public static void metodo1() {
16     String guga = "Gustavo Kuerten";
17     String sampras = 'Pete Sampras';
18     int guga = 1;
19     int samp = 0;
20     System.out.println("Tempo de Jogo: "+tempoDeJogo+" min.");
21     System.out.println(guga+" "+guga+" x "+samp+" "+sampras);
22   }
23
24   public static void metodo2() {
25     float num = 0.23423;
26     System.out.println("Imprimindo um caracter... "+c);
27   }
28
29 }
```

2.4 Operadores

2.4.1 Operadores Aritméticos

Os operadores aritméticos se aplicam aos tipos primitivos numéricos. Sua funcionalidade está descrita a seguir.

Operadores básicos:

Operador	Uso	Descrição
+	var1 + var2	Soma var1 e var2
-	var1 - var2	Diminui var2 de var1
*	var1 * var2	Multiplica var1 por var2
/	var1 / var2	Divide var1 por var2
%	var1 % var2	Resto da divisão de var1 por var2

Sempre que uma variável inteira e uma em ponto flutuante são usadas como operandos, o resultado é em ponto flutuante.

Além da forma binária, os operadores + e - tem uma versão unária:

Operador	Uso	Descrição
+	+var	Transforma var em int se var for byte, short ou char
-	-var	Transforma var em -var

Também existem operadores de incrementação / decrementação:

Operador	Uso	Descrição
++	var++	Incrementa var em 1, avalia o valor de var antes de decrementar
++	++var	Incrementa var em 1, avalia o valor de var depois de incrementar
--	var--	Decrementa var em 1, avalia o valor de var antes de decrementar
--	--var	Decrementa var em 1, avalia o valor de var depois de decrementar

2.4.2 Operadores Condicionais e Relacionais

Um operador relacional compara dois valores e determina a relação entre eles. Por exemplo, ! = retorna verdadeiro se os dois operandos são diferentes. A tabela seguinte sumariza os operadores relacionais:

Operador	Uso	Retorna True (verdade) se
>	var1 > var2	var1 é maior do que var2
>=	var1 >= var2	var1 é maior/igual do que var2
<	var1 < var2	var1 é menor do que var2
<=	var1 <= var2	var1 é menor/igual do que var2
==	var1 == var2	var1 e var2 são iguais
!=	var1 != var2	var1 e var2 são diferentes

Operadores relacionais são usados com operadores condicionais para construir expressões de decisão mais complexas. Seguem os operadores relacionais suportados por Java:

Operador	Uso	Retorna True (verdade) se
&&	var1 && var2	var1 e var2 são True (verdade)
||	var1 || var2	var1 ou var2 são True (verdade)
!	! var	var é False (falso)
&	var1 & var2	var1 e var2 são True
|	var1 | var2	var1 ou var2 são True
^	var1 ^ var2	var1 e var2 são diferentes (var1 xor var2)

2.4.3 Operadores de Atribuição

Operador	Uso	Equivalente a
+=	var1 += var2	var1 = var1 + var2
-=	var1 -= var2	var1 = var1 - var2
*=	var1 *= var2	var1 = var1 * var2
/=	var1 /= var2	var1 = var1 / var2
%=	var1 %= var2	var1 = var1 % var2
&=	var1 &= var2	var1 = var1 & var2
|=	var1 |= var2	var1 = var1 | var2
^=	var1 ^= var2	var1 = var1 ^ var2

2.4.4 Exercícios

Calcule o resultado das seguintes operações, considerando:

```
boolean var1 = true;
boolean var2 = false;
int var3 = 1;
int var4 = 10;

1) var1 && (var3 <= var4)
2) (var1 && var2) || !var1
3) !(!(var3 < var4) && !(var1 || var2))
4) var1 ^= (var3 > var4)
```

2.5 Controle de Fluxo

2.5.1 while e do-while

O **while** é usado para executar continuamente um bloco de código enquanto uma condição permanecer verdadeira. Sua sintaxe geral é

```
1 while (condição) {
2     // aqui vai o código
3 }
```

O **do-while** é semelhante ao while, a diferença é que ele avalia a condição depois de executar a primeira vez. É parecido com o repeat do Pascal. Sua sintaxe é:

```
1 do {
2     // aqui vai o código
3 } while (condição);
```

2.5.2 for

O **for** é uma maneira compacta de repetir um bloco de código n vezes. Sua sintaxe geral é:

```
1 for (inicio; fim; incremento) {
2     // aqui vai o código
3 }
4
5 // por exemplo:
6
7 for (int i = 0; i < 10; i++) {
8     // aqui vai o código
9 }
```

2.5.3 if/else

O **if** permite executar um bloco de código condicionalmente. A sintaxe do if em Java é:

```
1 if (condição) {
2     // código a executar se a condição for verdadeira
3 }
```

```
4
5  if (condição) {
6      // código a executar se a condição for verdadeira
7  } else {
8      // código a executar se a condição for falsa
9  }
10
11 condição ? verdadeira : falsa
```

A última linha apresenta uma versão compacta do if, e poderia ser usado como no exemplo a seguir:

```
1  System.out.println("O caracter " + umChar + " é " +
2             umChar.ehMaiusculo ? "maiúsculo" : "minúsculo"));
```

2.5.4 switch

O **switch** é usado para executar um bloco de código condicionalmente baseado no valor de uma expressão inteira. Os exemplos a seguir mostram possíveis usos para o switch.

```
1  public class SwitchDemo {
2      public static void main(String[] args) {
3          int mes = 8;
4          switch (mes) {
5              case 1:  System.out.println("Janeiro"); break;
6              case 2:  System.out.println("Fevereiro"); break;
7              case 3:  System.out.println("Março"); break;
8              case 4:  System.out.println("Abril"); break;
9              case 5:  System.out.println("Maio"); break;
10             case 6:  System.out.println("Junho"); break;
11             case 7:  System.out.println("Julho"); break;
12             case 8:  System.out.println("Agosto"); break;
13             case 9:  System.out.println("Setembro"); break;
14             case 10: System.out.println("Outubro"); break;
15             case 11: System.out.println("Novembro"); break;
16             case 12: System.out.println("Dezembro"); break;
17         }
18     }
19 }
20
21
22
23 public class SwitchDemo2 {
24     public static void main(String[] args) {
25         int mes = 2;
26         int ano = 2000;
27         int numDias = 0;
28         switch (mes) {
29             case 1:
30             case 3:
31             case 5:
32             case 7:
33             case 8:
34             case 10:
35             case 12:
36                 numDias = 31;
37                 break;
```

```
38              case 4:
39              case 6:
40              case 9:
41              case 11:
42                  numDias = 30;
43                  break;
44              case 2:
45                  if ( ((ano % 4 == 0) && !(ano % 100 == 0))
46                       || (ano % 400 == 0) )
47                      numDias = 29;
48                  else
49                      numDias = 28;
50                  break;
51          }
52          System.out.println("Número de Dias = " + numDias);
53      }
54  }
```

2.5.5 Tratamento de Exceções

Java tem um mecanismo de tratamento de exceções que ajuda os programas a reportar e tratar erros. Quando um erro ocorre, o programa "joga uma exceção". Quando isto ocorre, o fluxo normal do programa é interrompido e é executado um tratador de exceção, um bloco de código que lida com determinado tipo de erro. A sintaxe geral do tratamento de exceções em Java é:

```
1  try {
2      // código
3  } catch (TipoDaExcecao nomeDaExcecao) {
4      // este bloco é executado se ocorrer uma exceção
5  }
6
7
8  try {
9      // código
10 } catch (TipoDaExcecao nomeDaExcecao) {
11     // este bloco é executado se ocorrer uma exceção
12 } finally {
13     // este bloco é executado incondicionalmente
14 }
```

2.5.6 Controle de Execução

break

O comando **break** serve para terminar um loop. Sua forma geral segue o seguinte exemplo:

```
1  for (int i = 0; i < 10; i++) {
2    if (aconteceuAlgo) {
3      facaAlgo();
4      break;
5    }
6  }
```

continue

O comando **continue** serve para ignorar a iteração atual de um loop.

```
1  for (int i = 0; i < 10; i++) {
2    facaAlgoPrimeiro();
3    if (aconteceuAlgo) {
4      continue; // o resto desta iteração do loop será ignorada.
5    }
6    facaAlgo();
7    facaOutraCoisa();
8  }
```

return

Todo método em Java tem um valor de retorno, expresso na assinatura. Este valor pode ser um Objeto, um tipo primitivo, ou um valor do tipo **null**, que indica que o método não retorna nenhum valor significativo.

O comando **return** tem duas funções. Ele indica o valor que será retornado pelo método, e pára a execução do mesmo. Os exemplos a seguir mostram a forma de uso do return.

```
1  public int metodo1() {
2    int i,j;
3    j = 0;
4    for (i = 0; i < 10; ++i) {
5      j += i;
6      if (j == i)
7        continue;
8      if (j == 100)
9        break;
10     if (j == 1000)
11       return (i + 1);
12     j++;
13   }
14   return j;
15 }
16
17 public void metodo2() {
18   if (condição) return;
19   outrometodo();
20 }
```

2.5.7 Exercícios

Traduza o seguinte pseudo-código para Java:

```
quant = 0
para i = 0..10
  para j = 5..100
    se i != j
      não execute esta iteração
    fim se
    quant = i * j + quant
  fim para j
  k = 0
  enquanto k != i
```

```
    quant = quant / (k + i)
    k = k + 1
  fim enquanto
fim para i
retorne quant
```

2.6 Arrays

Um Array é uma estrutura que contém vários dados de um mesmo tipo. Arrays podem guardar tanto objetos quanto tipos primitivos. Em Java, o tamanho dos Arrays é definido na criação.

2.6.1 Declaração, Criação e Acesso

A declaração de variáveis de Array em Java segue a seguinte estrutura:

```
1  int[ ] arrayDeInteiros;
2  char[ ] arrayDeCaracteres;
3  Object[ ] arrayDeObjetos
4  UmaClasseQualquer[ ] arrayDeUmaClasseQualquer;
5
6  // A declaração também pode seguir a forma
7  // Tipo variável[ ], como em
8  int arrayDeInts[ ];
9  char arrayDeChars[ ];
```

Como já foi mencionado, os Arrays em Java têm construtores especiais. Os exemplos a seguir mostram a criação de Arrays:

```
1  int[ ] arrayDeInts = { 2 , 3 , 4 };
2
3  // é equivalente a
4  int[ ] arrayDeInts = new int[3];
5  arrayDeInts[0] = 2;
6  arrayDeInts[1] = 3;
7  arrayDeInts[2] = 4;
8
9  // e também é igual a
10 int arrayDeInts[ ];
11 arrayDeInts = new int[3];
12 arrayDeInts[0] = 2;
13 arrayDeInts[1] = 3;
14 arrayDeInts[2] = 4;
15
16 // Isto também é válido para arrays de Objetos:
17 Object array[ ] = {
18   new Object(),
19   new Object()
20 }
```

O acesso a uma determinada posição de um array sempre segue a sintaxe `nomeDoArray[posicaoAAcessar]`. Como os índices do array começam no zero, o valor máximo que se pode acessar é tamanho - 1. A variável `length` dá o tamanho do array. Se for tentado o acesso à uma posição inexistente, será gerada uma exceção do tipo `ArrayIndexOutOfBounds`.

2.6.2 Exercícios

1. Qual é o índice de `Hommer` no seguinte Array?

```
String[] simpsons {
  "Bart", "Lisa", "Hommer", "Marge", "Meg"
};
```

2. Escreva uma expressão que se refira ao string `Lisa` no Array.
3. Qual é o valor da expressão `simpsons.length` ?
4. Qual é o índice o último item do array ?
5. Qual é o valor da expressão `simpsons[3]` ?

2.7 Exercícios de Revisão

1. Crie uma aplicação que mostre na tela de console o conteúdo de um Array de Strings, como o do exercício na Seção 2.6.2.

2. Corrija e compile o código a seguir:

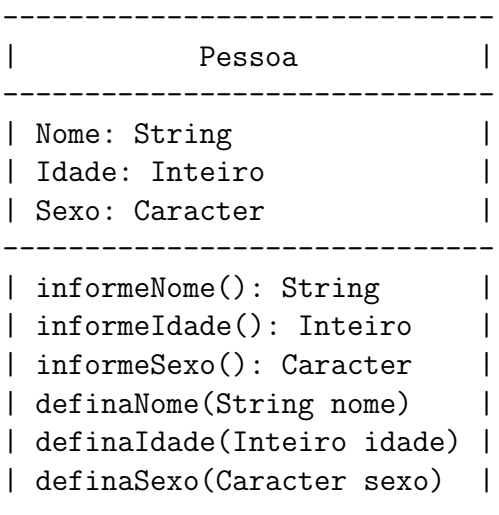
```
/ classe de exercícios.
public Class umaClasse {

  public void main (string[] args) {
    metodo1;
    metodo2;
  }

  public static void metodo1() {
    int = 5;
    for (i = 0; i < max; i++) {
      System.Out.println(Uma mensagem qualquer)
  }

  public static void metodo2 {
    variavel = 5.5f;
    System.Out.println(Outra mensagem qualquer)
  }

}









}
```

Capítulo 3

Orientação a Objetos em Java

3.1 Classes

Programar em Java é criar classes, e o primeiro passo para a construção de classes é o planejamento. Qual são os serviços (métodos) da classe ? Quais são seus atributos ? Um exemplo pode ser uma classe que represente uma pessoa:



No diagrama acima, a primeira parte representa o nome da classe. Na segunda divisão estão os atributos e seus tipos. Na terceira parte estão os métodos implementados pela classe. Os valores entre parênteses são os argumentos requeridos pelo método. Quando não há argumentos, os parênteses estão vazios. O tipo do valor que é retornado pelo método está depois dos dois pontos. Quando não há valor de retorno (void), os dois pontos são omitidos.

A classe Pessoa somente implementa métodos de acesso aos seus atributos. Métodos de acesso (também chamados "getters" e "setters") são funções que permitem obter e alterar os valores dos atributos de uma classe. Por estarem sempre presentes, os métodos de acesso normalmente são omitidos de diagramas como o anterior.

A segunda etapa de construção de uma classe é a tradução para o código. Em Java, a classe Pessoa pode ser implementada da seguinte forma:

```
1  /**
2  * Classe que representa uma Pessoa
3  */
4  public class Pessoa
5  {
6          // aqui começa a declaração dos atributos
7          // os atributos devem ser sempre protegidos!!
8
9          protected String nome;
10         /* a variável nome é um String protegido
11            (só pode ser acessado dentro da classe)*/
12         protected int idade;
13         protected char sexo;
14         /* os outros atributos também são protegidos,
15            mas são tipos primitivos (int e char) */
16
17         // aqui começam os métodos da classe
18
19         /* o método informeNome é publico,
20            retorna um String, e não tem argumentos */
21         public String informeNome() {
22                 // a única coisa que este método faz é
23                 // retornar a variável nome.
24                 return(nome);
25         }
26
27         public int informeIdade() {
28                 return(idade);
29         }
30
31         public char informeSexo() {
32                 return sexo;
33         }
34
35
36         /* o método definaNome é publico,
37            não tem nenhum valor de retorno
38            e toma um String como argumento */
39         public void definaNome (String nome)
40         {
41                 // este método define o attributo
42                 // nome desta classe (this.nome) como
43                 // o nome que foi informado como parâmetro
44                 this.nome = nome;
45         }
46
47         public void definaIdade (int idade) {
48                 this.idade = idade;
49         }
50
51         public void definaChar (char sexo) {
52                 this.sexo = sexo;
53         }
54
55         // aqui são definidos os construtores da classe
56
57         // O Primeiro construtor não tem argumentos
58         public Pessoa ()
59         {
```

```
60                  // como não foram definidos argumentos,
61                  // os atributos recebem valores nulos.
62                  this.nome = "";
63                  this.idade = 0;
64                  this.sexo = 'I';
65          }
66
67          // o segundo construtor toma como argumento
68          // somente o nome da pessoa;
69          public Pessoa (String nome) {
70                  this.nome = nome;
71                  this.idade = 0;
72                  this.sexo = 'I';
73          }
74
75          // um terceiro construtor toma como argumento
76          // todos os dados da pessoa
77          public Pessoa (String nome, int idade, char sexo) {
78                  this.nome = nome;
79                  this.idade = idade;
80                  this.sexo = sexo;
81          }
82  }
```

Um exemplo um pouco mais complexo é o de uma fila do tipo FIFO[1]:

```
------------------------------
|            Fila            |
------------------------------
| elementos: Object[]        |
| tamanho: int               |
------------------------------
| inclua(Object elemento)    |
| remova(): Object           |
| informeTamanho():int       |
------------------------------
```

Os atributos da classe são um array de Objetos e um valor inteiro que indica o tamanho atual da fila. Seus métodos são `inclua`, que inclui um objeto no final da fila, e `remova`, que remove e retorna o elemento que está no início da fila. Também há um método que informa o tamanho atual da fila. A implementação fica assim:

```
1   /**
2   * Classe que implementa uma fila FIFO
3   */
4   public class Fila
5   {
6     protected Object[] elementos;
7     protected int tamanho;
8
9     /**
10     * Cria uma nova fila, de tamanho máximo 100
11     */
```

[1]First in, First Out - O primeiro que entrou é o primeiro a sair

```
12    public Fila() {
13       tamanho = 0;
14       elementos = new Object[100];
15    }
16
17    /**
18     * Cria uma nova fila, de tamanho
19     * máximo indicado no argumento
20     */
21    public Fila(int tamanhoMax) {
22       tamanho = 0;
23       elementos = new Object[tamanhoMax];
24    }
25
26    /**
27     * Retorna o tamanho da fila
28     */
29    public int informeTamanho() {
30       return tamanho;
31    }
32
33    /**
34     * Remove e retorna o primeiro elemento da fila
35     */
36    public Object remova() {
37       // guarda o primeiro elemento da fila.
38       Object primeiro = elementos[0];
39       // desloca todos os elementos uma posição
40       // para a esquerda.
41       for (int i = 0;i < tamanho - 1 ; i++ ) {
42          elementos[i] = elementos[i+1];
43       }
44       // diminui o tamanho da fila
45       tamanho--;
46       // retorna o elemento que foi guardado
47       return(primeiro);
48    }
49
50    /**
51     * Inclui um elemento na fila
52     */
53    public void inclua(Object elemento) {
54       // inclui o elemento na última posição
55       elementos[tamanho] = elemento;
56       // aumenta o tamanho da fila
57       tamanho++;
58    }
59
60 }
```

3.1.1 Exercícios

Construa uma classe que implemente uma Pilha (fila do tipo LIFO[2]). Os métodos são o mesmos da fila FIFO, mas a funcionalidade interna deve ser alterada para que o elemento removido seja sempre o último elemento que foi inserido.

[2]Last In, First Out - O último que entrou é o primeiro a sair

3.2 Interfaces

Uma interface é uma série de definições de métodos sem a sua implementação. Interfaces podem ser usadas para:

- Denotar similaridades entre classes sem ter que forçar um relacionamento (como herança) entre as duas.
- Declarar métodos que uma ou mais classes devem implementar
- Revelar a interface do objeto sem revelar sua classe.

No exemplo anterior das filas, poderia ser criada uma interface Fila, na qual seriam especificados os métodos que uma fila deve implementar. Diferentes tipos de fila (LIFO, FIFO) implementariam esta interface de maneira diferente, mas todos eles ainda seriam Filas. A interface Fila poderia ser definida assim:

```
1  public interface InterfaceFila {
2
3    /**
4     * Retorna o tamanho da fila
5     */
6    public int informeTamanho();
7
8    /**
9     * Remove e retorna um elemento da fila,
10    * de acordo com a hieraquia da fila (LIFO ou FIFO)
11    */
12   public Object remova();
13
14   /**
15    * Inclui um elemento na fila,
16    * de acordo com a hieraquia da fila (LIFO ou FIFO)
17    */
18   public void inclua(Object elemento);
19
20 }
```

Construtores podem ser definidos em uma interface. Para implementar uma interface usa-se a palavra-chave `implements`:

```
1  public class FilaFIFO implements Fila {
2
3
4    // aqui vai a implementação de todos
5    // os métodos indicados na interface Fila
6
7
8  }
9
10
11
12 public class UmaClasse implements UmaInterface, OutraInterface {
13
14   // uma classe pode implementar várias interfaces
15
16 }
```

3.3 Herança

Embora nem sempre declarem explicitamente, todas as classes em Java usam herança, ou seja, todas elas tem uma superclasse, ou classe base. A classe base de Java é Object, e dela todas as outras são derivadas, direta ou indiretamente. Para derivar uma classe de outra, usa-se a palavra-chave **extends**.

Subclasses:

- Herdam os membros da superclasse que são declarados públicos (public), protegidos (protected) e que não tem especificador de acesso.
- Não herdam um membro da superclasse se a subclasse declara um membro com o mesmo nome. No caso das variáveis, a subclasse esconde a variável da superclasse, e no caso dos métodos a subclasse sobrescreve os métodos da superclasse.

A subclasse pode referir-se a um membro da superclasse através da palavra-chave **super**. Métodos da superclasse que forem declarados como **final** não podem ser sobrescritos.

O exemplo a seguir mostra a classe `Cliente`, que é uma subclasse de `Pessoa`.

```
1  public class Cliente extends Pessoa {
2
3
4    // a classe cliente tem todos os atributos
5    // de pessoa, além dos especificados abaixo
6    protected int cpf;
7    protected int numeroDeCliente;
8
9    public int informeNumeroDeCliente ()  {
10      return(numeroDeCliente);
11   }
12
13   // aqui vai o restante dos métodos da classe
14
15   // a subclasse pode reescrever os métodos
16   // da superclasse:
17   public void definaNome (String nome)   {
18      /* este método sobreescreve o método
19         da superclasse, e define o nome
20         do cliente (que é uma Pessoa)
21         como Bart Simpson, sem considerar o argumento  */
22      this.nome = "Bart Simpson";
23   }
24
25   public void definaIdade(int idade)  {
26      // este outro método chama o método da superclasse
27      // depois aumenta a idade em 1
28      super.definaIdade(idade);
29      this.idade++;
30   }
31
32
33 }
```

3.4 Classes Abstratas

Uma classe abstrata pode ser definida como um estágio intermediário entre a interface e a classe final. No exemplo das Filas LIFO e FIFO, poderia ser construída uma classe abstrata que implementasse as funções comuns às duas classes. Neste caso, a única função em comum seria `informeTamanho`, já que esta é implementada da mesma maneira nas duas classes. Os atributos em comum também podem ser definidos. A classe abstrata ficaria assim:

```
1  public abstract class FilaAbstrata implements InterfaceFila {
2
3    /* a classe abstrata pode definir os atributos comuns
4       às suas subclasses */
5    protected Object[] elementos;
6    protected int tamanho;
7
8    /* O único método definido pela classe
9       abstrata é informeTamanho()  */
10   public int informeTamanho()   {
11      return(tamanho);
12   }
13
14 }
```

Agora as subclasses devem extender a classe abstrata e implementar o restante da interface:

```
1  public class FilaFIFO extends FilaAbstrata implements InterfaceFila {
2
3    // os atributos já foram definidos na FilaAbstrata
4
5    // só é preciso implementar os métodos restantes
6
7  }
```

3.5 Tratamento de Exceções

Uma exceção é um evento que ocorre durante a execução de um programa e que interrompe o fluxo natural de instruções. Apesar de não ser um conceito da Programação Orientada a Objetos, o tratamento de exceções é uma parte fundamental de qualquer aplicação. Java tem um dos sistemas de tratamento de exceção mais eficientes entre as linguagens atuais, e proporciona uma maneira segura e relativamente simples de lidar com os erros: todas as exceções são objetos.

No exemplo anterior das Filas, há uma série de erros que podem ocorrer, mas não foram tratados:

- E se o usuário tentar remover um elemento de uma fila vazia ?
- E se o usuário tentar inserir um elemento em uma fila cheia ?

Para tratar estas possíveis falhas poderiam ser definidas duas exceções:

```
// uma exceção extende Exception
public class ExcecaoFilaCheia extends Exception {

  public ExcecaoFilaCheia () {
     // o construtor chama o construtor da superclasse
     super("**ERRO** A fila esta cheia!");
  }
}
```

```
// uma exceção extende Exception
public class ExcecaoFilaVazia extends Exception {

  public ExcecaoFilaVazia () {
     // o construtor chama o construtor da superclasse
     super("**ERRO** A fila esta vazia!");
  }
}
```

Agora é necessário adaptar o código da classe fila:

```
  /**
   * Remove e retorna o primeiro elemento da fila
   * Joga a ExcecaoFilaVazia se a fila estiver vazia
   */
  public Object remova() throws ExcecaoFilaVazia {
     // se o tamanho da fila for = 0, joga a exceção
     if (tamanho == 0)
        throw new ExcecaoFilaVazia();
     /* quando o método joga a exceção, o restante
        do código é ignorado.

     /* se não houve erros,
        guarda o primeiro elemento da fila. */
     Object primeiro = elementos[0];
     // desloca todos os elementos uma posição
     // para a esquerda.
     for (int i = 0;i < tamanho - 1 ; i++ ) {
        elementos[i] = elementos[i+1];
     }
     // diminui o tamanho da fila
     tamanho--;
     // retorna o elemento que foi guardado
     return(primeiro);
  }

  /**
   * Inclui um elemento na fila
   * Joga a ExcecaoFilaCheia se a fila estiver cheia
   */
  public void inclua(Object elemento) throws ExcecaoFilaCheia{
     /* se o tamanho da fila for igual ao tamanho do
        array, joga a exceção */
     if (tamanho == elementos.length)
        throw new ExcecaoFilaVazia();

     /* se não houver erros,
        inclui o elemento na última posição */
     elementos[tamanho] = elemento;
     // aumenta o tamanho da fila
```

```
40        tamanho++;
41      }
```

Agora é preciso adaptar o uso desta fila para "pegar" a exceção:

```
1  // antes, a fila era usada simplesmente chamando os métodos:
2  Fila fila = new Fila(100);
3  fila.inclua(umObjeto);
4  fila.inclua(outroObjeto);
5  .
6  .
7  .
8  outroObjetoAinda = fila.remova();
9
10
11 /* agora é necessário "pegar" a exceção
12    (e fazer algo com ela) */
13 Fila fila = new Fila(100);
14
15 // agora, tenta-se incluir elementos
16 try {
17   fila.inclua(umObjeto);
18   fila.inclua(outroObjeto);
19 } catch (ExcecaoFilaCheia excecao) {
20   // se ocorreu uma exceção deste tipo
21   // mostre uma mensagem na tela.
22   System.out.println("Impossível incluir elemento, a fila está cheia");
23 }
24
25 // tentando remover elementos
26 try {
27   outroObjetoAinda = fila.remova();
28 } catch (ExcecaoFilaVazia excecao) {
29   // se ocorreu uma exceção deste tipo
30   // mostre uma mensagem na tela.
31   System.out.println("Impossível remover elemento, a fila está vazia");
32 }
```

3.6 Exercícios de Revisão

Construa um aplicativo que use uma fila LIFO e outra FIFO. A interface deve definir os métodos já apresentados anteriormente, e outros dois métodos, `estaCheia()` e `estaVazia()`. Além disto, as exceções que serão jogadas pelos métodos devem ser definidas na interface. Deve ser construída uma classe abstrata, que defina métodos em comum entres as filas LIFO e FIFO, e estas devem estender a classe abstrata e implementar a interface.

A aplicação deve simplesmente incluir e remover elementos do tipo String das filas, e mostrar os elementos incluídos e removidos na tela de console. Também devem ser mostradas mensagens quando houverem exceções. Inclua documentação no formato JavaDoc.

Capítulo 4

Construção de Aplicações em Java

4.1 Pacotes

Se programar em Java é criar classes, construir aplicações em Java é usar classes. Pacotes, através dos arquivos JAR, fornecem uma maneira conveniente de distribuir e usar um conjunto de classes, como uma livraria.

4.1.1 Construção de Pacotes

Para incluir uma classe em um pacote, inclui-se a palavra-chave `package` antes do início da declaração da classe. A classe `FilaFIFO` poderia estar incluída em um pacote:

```
1 package cursojava.ed.filas
2
3 public class FilaFIFO {
4   ...
5 }
```

O pacote definido foi `cursojava.ed.filas`, ou seja, o Pacote `filas` faz parte de um pacote `ed` (estruturas de dados), que faz parte de um pacote `cursojava`. A estrutura do nome do pacote deve estar refletida na estrutura de diretórios onde estes arquivos estão localizados. Sendo assim, os arquivos `FilaFIFO.class` e `FilaFIFO.java` devem estar no diretório `/cursoJava/ed/filas`. Um conjunto de classes que implementasse diferentes tipos de árvores poderia estar localizado em `/cursoJava/ed/arvores`, e assim por diante.

A ferramenta **jar** junta todas as classes de um pacote em um único arquivo no formato zip. Para criar um arquivo jar usa-se o seguinte comando[1]:

```
jar cvf nomedopacote.jar diretorio/UmaClasse.class
                         outroDiretorio/OutraClasse.class
```

[1]Para mais informações sobre a ferramenta JAR, consulte a documentação do JDK [4].

Conforme o número de classes aumenta, pode-se tornar cansativo digitar todos os nomes cada vez que o pacote for atualizado, portanto pode ser útil manter um arquivo de lote que "empacote" todas as classes.

4.1.2 Utilização de Pacotes

Para usar uma classe que está em um pacote exterior ao atual, usa-se a palavra-chave `import`. Uma aplicação que usasse as classes contidas no pacote `cursojava.ed.filas` ficaria assim:

```
1  import cursojava.ed.filas;
2
3  public class Aplicacao {
4
5    public static void main(String[] args) {
6    }
7
8  }
```

Também é necessário informar ao compilador e ao interpretador o arquivo onde as classes do pacote estão localizadas. Supondo que o arquivo do pacote usado seja `cursojava.jar`, a sintaxe de compilação e execução ficaria assim (no Windows):

```
javac -classpath .;cursojava.jar Aplicacao.java

java -classpath .;cursojava.jar Aplicacao.java
```

Diferentes locais de classpath são separados por `;` no Windows e `:` no UNIX. O `.` indica o diretório atual.

O API do Java possui uma série de classes úteis que estão distribuídas por diversos pacotes. O uso destes pacotes segue a mesma lógica do exemplo acima. Informações estas classe estão disponíveis na documentação do Java API.

Se por alguma razão for necessário extrair as classes de um pacote usa-se o seguinte comando:

```
jar -xf pacote.jar
```

Alternativamente, pode-se usar um descompactador de arquivos zip, como o Winzip ou o unzip.

4.1.3 Exercícios

"Empacote" as classes criadas na Seção 3.6, na página 33. Inclua as diretivas package e import nas classes, crie uma estrutura de diretórios, recompile as classes, e use a ferramenta `jar` para criar o pacote.

4.2 Documentação

A documentação é peça fundamental na reusabilidade de classes. Se não houvesse a documentação detalhando o funcionamento dos métodos, como estes poderiam ser usados ? Como seriam diferenciados métodos com nomes parecidos, ou mesmo iguais em classes diferentes ?

Javadoc é uma ferramenta que compila a documentação de uma série de classes Java em páginas HTML, descrevendo as classes, construtores, métodos e atributos protegidos e públicos.

É possível usar o Javadoc em pacotes inteiros, classes individuais, ou ambos. No primeiro caso, os argumentos são uma série de nomes de pacotes. No segundo, uma série de arquivos fonte. Vale lembrar que o Javadoc processa os arquivos `.java`, e não os `.class`, portanto é necessário que estes arquivos estejam no pacote para que a documentação seja construída.

O exemplo a seguir mostra o uso básico do Javadoc[2].

```
javadoc -classpath .;umpacote.jar  -d umdiretorio/docs
cursojava.ed.filas cursojava.ed.arvores cursojava.aplicacoes
```

4.2.1 Exercícios

Use a ferramenta Javadoc para criar a documentação do pacote criado na Seção 4.1.3.

4.3 Entrada e Saída

Muitas vezes é necessário que uma aplicação carregue informações de uma fonte externa, ou guarde informações em um local externo. Este local pode ser um arquivo, um disco, um lugar na rede, etc. A informação também pode ser de qualquer tipo: objetos, caracteres, imagens ou sons. Para trazer informações externas, um programa abre uma "corrente" (*stream*) de dados e lê a informação da seguinte maneira:

```
abre uma corrente
enquanto houver mais informação
   lê informação
fecha a corrente
```

De maneira similar, pode-se enviar informações para uma fonte externa.

```
abre uma corrente
enquanto houver mais informação
   escreve informação
fecha a corrente
```

O pacote `java.io`, do Java API, tem uma série de classes para leitura e escrita de dados, tanto na forma de bytes como caracteres.

[2]Para mais informações sobre a ferramenta Javadoc, consulte a documentação do JDK [4].

4.3.1 O Básico do I/O em Java

O exemplo a seguir mostra um exemplo simples de uma aplicação que copia um texto de um arquivo para outro:

```
1  // importa todas as classes do pacote java.io
2  import java.io.*;
3
4  public class IOBasics {
5      // o método main joga a execeção IOException
6      public static void main(String[] args) throws IOException {
7
8          // define dois arquivos com a classes File
9          File arquivoEntrada = new File("c:/The Simpson's Episode List.txt");
10         File arquivoSaida = new File("d:/copia.txt");
11
12         // define um leitor e um escritor de caracteres
13         // (classes FileReader e FileWriter)
14         FileReader in =  new FileReader(arquivoEntrada);
15         FileWriter out = new FileWriter(arquivoSaida);
16
17         // caracteres podem ser representados como inteiros
18         // (o contrário não é válido)
19         int caracter;
20
21         // O caracter -1 indica que o arquivo acabou.
22         while ((caracter = in.read()) != -1)
23            // enquanto o arquivo não acabar, escreve o que foi lido.
24            out.write(caracter);
25
26         // fecha o leitor e o escritor de arquivos
27         in.close();
28         out.close();
29     }
30 }
```

Vale lembrar que Java trabalha com caracteres Unicode, o que não acontece com a maioria dos sistemas operacionais atuais, portanto pode haver diferenças entre os arquivos.

4.3.2 Arquivos de Acesso Aleatório

Classes como `FileReader` e `FileWriter` somente permitem acesso seqüencial à informação. Ainda que bastante útil, o acesso seqüencial a arquivos é uma herança de meios de estocagem antigos, como fitas magnéticas, que só permitiam este tipo de acesso. Arquivos de acesso aleatório, por outro lado, permitem aceso não seqüencial (aleatório) aos conteúdos de uma arquivo.

A classe `RandomAcessFile`, além de permitir escrita e leitura não seqüencial de arquivos, implementa uma série de facilidades para I/O, e tem métodos para leitura e escrita de caracteres, números inteiros, em ponto flutuante, etc.

O exemplo a seguir mostra o uso de alguns métodos da classe (cuja documentação completa é encontrada na documentação do Java API [3]). Nesta aplicação, considera-se que o arquivo `randomacess.dat` tem a seguinte estrutura:

1. Um número inteiro (int) que indica o número de caracteres contidos no próximo item.
2. Uma seqüencia de caracteres, de tamanho indicado no item anterior
3. Um número inteiro curto (short), que indica o número de valores em ponto flutuante contidos no próximo item
4. Uma série de números em ponto flutuantes, de tamanho indicado no item anterior

```
1  import java.io.*;
2
3  public class IOAcessoAleatorio {
4
5    public static void main(String[] args) throws Exception {
6       escrevaArquivo();
7       leiaArquivo();
8    }
9
10   // o método escrevaArquivo joga todas as exceções
11   // que possivelmente ocorrerem
12   public static void escrevaArquivo() throws Exception {
13      int numChars = 0;
14      File file = new File("c:/randomacess.dat");
15      // cria um RandomAcessFile com parâmetros: um arquivo,
16      // e indicação de que o acesso será de leitura e escrita ("rw")
17      RandomAccessFile out = new RandomAccessFile(file,"rw");
18      String texto = "The Long Dark Tea-Time Of The Soul";
19      // escreve um inteiro do arquivo, que indica o numero
20      // de caracteres do string a ser escrito
21      out.writeInt(texto.length());
22      // escreve a sequencia de caracteres
23      out.writeChars(texto);
24      float[] floats =  {
25         1.1f, 2.2f, 3.3f, 4.4f, 5.5f,
26         6.6f, 7.7f, 8.8f, 9.9f, 0.0f
27      };
28      // escreve um inteiro curto do arquivo, que indica o numero
29      // de floats que serão escritos escrito
30      out.writeShort(floats.length);
31      // escreve os floats
32      for (int i = 0; i < floats.length; i++ ) {
33         out.writeFloat(floats[i]);
34      }
35      out.close();
36   }
37
38
39   // o método leiaArquivo joga todas as exceções
40   // que possivelmente ocorrerem
41   public static void leiaArquivo() throws Exception {
42      int numChars = 0;
43      File file = new File("c:/randomacess.dat");
44      // cria um RandomAcessFile com parâmetros: um arquivo,
45      // e indicação de que o acesso será de leitura e escrita ("rw")
46      RandomAccessFile in = new RandomAccessFile(file,"rw");
47      // lê um inteiro do arquivo
```

```
48        numChars = in.readInt();
49        // lê numChars caracteres unicode e os armazena em um
50        // StringBuffer (Classe de Strings variáveis)
51        StringBuffer texto = new StringBuffer();
52        for (int i = 0; i < numChars; i++) {
53           // o método append adiciona um caracter no final do String
54           texto.append(in.readChar());
55        }
56        // lê um inteiro curto do arquivo
57        short numFloats = in.readShort();
58        float[] floats = new float[numFloats];
59        // lê numFloats valores em ponto flutuante
60        // e os armazena em um array de floats
61        for (int i = 0; i < numFloats; i++ ) {
62           floats[i] = in.readFloat();
63        }
64        in.close();
65     }
66
67
68
69  }
```

4.3.3 Serialização

A serialização (*serialization*), através das classes `ObjectInputStream` e `ObjectOutputStream`, permite ler e escrever objetos. Além de permitir estocagem de objetos no disco para o uso futuro em uma aplicação, a serialização também é usada na comunicação entre objetos via sockets, através do RMI[3].

Leitura e Escrita de Objetos

As classes `ObjectOutputStream`, `FileInputStream` juntamente com as classes `FileOutputStream`, `FileInputStream` permitem que um objeto "serializado" seja escrito e lido em um arquivo. Sua utilização é bastante simples:

```
1  FileOutputStream out = new FileOutputStream("c:/aData.dat");
2  ObjectOutputStream s = new ObjectOutputStream(out);
3  s.writeObject("Hoje");
4  s.writeObject(new Date());
5  s.flush();
6
7  FileInputStream in = new FileInputStream("c:/aData.dat");
8  ObjectInputStream s = new ObjectInputStream(in);
9  String hoje = (String)s.readObject();
10 Date data = (Date)s.readObject();
```

Classes "Serializáveis"

Somente classes "serializáveis" podem ter seus objetos escritos em arquivos. As classes serializáveis devem implementar a interface `Serializable`:

```
1  public class UmaClasseSerializavel implements Serializable {
2      // não é preciso implementar nenhum método!
3  }
```

[3]Remote Method Invocation - Invocação de Métodos Remotos

4.3.4 I/O no Console

A classe System tem como atributo os objetos `in` e `out`, das classes `InputStream` e `PrintStream`, respectivamente. Como já foi visto, a exibição de uma mensagem na tela de console pode ser feita chamando os métodos `print(String msg)` e `println(String msg)` da classe `PrintStream`. Entretanto, a leitura de mensagens ou caracteres digitadas através de um teclado não é tão simples. O método `read()` da classe `InputStream` lê somente bytes, que devem ser convertidos para caracteres, inteiros, etc. O método `read()` é bastante simplista, e dificilmente será usado diretamente, portanto é necessário criar ou usar outros métodos para fazer a leitura. A classe a seguir usa um `BufferedReader` (Classe do API Java [3]) para ler mensagens do teclado.

```
1  // a classe BufferedReader está no pacote java.io
2  import java.io.*;
3
4  public class LeitorDeTeclado {
5
6      private BufferedReader br;
7
8      public LeitorDeTeclado() {
9          // cria um novo BufferedReader, indicando
10         // que a entrada será a "default", ou seja,
11         // System.in
12         br = new BufferedReader(new InputStreamReader(System.in));
13     }
14
15     // método leitor padrão, mostra uma mensagem(argumento)
16     // e lê um string digitado
17     private String leia(String s)
18     {
19         String string = "";
20         System.out.print(s);
21         try {
22             // tenta ler uma linha
23             string = br.readLine();
24         }
25         catch(IOException ioexception) {
26             // se houver uma exceção, indica um erro e sai da aplicação
27             System.out.println("Erro, nao foi possivel ler...");
28             System.exit(1);
29         }
30         return string;
31     }
32
33     // os métodos a seguir convertem o que foi lido para
34     // um tipo específico.
35
36     public byte leiaByte(String s){
37         return Byte.parseByte(leia(s));
38     }
39
40     public int leiaInt(String s){
41         return Integer.parseInt(leia(s));
42     }
43
```

```
44      public long leiaLong(String s){
45          return Long.parseLong(leia(s));
46      }
47
48      public short leiaShort(String s){
49          return Short.parseShort(leia(s));
50      }
51
52      public double leiaDouble(String s){
53          return Double.parseDouble(leia(s));
54      }
55
56      public float leiaFloat(String s){
57          return Float.parseFloat(leia(s));
58      }
59
60      public char leiaChar(String s){
61          return leia(s).charAt(0);
62      }
63
64      public String leiaString(String s){
65          return leia(s);
66      }
67
68      // retorna true se o que foi digitado foi igual a msgtrue,
69      // e retorna false se for diferente de msgtrue.
70      public boolean leiaBoolean(String s, String msgtrue, String msgfalse) {
71          return msgtrue.equalsIgnoreCase(leia(s+" "+msgtrue+" "+msgfalse));
72      }
73
74  }
```

4.4 Threads

Um programa seqüencial tem um início, uma seqüência de execução e um fim. Uma *thread* é similar ao programa seqüencial, mas não é um programa em si, e roda *dentro* de um programa.

O uso de uma só thread (ou tarefa) não é útil. O benefício de usar threads está na possibilidade de executar várias threads em um único programa, rodando ao mesmo tempo. Para uma classe ser uma thread, ela deve extender a classe `Thread`:

```
1   public class ThreadSimples extends Thread {
2
3       public ThreadSimples(String str) {
4           // o construtor chama o construtor da superclasse.
5           // str será o nome da thread, que será passado como
6           // argumento, e será usado mais tarde
7           super(str);
8       }
9
10      // este método é o principal de qualquer thread
11      public void run() {
12          /* em cada iteraçao é mostrado o numero da iteração,
13             e o nome da Thread,
14             depois a thread "dorme" (sleep) por um intervalo
15             randômico entre 0 e 1000 milisegundos */
```

```
16          for (int i = 0; i < 10; i++) {
17              System.out.println(i + " " + getName());
18              try {
19                  sleep((long)(Math.random() * 1000));
20              } catch (InterruptedException e) {}
21          }
22          /* quando todas as iterações terminaram,
23             é mostrada a mensagem "Pronto!" e o nome da thread */
24          System.out.println("Pronto! " + getName());
25      }
26  }
```

A thread acima pode ser usada como um "conselheiro amoroso" na aplicação "Bem me quer, mal me quer".

```
1   public class BemMeQuer {
2       public static void main (String[] args) {
3           // duas ThreadSimples são criadas e iniciadas
4           new ThreadSimples("Bem me quer").start();
5           new ThreadSimples("Mal me quer").start();
6       }
7   }
```

A saída deve ser algo assim:

```
0 Bem me quer
0 Mal me quer
1 Bem me quer
1 Mal me quer
2 Bem me quer
2 Mal me quer
3 Bem me quer
3 Mal me quer
4 Bem me quer
4 Mal me quer
5 Mal me quer
5 Bem me quer
6 Mal me quer
7 Mal me quer
6 Bem me quer
8 Mal me quer
7 Bem me quer
9 Mal me quer
Pronto! Mal me quer
8 Bem me quer
9 Bem me quer
Pronto! Bem me quer
```

Bem me quer! Estou com sorte...

Outra técnica para a criação de threads é a implementação da interface `Runnable`. Para mais informações, consulte o Tutorial Java [2].

Apêndice A

Um Primeiro Programa em Java

A.1 Instalação e Configuração do JDK

A última versão Java Standard Development Kit (J2SDK) pode ser obtida em `http://java.sun.com`. Neste sítio também podem ser obtidos a documentação e o Tutorial Java. Após descarregar e instalar o kit, deve-se adicionar o diretório que contém os arquivos executáveis ao Path (caminho) do sistema. No Windows basta adicionar a seguinte linha ao arquivo `autoexec.bat`:

```
PATH = %PATH%;C:\JDK1.x.x\bin
```

onde `C:\JDK1.x.x` é o local da instalação do JDK. Para testar se o sistema está configurado, digite "`javac`" em uma janela de console (Prompt de Comando). Se tudo estiver OK, o compilador será executado e imediatamente finalizado.

A.2 Criação do Código Fonte

Digite o código a seguir em um editor de texto qualquer (Notepad, vi, pico, etc.). Salve o arquivo com o nome de `OlaJava.java`. Durante a digitação, tome cuidado com letras maiúsculas e minúsculas, pois Java faz distinção entre as duas. Cuide também dos ponto-e-vírgulas e do fechamento de chaves.

```
1  /**
2   * A classe OlaJava implementa uma aplicação que
3   * simplesmente mostra "Ola Java !" na tela de console.
4   */
5
6  public class OlaJava
7  {
8      public static void main(String[ ] args)
9      {
10         System.out.println("Ola Java !");
11     }
12 }
```

A.3 Compilação

Para compilar o código, abra uma janela de console e vá ao diretório onde o arquivo `OlaJava.java` se encontra. Digite o seguinte comando:

```
javac OlaJava.java
```

Se não houver erros, o código será compilado e será gerado um arquivo chamado `OlaJava.class`. Um possível erro é o sistema não reconhecer o comando `javac`. Se for este o caso, consulte a seção de instalação e configuração. Podem também ocorrer erros de compilação. Neste caso, reedite o arquivo, procurando por erros.

A.4 Execução

Para executar a aplicação, digite:

```
java OlaJava
```

A linha

```
Ola Java !
```

deve aparecer na tela. Se ocorrer o erro

```
Exception in thread "main"
java.lang.NoClassDefFoundError: OlaJava
```

o arquivo compilado não foi gerado.

Apêndice B

Glossário de Termos

Abstract Window Toolkit (AWT) Coleção de componentes para construção de Interfaces Gráficas com o Usuário. O AWT está desatualizado, e foi substituído pelo Projeto Swing.

abstract Palavra-chave que indica que uma classe não deve ser instanciada, mas sim servir como tipo base para outras classes.

API Application Programming Interface. Especificação de como é feito o acesso ao comportamento e estado de classes e objetos.

applet Um programa escrito em Java para rodar em um browser de Internet.

argumento Um item de dado especificado em uma chamada de método.

array Uma coleção de itens de dados do mesmo tipo, no qual a posição de cada dado é indicada por um número inteiro

bit A menor unidade de informação em um computador, com valor 1 ou 0.

bloco de código Em Java, qualquer código entre chaves.

booleano Uma variável ou expressão que pode ter unicamente um valor verdadeiro ou falso.

byte Uma seqüência de oito bits

bytecode Código independente de máquina gerado pelo compilador Java e executado pelo interpretador Java.

casting Conversão de um tipo de dado para outro.

classe abstrata Uma classe que contém um ou mais métodos abstratos, portanto não pode ser instanciada.

compilador Programa que traduz o código fonte para um código que pode ser executado por um computador.

escopo Indica o alcance de um membro de uma classe.

GUI Graphical User Interface. Interface gráfica com o usuário.

máquina virtual Definição abstrata de um computador, que pode ser implementada de maneiras diferentes, via software ou hardware.

método abstrato Um método que não tem implementação.

multithreaded Uma aplicação que tem partes do código executada em paralelo.

null Palavra-chave usada para especificar um valor indefinido a uma variável.

operador binário Um operador que tem dois argumentos

overriding (sobrescrever) Designar uma implementação de um método em uma subclasse diferente daquela da superclasse.

pacote Um grupo de classes.

precisão simples Numero em ponto flutuante com precisão de 32 bits.

subclasse Uma classe derivada de outra classe.

super Palavra-chave para acessar membros da superclasse da classe na qual é ela é usada.

superclasse Uma classe da qual outra(s) classes são derivadas

Swing Coleção de componentes independentes de plataforma usados para construção de Interfaces Gráficas com o Usuário.

this Palavra-chave usada para designar a classe na qual ela aparece.

thread Unidade básica de execução. Um processo pode ter várias threads rodando ao mesmo tempo, cada uma desempenhando uma tarefa diferente.

Unicode Codificação de 16-bits de caracteres definida pelo ISO 10646

variável Um item de dados identificado por um identificador

void Palavra-chave usada na declaração de um método para indicar que o mesmo não retorna nenhum valor.

wrapper Um objeto que encapsula e delega a outro objeto para alterar sua interface e comportamento.

Referências Bibliográficas

[1] Bruce Eckel. *Thinking In Java, 2nd Edition*. Prentice-Hall, Segunda Edição, 2000. Disponível para download em http://www.BruceEckel.com

[2] Lisa Friendly, Mary Campione, Kathy Walrath, Alison Huml. *The Java Tutorial*. Sun Microsystems, Segunda Edição, 2000. Disponível para download e online em http://java.sun.com/docs/books/tutorial/

[3] Sun Microsystems *Java 2 Platform, Standard Edition, v 1.3.1 API Specification*. Sun Microsystems, 2001. Disponível online e para download em http://java.sun.com/docs/

[4] Sun Microsystems *Java 2 SDK, Standard Edition Documentation*. Sun Microsystems, 2001. Disponível online e para download em http://java.sun.com/docs/

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Segunda Edição, 2000. Disponível online em http://java.sun.com/docs/

[6] Leandro J. Komoniski. *Programação em Linguagem Java*. Departamento de Informática e Estatística, UFSC, 2001.