

Henrique Duarte Moura (H3dema)

Versão 0.3

Introdução ao **Java 7 +**



Sumário

| | | |
|----------|--------------------------------------------------------|-----------|
| 1 | Sobre | 7 |
| 1.1 | Para quem é este livro? | 7 |
| 1.2 | Licença | 7 |
| 2 | Introdução | 9 |
| 2.1 | O que é Java? | 9 |
| 2.2 | Por que usar Java? | 9 |
| 2.3 | Conceitos | 10 |
| 2.4 | Bytecode | 11 |
| 2.5 | Java e Orientação a Objetos | 11 |
| 2.6 | Fazendo o Download do Java | 13 |
| 2.6.1 | Instalação | 13 |
| 2.7 | Convenções básicas para Java | 14 |
| 2.8 | Onde obter mais informação sobre o Java | 16 |
| 3 | Tipos de Dados em Java | 17 |
| 3.1 | Tipos primitivos | 17 |
| 3.1.1 | Inteiros e reais | 17 |
| 3.1.2 | Booleanos | 18 |
| 3.1.3 | Caracteres | 18 |
| 3.1.4 | Erros de arredondamento | 18 |
| 3.2 | Strings | 19 |
| 3.2.1 | Métodos úteis da classe java.lang.String | 20 |
| 4 | Variáveis, operadores e estruturas de controle | 23 |
| 4.1 | Declarando variáveis | 23 |
| 4.2 | Precedência de operações em Java | 26 |
| 4.3 | Escopo de uma variável | 28 |
| 4.4 | Variáveis de instância e variáveis de classe | 29 |
| 4.5 | Restrição de acesso | 29 |
| 4.6 | Constantes em Java | 31 |
| 4.7 | Estruturas de controle | 31 |
| 4.7.1 | Condicional if..then..else | 31 |
| 4.7.2 | Switch case | 32 |
| 4.7.3 | Operador ternário | 32 |
| 4.8 | Estruturas de repetição | 32 |
| 4.8.1 | Loop for | 32 |
| 4.8.2 | While e Do ... While | 33 |
| 4.9 | break e continue | 33 |
| 4.10 | Criando um fatorial diferente | 33 |

| | | |
|-----------|----------------------------------------------------|-----------|
| 5 | Arrays e Matrizes em Java | 37 |
| 5.1 | Arrays | 37 |
| 5.1.1 | Copiando elementos de um array | 38 |
| 5.2 | Vector | 38 |
| 6 | Datas em Java | 41 |
| 6.1 | Formatando uma data com SimpleDateFormat | 41 |
| 6.2 | Operações com data usando DateUtils | 44 |
| 7 | Usando Arquivos | 47 |
| 7.0.1 | StringTokenizer | 50 |
| 7.1 | Comparar datas de arquivos | 51 |
| 7.2 | Renomear arquivos | 52 |
| 7.3 | Apagando arquivos com Java | 57 |
| 7.4 | Criando diretórios | 57 |
| 8 | Tratamento de Exceção | 59 |
| 8.1 | Classes de Exceções | 59 |
| 8.2 | Lançando exceções | 61 |
| 8.3 | Tratando exceções | 62 |
| 9 | Estruturas de dados | 67 |
| 9.1 | Listas encadeadas | 67 |
| 9.2 | Pilhas e Filas | 68 |
| 9.3 | Sets | 68 |
| 9.4 | Maps | 68 |
| 9.5 | Hash Tables | 68 |
| 9.6 | Binary Search Trees | 68 |
| 9.7 | Binary Tree Traversal | 68 |
| 9.8 | Priority Queues | 68 |
| 9.9 | Heaps | 68 |
| 9.10 | O algoritmo Heapsort | 68 |
| 10 | Algoritmos de ordenacao | 69 |
| 10.1 | Bubble Sort | 69 |
| 10.2 | Selection Sort | 69 |
| 10.3 | Merge Sort | 69 |
| 10.4 | Quick Sort | 69 |
| 10.5 | Pesquisa | 69 |
| 10.6 | Busca Binária | 69 |
| 11 | Interface gráfica | 71 |
| 11.1 | Criando janelas | 71 |
| 11.1.1 | Janelas | 71 |
| 11.1.2 | Paineis | 71 |
| 11.1.3 | Mensagens | 71 |
| 11.2 | Interagindo com o usuários | 71 |
| 11.2.1 | Botões | 71 |
| 11.2.2 | Entradas de seleção | 71 |
| 11.2.3 | Áreas de texto | 71 |
| 11.2.4 | Menus | 71 |

| | |
|----------------------------------------------------------------------------|-----------|
| 12 Threads | 73 |
| 12.1 Threads | 73 |
| 12.2 Criando <i>threads</i> | 75 |
| 12.2.1 <i>Thread.sleep()</i> | 76 |
| 12.2.2 Esperando uma <i>thread</i> terminar para poder continuar | 77 |
| 12.2.3 Priorizando threads | 78 |
| 12.3 Bloqueios intrínsecos | 79 |
| 12.4 Semáforos em Java | 82 |
| 12.5 Tipos atômicos em Java | 83 |
| 12.6 Monitores em Java ou o problema do Jantar dos Filósofos | 84 |
| 12.6.1 Implementando Dining Philosophers | 86 |
| 12.7 Grupos de <i>threads</i> | 88 |

Capítulo 1

Sobre

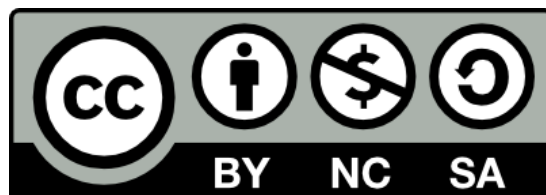
Bem vindos ao meu livro de introdução à programação usando Java. A algum tempo tenho programado em Java e tenho mantido um blog sobre Java em <http://h3dema.blogspot.com.br/>. O pessoal que me manda mensagens pelo blog às vezes tem dificuldades com o formato em vídeo das dicas. Desta forma decidi colocar neste livro uma coleção de dicas dedicadas a este tema. Este livro contém algumas das informações, das dicas e dos programas que estão no meu blog.

Estou seguindo aqui a mesma ideia básica do blog, com a diferença que neste livro os assuntos foram agrupados em tópicos mais ou menos relacionados. Assim eu estou dedicando um capítulo para cada grande tópico. Nestes capítulos apresento a teoria e as informações sobre o tema. Coloco ainda alguns exemplos para cada seção.

1.1 Para quem é este livro?

1.2 Licença

Este trabalho está licenciado sob uma Licença Creative Commons Atribuição-NãoComercial-Compartilha Igual 4.0 Internacional. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/4.0/>.



Se você gostou deste livro e quer me recompensar monetariamente por ele, você fazer uma doação via paypal. O link para a doação é este que está no pdf.



Capítulo 2

Introdução

Este livro trata da estrutura, sintaxe e paradigma de programação da linguagem e plataforma Java. O conteúdo foi preparado para que você aprenda a sintaxe do Java e os recursos básicos de programação. Este conhecimento o habilitará como programador capaz de usar a linguagem Java para desenvolver aplicativos robustos e de fácil manutenção. Para ler este livro você precisará conhecer os fundamentos básicos da programação orientada a objetos, pois a plataforma Java utiliza estes conceitos. Ao longo do livro iremos criar diversas classes em Java, incluir nelas comportamento e relacionamentos com outras classes para compor um programa.

Você pode fazer o download da versão mais atualizada deste livro em <https://goo.gl/1uZoPv>. Neste endereço ficarão também as classes de exemplo utilizadas no livro.

Vamos começar pelo básico:

- O que é Java?
- Por que usar?
- Onde consigo este software?

2.1 O que é Java?

Java é simultaneamente uma linguagem de programação e uma plataforma de computação. Foi lançada pela Sun Microsystems em 1995 e hoje é desenvolvida pela Oracle que adquiriu a Sun em 2010. A tecnologia Java capacita muitos programas da mais alta qualidade, como utilitários, jogos e aplicativos corporativos, entre muitos outros, sendo executado em computadores pessoais bem como diversos outros dispositivos, inclusive telefones celulares e dispositivos de televisão. Diferentemente das linguagens convencionais, que são compiladas para código nativo (como por exemplo os arquivos .exe no Microsoft Windows), a linguagem Java é compilada para um *bytecode* que é executado por uma máquina virtual. Este *bytecode* é único para qualquer plataforma, tornando a linguagem Java portátil.

Um pouco da história da linguagem pode ser encontrada na Wikipédia em https://pt.wikibooks.org/wiki/Java/A_hist%C3%B3ria_de_Java.

2.2 Por que usar Java?

A linguagem Java é atualmente uma das mais utilizadas em todo o mundo. Java não é apenas uma linguagem, mas sim uma plataforma de desenvolvimento. Existem diversos motivos para usar Java:

1. Java é gratuita. A máquina virtual, o compilador, o interpretador e até IDEs como Eclipse e Netbeans são gratuitas. Há vários *frameworks* e ferramentas para a implementação de sistemas em Java que são gratuitos. A documentação é gratuita. Servidores e drivers de banco de dados são gratuitos.
2. Java é uma linguagem padronizada.
3. Um código Java é interoperável. Java funciona em várias arquiteturas distintas. Java pode ser usado em sistemas operacionais diferentes, como linux, android, windows etc. Trabalha com paradigmas de programação distintos como desktop e web.
4. Java possui uma boa API. Java vem com API pronta para diversas coisas como socket, criptografia, listas, filas, acesso a arquivos, uso de compactadores, Imagem 2D, Imagem 3D, música (MP3, Wav e Midi). Sendo uma API padronizada, basta ler a documentação e utilizar as classes que a linguagem já traz prontas. A API do Java é facilmente extensível por vários outros componentes e *frameworks* que a complementam.
5. Java permite a implementação de sistemas distribuídos. Java permite a utilização de programação em redes com implementações em sua API nativa para sockets, RMI, CORBA e Webservices. Existem ainda diversos *frameworks* que exploram e ampliam a capacidade da linguagem.
6. Java possui suporte a diversos Gerenciadores de Bancos de dados.
7. Java é multi-thread. Java baseia-se na comunicação entre objetos mediante eventos. O controle de *threads* em Java é simples de ser implementado e bastante eficiente. Toda classe Java possui métodos para implementação de semáforos. Existem especificações na API Java para implementar métodos que não permitam acesso concorrente.

2.3 Conceitos

Existem alguns termos que são encontrados na literatura Java que devemos conhecer para compreender melhor esta linguagem.

- **Cache** do Java Runtime Environment (JRE) é uma área de armazenamento no console Java.
- **Java Virtual Machine (JVM)** é um conjunto de programas de software que permite a execução de instruções escritas em bytecode Java. Existem diversas JVMs disponíveis muitas plataformas de software e hardware mais comuns.
- **Applet** é um componente de software que executa uma função limitada em outro ambiente de programa, como um navegador da Web. Os applets Java fornecem recursos interativos em um navegador da Web por meio do Java Virtual Machine (JVM).
- **Java SE** é uma plataforma Java denominada Standard Edition, composto por um kit de desenvolvimento de software usado para criar mini-aplicativos e aplicativos que utilizam a linguagem de programação Java.
- **Java Development Kit (JDK)** é um pacote de software que você pode usar para desenvolver aplicativos baseados em Java. Inclui o JRE, conjunto de classes de API, compilador Java, Webstart e arquivos adicionais necessários para criar applets e aplicativos Java.
- **Java EE** é um ambiente independente da plataforma Java que cria e implementa aplicativos corporativos baseados na Web on-line. Inclui muitos componentes do Java SE e consiste em um conjunto de servidores APIs e protocolos que fornecem a funcionalidade para desenvolver aplicativos multicamadas com base na Web.

- **Java DT** é uma ferramenta usada por applets e aplicativos Java no gerenciamento da versão correta do programa Java no sistema do usuário. O Java DT proporciona para o programador uma interface JavaScript. A interface gera automaticamente o código HTML necessário para implantar RIAs (Rich Internet Applications).

2.4 Bytecode

O código *bytecode* (arquivo com extensão *.class*) é o resultado da compilação do código fonte Java (arquivo texto com extensão *.java*). O compilador que vem no pacote do SDK é o programa *javac*. O arquivo *bytecode* pode ser carregado e interpretado pelo *java* (este é inclusive o nome do programa) que é responsável pela JVM. A JVM é um programa que deve estar instalado no ambiente que você pretende rodar o programa *java*. Resumidamente o processo pode ser representado pelo diagrama abaixo:

Assim podemos identificar as principais características da linguagem Java como sendo:

- **Portabilidade:** O *bytecode* pode ser executado em diversas plataformas de hardware e software desde que o computador de destino tenha instalado a JVM.
- **Orientação a Objetos:** todo código *java* é orientado a objetos. O arquivo *bytecode* é uma classe.
- **Segurança:** a linguagem Java já provê mecanismos de execução local ou em rede com restrições de execução e inclusive protege o sistema cliente contra ataques.
- **Facilidade:** a linguagem Java retira do programador a responsabilidade de gerenciar a memória e ponteiros (estes últimos não existem).

2.5 Java e Orientação a Objetos

Programas em Java são formados por partes denominadas classes, que são compostos por métodos e atributos. Java é uma linguagem completamente orientada a objetos, isto é não é possível desenvolver nenhum programa sem seguir tal paradigma. Um sistema orientado a objetos é composto por um conjunto de classes e objetos bem definidos que interagem entre si, de modo a gerar o resultado esperado.

Dê uma olhada no tema Orientação a objetos para conhecer mais sobre o assunto. É absolutamente necessário conhecer os conceitos de orientação a objeto para poder programar corretamente em Java. No site da Oracle existem algumas lições sobre o assunto. Na página da Wikilivros existem informações que merecem ser lidas. Os conceitos abaixo são importantes e você deve entendê-los:

- **Pacote** é formado por um conjunto de classes e outros arquivos (imagens, xml etc) que funcionam em conjunto ou atuam com dependências entre si. Fisicamente são pastas e arquivos.
- **Instância**, objeto é uma variável do tipo de uma classe, isto é, é uma classe que tem seus atributos próprios e está em memória.
- **Métodos** são as funções que compõem classe.
- **Construtor** é um dos módulos de uma classe que é responsável por iniciar a criação e inicialização de uma instância de classe.
- **Modificador de acesso** descreve que outras classes podem ter acesso a classe que está se criando e é usado para indicar que uma classe pode ser acessada de fora de seu pacote.

- **Hierarquia de classes** é composta por um grupo de classes que estão relacionadas por herança.
- **Superclasse** é a classe que é estendida por uma determinada classe.
- **Subclasse** é a classe que estende determinada classe.
- **Classe base** é a classe de uma hierarquia que é uma superclasse de todas as outras classes.

Temos abaixo um exemplo de uma classe simples que representa um ponto no plano:

Listagem 2.1: Ponto2D.java

```
1  /**
2   * @author Henrique
3   * @since 12/12/2011
4   *
5   * Esta classe representa um ponto no plano cartesiano
6   */
7  public class Ponto2D {
8      private int x; // ordenada
9      private int y; // abscissa
10     // cria a classe passando as coordenadas como parâmetro
11     public void Ponto2D(int a, int b) {
12         x = a;
13         y = b;
14     }
15     // para obter a ordenada
16     public int getX() { return x; }
17     // para obter a abscissa
18     public int getY() { return y; }
19     // move o ponto para uma nova posição (x,y)
20     public void moveParaNovaPosicao(Ponto2D novaPosicao) {
21         x = novaPosicao.x;
22         y = novaPosicao.y;
23     }
24     /**
25     * desloca o ponto na horizontal
26     * param deslocamento pode ser negativo para deslocar para esquerda
27     */
28     public void moveHorizontal(int deslocamento) {
29         x += deslocamento;
30     }
31     /**
32     * desloca o ponto na vertical
33     * param deslocamento pode ser negativo para deslocar para baixo
34     */
35     public void moveVertical(int deslocamento) {
36         y += deslocamento;
37     }
38 }
```

A orientação a objetos permite que o programador tire proveito de coleções de classes existentes em bibliotecas de classes Java. Estas bibliotecas de classes são também conhecidas como Java APIs (*Applications Programming Interfaces* - interfaces de programação de aplicações).

Para aprender Java devemos aprender a sintaxe e semântica da linguagem Java de modo que possamos programar nossas classes, mas também é necessário conhecer as classes das bibliotecas de classes Java. As bibliotecas de classes são fornecidas pelos fornecedores de compiladores e por fornecedores independentes de software (ISV - independent software vendor). É possível achar na Internet diversas classes disponíveis *freeware* ou *shareware*.

2.6 Fazendo o Download do Java

O download do Java pode ser feito diretamente do site www.java.com. Para rodar o Java em seu computador é necessário que esteja instalado o **Java Runtime Environment (JRE)**. O *JRE* consiste no Java Virtual Machine (*JVM*), nas classes centrais e bibliotecas de suporte da plataforma Java.

O JRE permite que os aplicativos escritos em Java sejam executados em navegadores, bem como programas completos seja rodados diretamente no sistema operacional. O software Java Plug-in não é um programa independente e não pode ser instalado separadamente.

2.6.1 Instalação

Vamos fazer a instalação do ambiente de desenvolvimento Java no Windows. No nosso exemplo irei utilizar a versão mais atual do java 6 que é hoje Java SE 6 Update 29. Ver como baixar e instalar o programa JDK para instalação. É uma típica instalação NEXT NEXT FINISH.

Normalmente o único problema que enfrentamos é a necessidade de cadastrar o caminho dos arquivos binários do Java. Veja o nosso exemplo no vídeo abaixo. Queremos compilar a classe Ponto2D.java (em Listings 2.1) com o javac, mas ele não é achado no caminho. É necessário cadastrá-lo na variável *PATH* e reiniciar o Prompt de comando do Windows para poder compilar corretamente.

Instalando no linux Ubuntu

O Ubuntu não vem com a distribuição da Oracle que é a distribuição oficial. Você pode instalá-lo utilizando apt-get. Para instalar a versão 8, são necessários alguns comandos para adequar o apt-get. Primeiro execute os seguintes comandos:

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

Em cada caso siga as instruções na tela. O Java 8 é instalado por padrão no diretório `/usr/lib/jvm/java-8-oracle/`. Para instalar o Java 6, basta trocar o número da última linha da listagem acima para 6. E para instalar o Java 7, basta trocar o número para 7.

Você deve em seguida escolher qual das instalações existentes em seu computador você irá utilizar como padrão. Para isto execute o comando

```
sudo update-alternatives --config java
```

Você deve digitar o número correspondente à versão que será utilizada como padrão. Isto também deve ser feito para o compilador Java (javac).

```
sudo update-alternatives --config javac
```

Confira se o Java 8 foi instalado no diretório padrão. Por fim você precisa definir duas variáveis de ambiente do Java denominadas *JAVA_HOME* e *CLASSPATH*. A primeira indica onde está instalado o Java. A segunda indica onde as aplicações em Java onde elas irão procurar as classes dos usuários. O valor de *CLASSPATH* pode também ser definido usando o parâmetro *-cp* das ferramentas java e javac. Para alterar (ou criar) estas variáveis de ambiente precisamos, portanto, editar o arquivo `/etc/environment`. Neste arquivo acrescentamos as linhas:

```
JAVA_HOME="/usr/lib/jvm/java-8-oracle/"
CLASSPATH=".:$JAVA_HOME/jre/lib:$JAVA_HOME/db/lib"
```

Para ativar imediatamente a configuração entre com a linha

```
source /etc/environment
```

Para saber se ela está configurada, digite:

```
echo $JAVA_HOME  
echo $CLASSPATH
```

Se você quiser saber mais informações sobre o *CLASSPATH*, veja um link <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>. Informações sobre a configuração de *JAVA_HOME* podem ser obtidas em http://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/index.html.

2.7 Convenções básicas para Java

A linguagem Java é sensível ao caso ("*Case Sensitive*"), quer dizer, o interpretador diferencia palavras escritas com letras maiúsculas de letras minúsculas. Por exemplo são diferentes:

```
OiMundo.java  
OIMUNDO.java  
OIMundo.java  
oiMundo.java
```

As convenções principais convenções utilizadas em Java são:

- Nomes de variáveis e métodos devem começar com letras minúsculas
- Nomes de classes iniciam com letras maiúsculas
- Nomes compostos (para classes, variáveis ou métodos) devemos utilizar letras maiúsculas para as iniciais das palavras
- As constantes devem ser escritas com letras maiúsculas
- Um nome de variável pode ser qualquer identificador válido composto por uma sequência de códigos Unicode de letras e dígitos começadas com uma letra ou cifrão "\$" ou um underscore "_".
- Por convenção, os nomes devem ser sempre começados por letras.
- Também por convenção não se utiliza o cifrão apesar da compilador permitir seu uso.
- Os nomes das variáveis não podem ser palavras reservadas do Java.
- Existem três formas de se inserir comentários em um código Java. São usados caracteres especiais para caracterizar cada um destes formatos:
 - Comentário em uma linha: `//`
 - Comentário em uma ou mais linhas: `/* */`
 - Comentário a ser inserido em documentação do código: `/** */`
- Quando é colocado imediatamente acima da declaração de um método, variável ou classe, indica que o comentário será incluído automaticamente em uma página HTML gerada pelo comando `javadoc`.

Vamos ver tudo isto com um exemplo:

Listagem 2.2: OiMundo.java

```
1  /**
2   * @author Henrique
3   * @since 08/12/2011
4   * @param args é um array de strings que não é utilizado neste exemplo
5   * @ returns nada!
6   *
7   * Este é um programa completo em java que apresenta na linha de comando a
8   * mensagem
9   * Oi, Mundo!
10  * É uma variação do tradicional HelloWorld.java
11  */
12
13  public class OiMundo {
14      /*
15       * A linguagem Java tem por padrão a definição de um método
16       * main que indica onde o programa começa a ser executado,
17       * assim ao executar a linha de comando
18
19       * java OiMundo
20
21       * o interpretador Java irá procurar o método main() abaixo
22       * e por isto será apresentada a mensagem Oi, Mundo!
23
24       */
25      public static void main(String[] args) { // este programa não tem parâ
26          metros de entrada
27          String MENSAGEM = "Oi, Mundo!"; // veja como está escrito o nome da
28          constante!
29
30          System.out.println(MENSAGEM); // mais informações no site da Oracle
31          /*
32           * veja mais informações sobre System.out.println em
33           * url{http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.
34           * html}
35       */
36      }
37  }
```

Algumas das principais tags do javadoc estão relacionadas abaixo:

@ author: Atribui uma classe ou método a um autor.

@ since: deve ser utilizada como referência para a data da escrita do código ou para o número da versão

@ param: Parâmetro de um método

@ return: Retorno de um método

@ version: Versão de determinada classe ou método

@ see: string com indicação de referência adicional, tipo "Leia Também". Existe uma variação que permite a criação de um link html e referência para membros na mesma classe, para outra classe e até mesmo para outro pacote.

@ throws ou @exception: Indica as exceções lançadas pelo método que está sendo documentado

@ deprecated: Indica que a API não deve ser utilizada apesar de ainda existir na classe. Mais informações sobre o Javadoc vale a pena ir à página da Oracle.

Veja a mesma classe escrita sem os comentários. Esta é a estrutura básica de uma classe que pode ser rodada em uma JVM:

Listagem 2.3: OiMundo.java sem comentários

```
1  public class OiMundo {
2      public static void main(String[] args) {
3          String MENSAGEM = "Oi, Mundo!";
```

```
4      System.out.println(MENSAGEM);  
5  }  
6 }
```

Se quiser testá-la, entre no notepad++ (ou em outro editor de textos). Copie e cole o texto em azul acima. Grave o arquivo como OiMundo.java (note as letras maiúsculas e minúsculas - o Java é sensível ao caso). Compile com o javac e rode com o java (como na figura 2.1).

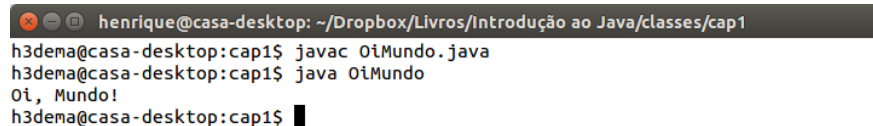
A terminal window with a dark background. The title bar shows 'henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap1'. The terminal content shows the following commands and output:
h3dema@casa-desktop:cap1\$ javac OiMundo.java
h3dema@casa-desktop:cap1\$ java OiMundo
Oi, Mundo!
h3dema@casa-desktop:cap1\$

Figura 2.1: Execução da classe OiMundo

2.8 Onde obter mais informação sobre o Java

Existem boas fontes de informação na internet.

A documentação da API oficial do Java 8 pode ser encontrada no link <http://docs.oracle.com/javase/8/docs/api/index.html?java/>. A plataforma Java 8 (Java SE) está descrita em <https://docs.oracle.com/javase/8/docs/index.html>. A Oracle mantém ainda uma seção de tutoriais sobre o Java no endereço <http://docs.oracle.com/javase/tutorial/>. Uma grande quantidade de classes reutilizáveis pode ser encontrada no site da Apache Commons. Eu utilizo diversas classes nos exemplos deste livro.

Vocês poderão acessar ainda o meu blog Pão de queijo com Java onde estão muito mais dicas. Vocês poderão ainda me mandar algum questionamento, crítica ou sugestão pelo site.

Capítulo 3

Tipos de Dados em Java

O Java possui diversos tipos primitivos de dados. Estas são aqueles tipos de informação mais usuais e básicos. Estes tipos são os habituais em outras linguagens de programação. Os tipos considerados primitivos são Boolean, Char, Inteiros, Byte, Short, Int, Long, Reais, Float e Double.

Existem classes em Java que correspondem aos tipos primitivos: `java.lang.Byte`, `Java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.Boolean` e `java.lang.Character`. Resumindo, lembre-se:

- Números inteiros: `byte`, `short`, `int`, `long`
- Números reais: `float`, `double`
- Números booleanos: `boolean`
- Caracteres: `char`
- Cadeia de caracteres: `java.lang.String`

3.1 Tipos primitivos

3.1.1 Inteiros e reais

Vamos começar com os oito tipos de dados chamados primitivos na linguagem Java. Notem o formato que foram escritos os valores *default*. Os primeiros 4 formatos são utilizados para armazenar números inteiros e ficam na memória como formato com sinal em complemento de dois:

1. **byte**: este dado é composto por uma posição de memória de 8 bits, permitindo armazenar números de -128 a 127 (inclusive). valor default: 0
2. **short**: este dado é composto por uma posição de memória de 16 bits, permitindo armazenar números de -32.768 a 32.767 (inclusive). valor default: 0
3. **int**: este dado é composto por uma posição de memória de 32 bits, permitindo armazenar números de -2.147.483.648 a 2.147.483.647 (inclusive). Este é o tipo mais comum nos programas. valor default: 0
4. **long**: este dado é composto por uma posição de memória de 64 bits, permitindo armazenar números de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (inclusive). valor default: 0L

Para a utilização de tipos `byte`, `short`, `int` ou `long` deve ser pesado a necessidade economizar memória, a capacidade do tipo conter todos os valores possíveis para o que se quer armazenar (por exemplo, para os meses do ano basta `byte`, pois sabemos que no máximo serão 12).

Para armazenar números reais o Java possui dois tipos que são armazenados em memória no formato IEEE 754:

1. **float**: este dado é composto por uma posição de memória de 32 bits, para armazenar números em precisão simples. valor default: `0.0f`
2. **double**: este dado é composto por uma posição de memória de 64 bits. valor default: `0.0d`

A biblioteca Java possui a classe **`java.math.BigDecimal`** que deve ser utilizar quando desejamos maior precisão com número reais. Também possui uma classe de grande precisão para números inteiros que é a classe **`java.math.BigInteger`**.

3.1.2 Booleanos

O Java possui um tipo capaz de representar dois valores possíveis (como “sim” e “não”). Este tipo é **boolean**: Este tipo possui somente dois valores possíveis: `true` e `false`. Representa somente um bit de informação, mas o tamanho na memória depende da plataforma. valor default: `false`

3.1.3 Caracteres

char: Este tipo representa um caracter Unicode ocupando 16 bits de memória. Se você ficou curioso é possível ver uma lista destes caracteres na Wikipédia. valor default: `'\0000'` este é o caracter nulo em Unicode

Além destes 8 tipos primitivos, a linguagem Java apresenta diversas classes para armazenamento de dados. Uma classe especial que oferece suporte a cadeias de caracteres é fornecida pela classe **`java.lang.String`**. Vamos tratar deste tipo com mais detalhes em outro post. A classe **`String`** não é tecnicamente um tipo primitivo mas em função do suporte desta classe pelo Java e pelo uso que você irá fazer em seus programas, a impressão é que a `String` é um tipo primitivo.

3.1.4 Erros de arredondamento

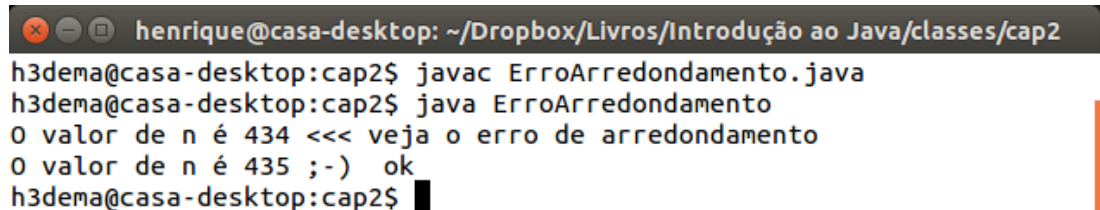
Erros de arredondamento são problemas que podem ocorrer quando trabalhamos com números em computadores, uma vez que a representação interna (no hardware) de um número não consegue cobrir todos os números possíveis. A ocorrência destes erros é esperada e é trabalho do programador controlar para que eles afetem o mínimo possível o funcionamento do programa que estão desenvolvendo. No exemplo mostrado no código 3.1 vemos um exemplo simples de um erro de arredondamento que pode ocorrer em função de um *cast*.

Listagem 3.1: ErroArredondamento.java

```
1 import java.lang.Math;
2
3 class ErroArredondamento {
4
5     public static void main(String[] args) {
6         int n;
7         double v = 4.35;
8
9         n = (int) (100 * v);
10        System.out.println("O valor de n é " + n + " <<< veja o erro de
            arredondamento");
```

```
11 |  
12 |     n = (int) Math.round(100 * v);  
13 |     System.out.println("O valor de n é "+n+" ;-) ok");  
14 | }  
15 | }
```

Veja na figura 3.1 que este erro pode ser evitado utilizando a função de arredondamento do Java.



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap2  
h3dema@casa-desktop:cap2$ javac ErroArredondamento.java  
h3dema@casa-desktop:cap2$ java ErroArredondamento  
O valor de n é 434 <<< veja o erro de arredondamento  
O valor de n é 435 ;-) ok  
h3dema@casa-desktop:cap2$
```

Figura 3.1: Rodando ErroArredondamento.java

3.2 Strings

Como já vimos em Java, as cadeias de caracteres (strings) são manipuladas por uma classe - `java.lang.String`. Um ponto importante sobre `String` em Java é que elas não podem ser alteradas, porém (felizmente) as variáveis de referências podem. Pode parecer estranha a afirmação, mas na verdade é que um objeto `String` quando criado na memória, não poderá ser alterado. Vamos ver um exemplo:

```
String s1= "Esta é a primeira string";  
String s2 = s1;  
s1 = "Altereí a referência de s1";  
s2 = "Altereí a referência de s2";
```

Os passos contendo as atribuições das imagens pode ser visto na figura 3.2. Vemos as seguintes etapas:

1. O objeto `String` "Esta é a primeira string" é criado em memória e `s1` passa a referenciá-lo.
2. Na segunda linha `s2` também referencia o mesmo objeto da memória. Não há desperdício.
3. Na terceira linha do código é criado um objeto `String` "Altereí a referência de s1" na memória que passa a ser referenciado por `s1`.
4. Na terceira linha do código é criado um objeto `String` "Altereí a referência de s2" na memória que passa a ser referenciado por `s2`.
5. O *garbage collector* do Java remove o objeto `String` "Esta é a primeira string" já que ele não é mais referenciado por nenhuma variável.

Note que não foi feita alteração no objeto `String` da memória, somente as variáveis foram alteradas. As `Strings` são criadas em um pool que armazena todas as `Strings` coincidentes em um programa Java, ou seja, se você tiver duas variáveis que referenciam uma string "abcde" saiba que só terá uma string no pool, porém duas referências e, é por esta razão que as strings são imutáveis.

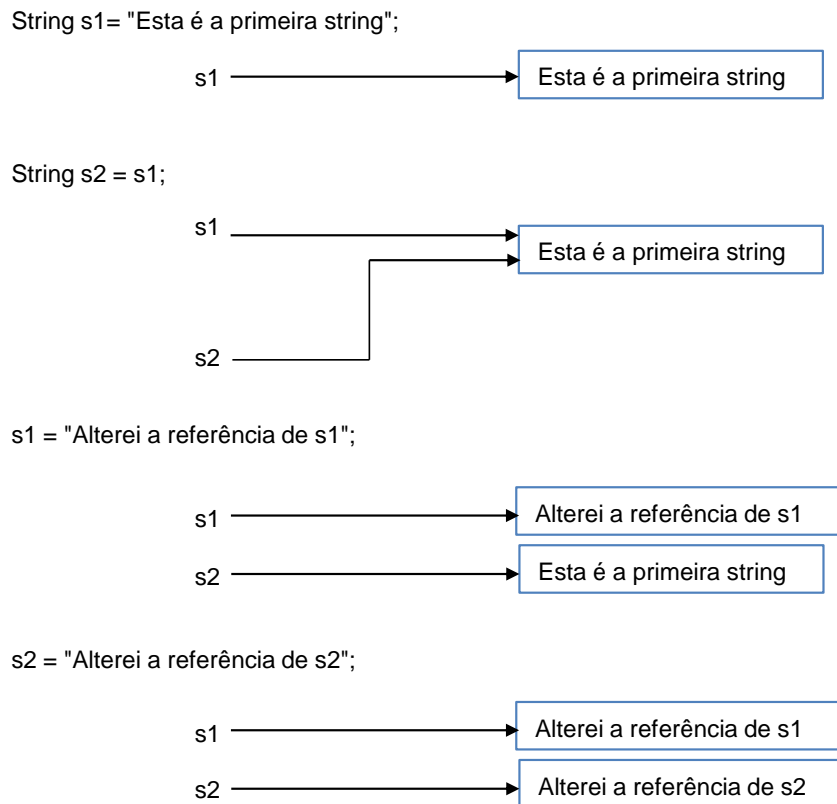


Figura 3.2: Sequencia de atribuições de strings

3.2.1 Métodos úteis da classe java.lang.String

length: Obtém o tamanho da string

substring: O nome deste método que não segue a convenção do Java. Deveria ser `subString` para ficar correto. Ele tem dois formatos:

```
substring(int inicio)
substring(int inicio , int fim)
```

onde *inicio* define a posição inicial da string (começando do 0) a ser retornada e *fim* define onde terminar a string (começando de 1).

concat: Adiciona uma string a outra que retorna como resultado do método, porém não altera a string em que o método está sendo executado.

equals - teste se uma string (passada como parâmetro) é igual a outra (do objeto).

equalsIgnoreCase - testa se uma string (do parâmetro) é igual a outra ignorando a diferença entre letras maiúsculas e minúsculas.

```
String primeiraString = "teste";
String segundaString = "TESTE";
System.out.println(segundaString.equals(primeiraString));
// resultado: false
System.out.println(segundaString.equalsIgnoreCase(primeiraString));
// resultado: true
```

replace - Substitui os caracteres de uma string (objeto), buscando o primeiro parâmetro e substituindo pelo segundo parâmetro. Note que o resultado é retornado pelo método, não alterando o objeto original. Veja o exemplo abaixo:

```
String txtASubstituir = "Est2 é um t2st2 d2 substituição";
txtASubstituir = txtASubstituir.replace('2','e');
System.out.println(txtASubstituir);
// resultado: Este é um teste de substituição
```

substring - Extrai uma seção de uma string, sendo o primeiro parâmetro a posição inicial e o segundo a quantidade de caracteres, como no exemplo:

```
String texto = "0123456789";
System.out.println(texto.substring(1,3));
// resultado: 123
// Note que a indexação da string começa em 0
```

Se for especificado um valor inválido para qualquer um dos dois argumentos, uma exceção será lançada: `java.lang.StringIndexOutOfBoundsException: String index out of range: xx`

toLowerCase - troca todas as letras para minúsculas.

toUpperCase - troca todas as letras para maiúsculas.

```
String s = "teste de maiusculas";
s = s.toUpperCase();
System.out.println(s);
// resultado: TESTE DE MAIUSCULAS
```

trim - retira os espaços no início e do final da string.

```
String s = " String sem espaço no início e final ";
s = s.trim();
System.out.println("->" + s.trim() + "<-");
// resultado: ->String sem espaço no início e final<-
```


Capítulo 4

Variáveis, operadores e estruturas de controle

Agora que vimos no capítulo 3 quais são os tipos de dados mais comuns, temos todas as peças necessárias para definir variáveis em Java. Uma variável referencia sempre a um tipo primitivo de Java, uma classe da API Java ou a qualquer outro objeto criado ou usado em nosso programa. Uma variável especifica o estado da classe ou de um objeto instância desta classe.

4.1 Declarando variáveis

A declaração de variáveis em Java é bastante simples. Se for um tipo primitivo o formato será:

[final] tipo variavel [= valor];

Se for uma classe:

[final] NomeDaClasse variavel [= New NomeDaClasse(param1, param2,)];

As declarações entre parênteses são opcionais. Vamos ver isto em uma classe real (4.1) mostrada a seguir:

Listagem 4.1: ExemploDeclaracaoVariaveis.java

```
1 public class ExemploDeclaracaoVariaveis {
2
3     public static void main(String[] args) {
4         // ***** V A R I A V E I S
5         int x, y; //declarando duas variáveis inteiras
6         x = 1; //atribuindo valores a variáveis
7         y = 2;
8
9
10        float f = 0.0f; //ponto flutuante - PI
11
12
13        double w = 1.0d; //ponto flutuante - dupla precisão
14
15        //Existem duas maneiras de fazer conversões de tipos:
16        // 1) Conversão implícita:
17        // ocorre quando o tamanho da variável destino é maior
18        // que o tamanho da variável origem
19        // ou o valor que está sendo atribuído pode ser atribuído
20        double fAsDoubleImplicito = f;
21        // 2) Conversão explícita:
22        // declaramos o tipo de destino
23        double fAsDoubleExplicito = (double) f;
```

```

24 // note a declaração do tipo entre parênteses
25
26 char ch = 'a'; // letra a
27 String s = "Esta é uma string";
28
29 // ***** C O N S T A N T E
30 final float PI = 3.141516f;
31 // esta constante já existe em java.lang.Math.PI como static double
32
33 System.out.println(f);
34 System.out.println(w);
35 System.out.println(fAsDoubleImplicito);
36 System.out.println(fAsDoubleExplicito);
37 System.out.println(ch);
38 System.out.println(s);
39 System.out.println(PI);
40
41 }
42 }

```

Compile e rode a classe para ver o resultado que mostramos na figura 4.1.

```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap3
h3dema@casa-desktop:cap3$ javac ExemploDeclaracaoVariaveis.java
h3dema@casa-desktop:cap3$ java ExemploDeclaracaoVariaveis
0.0
1.0
0.0
0.0
a
Esta é uma string
3.141516
h3dema@casa-desktop:cap3$

```

Figura 4.1: Rodando ExemploDeclaracao.java

Em Java podemos realizar diversas operações. Vamos vê-las em uma classe também. A nossa classe de exemplo é mostrada no código 4.2 a seguir.

Listagem 4.2: ExemploOperadores.java

```

1 public class ExemploOperadores {
2     public static void main(String[] args) {
3
4
5     /*
6      * Operações aritméticas unitárias (aplicadas sobre uma única variável
7
8
9      Op  Operação          Uso          Descrição
10 ++  Incremento  var++ / ++var  Retorna e adiciona / adiciona e retorna.
11 --  Decremento  var-- / --var  Retorna e subtrai / subtrai e retorna.
12 -   Negativo    -var          Inverte o sinal da variável
13 +   Positivo    +var          Não tem efeito.
14 */
15
16
17     int x = 40;
18     int y = 5;
19
20
21     x++;
22     y++;

```



```

23     System.out.println("*****_Operacoes_aritmeticas_sobre_1_variavel");
24     System.out.println("Operador_++:_x="+x+"_y="+y);
25     System.out.println("Operador_x--:_(antes)_" + x--);
26     System.out.println("Operador_x--:_(depois)_" + x);
27     y=-y;
28     System.out.println("Operador_-_y:_"+y);
29
30
31  /*
32   Operações aritméticas sobre 2 variáveis
33
34   Op   Operação      Uso      Descrição
35   +   Adição         x + y   Soma x com y.
36   -   Subtração      x - y   Subtrai y de x.
37   *   Multiplicação  x * y   Multiplica x por y.
38   /   Divisão        x / y   Divide x por y.
39   %   Resto          x % y   Resto da divisão de x por y.
40  */
41
42
43     System.out.println("*****_Operacoes_aritmeticas_sobre_2_variaveis");
44     System.out.println("x="+x+"_y="+y);
45     System.out.println("Operador_x+y:_"+ (x+y));
46     System.out.println("Operador_x-y:_"+ (x-y));
47     System.out.println("Operador_x*y:_"+ (x*y));
48     System.out.println("Operador_x/y:_"+ (x/y));
49     System.out.println("Operador_x%y:_"+ (x%y));
50
51
52  /*
53
54   Operações de lógica e relacionais
55
56   Op      Operação      Uso      Descrição
57   >        Maior que     x > y     x maior que y.
58   >=       Maior ou igual a x >= y     x maior ou igual a y.
59   <        Menor que     x < y     x menor que y
60   <=       Menor ou igual a x <= y     x menor ou igual a y.
61   ==       Igual a      x == y     x igual a y.
62   !=       Diferente de x != y     x diferente de y.
63   !        NÃO lógico (NOT) !y        contrário de y.
64   &&       E lógico (AND)  x && y     x e y.
65   ||       OU lógico (OR) x || y     x ou y.
66
67
68  */
69
70
71
72     System.out.println("*****_Operacoes_lógicas");
73     System.out.println("x="+x+"_y="+y);
74     if (x > y) {
75         System.out.println("x_é_maior_que_y");
76     }
77     else
78     {
79         if (x < y) {
80             System.out.println("x_é_menor_que_y");
81         }
82         else {
83             System.out.println("x_é_igual_a_y");
84         }
85     }
86
87
88  /*

```

```

89
90
91  Operações sobre bits
92
93
94  Op      Nome      Uso      Descrição
95  ~      Inversão    ~x      Inversão dos bits de x.
96  &      E lógico    x & y    AND bit a bit entre x e y.
97  |      OU lógico    x | y    OR bit a bit entre x e y.
98  ^      OU excl. lógico x ^ y    XOR bit a bit entre x e y.
99  <<     Desloc. a esq.  x << y    Desloc. a dir os bits de x, y vezes.
100 >>     Desloc. a dir.  x >> y    Desloca a direita os bits de x, y vezes.
101 >>>    Desloc. a dir.  x >>> y    Preenche zero a esquerda de x, y vezes.
102
103
104 */
105
106
107     int b1 = 36;
108     int b2 = 15;
109     int b3 = 2;
110
111
112     System.out.println("*****Operacoes_sobre_bits");
113     System.out.println("b1=" + Integer.toBinaryString(b1));
114     System.out.println("b2=" + Integer.toBinaryString(b2));
115     System.out.println("b3=" + Integer.toBinaryString(b3));
116     System.out.println("~b2=" + Integer.toBinaryString(~b2));
117     System.out.println("b1&b2=" + Integer.toBinaryString(b1 & b2));
118     System.out.println("b1|b2=" + Integer.toBinaryString(b1 | b2));
119     System.out.println("b2<<b3=" + Integer.toBinaryString(b2<<b3));
120     System.out.println("b1>>b3=" + Integer.toBinaryString(b1>>b3));
121
122 }
123
124 }

```

Faça variações deste programa colocando outras operações para comparar o resultado.

4.2 Precedência de operações em Java

Se temos uma operação como "1 + 1 * 2" como sabemos o resultado? O Java possui uma ordem de precedência para execução das operações. Isto é, ele realiza primeiro a multiplicação e depois a soma, assim o resultado seria 3. A ordem precedência dos operadores é (no que as operações mais em cima ocorrem antes):

```

++  --  +  -  !  (cast)
*  /  %
+  -
<<  >>  >>>
<  >  <=  >=  instanceof
==  !=
&
^
|
&&
| |
?:
=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  |=

```

É boa prática a utilização de parênteses, assim na nossa operação acima escreveríamos "1 + (1 * 2)" e não há erro de interpretação do ser humano.

| Expressão Matemática | Expressão válida em Java | Comentários |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| $\frac{x+y}{2}$ | <code>(x + y) / 2</code> | É necessário o uso de parênteses. Sem parênteses o Java computa $x + \frac{y}{2}$ |
| $\frac{xy}{2}$ | <code>x * y / 2</code> | Não é necessário parênteses, pois a ordem de precedência da multiplicação garante que a operação será realizada como previsto |
| $\sqrt{b^2 - 4ac}$ | <code>Math.sqrt(b*b-4*a*c)</code> | Podemos utilizar <code>Math.pow(b,2)</code> porém a expressão <code>b*b</code> é mais eficiente. |
| $e^{1+\frac{x}{y}}$ | <code>Math.exp(1+x/y)</code> | As expressões complexas são achatadas, pois tem que ser escritas em uma única linha |
| $\frac{x+y}{2}$ | <code>(x + y) / 2.0</code> | Se x e y são inteiros e desejamos um valor real como resposta, devemos trocar a constante para 2.0 que força a divisão ser em ponto flutuante. |
| $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ | <code>delta = Math.sqrt(b*b-4*a*c)</code> <code>x1 = (-b + delta) / (2 * a)</code> <code>x2 = (-b - delta) / (2 * a)</code> | Temos que desmembrar em diversas expressões. |

4.3 Escopo de uma variável

Podemos criar variáveis fora de qualquer método, como podemos criar variáveis definidas dentro de um determinado método. Quando uma variável é criada dentro de um método, seu escopo será limitado a este método, ou possivelmente a um sub-bloco deste método. Estas variáveis são chamadas locais. Isto é, não conseguimos acessar os seus valores fora deste método. Em contraposição às variáveis locais, as variáveis fora dos métodos são chamadas globais. Para que conhece linguagens procedurais como C ou Pascal, o conceito de global é um pouco diferente. No Pascal, por exemplo, uma variável global é acessível em todo o programa. Já no Java, uma variável global está embutida em uma classe. Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado.

```
class Classe1 {
    ...
    TipoDaVariavel1 var1;
    TipoDaVariavel2 var2;
    ...
    TipoDeRetorno1 metodo1() {
        TipoDaVariavel3 var3;
        ...
    }
    TipoDeRetorno2 metodo2() {
        TipoDaVariavel4 var4;
        ...
    }
    ...
}
```

```
}
```

Vamos analisar a listagem acima que representa a Classe1. Dentro de um bloco, podemos declarar variáveis e usá-las. Por exemplo, em metodo1 temos a variável var3. var3 pode ser acessada somente dentro deste método. Se tentarmos usar var3 dentro de metodo2, teremos um erro de compilação. A variável var4 também não pode ser acessada no metodo1, pois ela é local ao bloco do metodo2. Em compensação, as variáveis var1 e var2 podem ser utilizadas dentro de metodo1 e metodo2.

4.4 Variáveis de instância e variáveis de classe

Vamos ver mais um exemplo. Seja a classe Classe2 como mostrada na listagem abaixo.

```
class Classe2 {  
    ...  
    static TipoDaVariavel1 var1;  
    TipoDaVariavel2 var2;  
    ...  
}
```

Vemos nesta classe um exemplo de uma variável de instância e de uma variável de classe:

- Uma *variável de instância* é uma variável cujo valor é específico ao objeto e não à classe. Uma variável de instância em geral possui um valor diferente em cada objeto representante da classe. No nosso exemplo var2 é uma variável de instância. A maioria das variáveis que usamos (além das variáveis locais dentro dos métodos) são de instância. Uma variável é considerada como de instância por padrão.
- Uma *variável de classe* é uma variável cujo valor é comum a todos os objetos representantes da classe. Mudar o valor de uma variável de classe em um objeto automaticamente muda o valor para todos os objetos instâncias da mesma classe. Para declarar uma variável de classe, acrescenta-se a palavra-chave *static*, como mostramos em var1. As variáveis de classe são também chamadas *variáveis estáticas*.

4.5 Restrição de acesso

Algumas palavras-chaves existem na linguagem Java para ampliar ou restringir o acesso a uma variável. Estas palavras-chaves são acrescentadas à declaração da variável, da mesma forma que fizemos para *static*. Na verdade estes modificadores de acesso podem ser combinados ou não com o uso de *static*. Vamos ver na classe Classe3 abaixo como estes modificadores funcionam.

```
/* a classe definida neste arquivo pertence ao pacote chamado  
   meuPacote */  
package meuPacote;  
  
public class Classe3 {  
  
    /* a variável x é acessível por qualquer classe */  
    public int x;  
  
    /* a variável y é acessível pelos métodos da classe Classe3 e das  
       suas subclasses */  
    protected int y;
```

```
/* a variável z é acessível pelos métodos da classe Classe3 e das
   outras classes que pertencem ao pacote chamado algumPacote */
int z;

/* a variável w é acessível somente pelos métodos da classe
   Classe3 */
private int w;
...
}
```

Temos portanto as variáveis podem ser alteradas pelos seguintes modificadores:

- *public*: a variável pode ser acessada por qualquer outra classe e é chamada pública;
- *private*: a variável pode ser acessada somente por métodos da própria classe e é dita privada;
- *protected*: a variável pode ser acessada por métodos da própria classe e também pode ser acessada pelas subclasses da classe na qual ela é declarada, é chamada de variável protegida;
- sem modificador: Uma variável para a qual não foi especificada nenhuma das três palavras-chaves acima é dita amigável e pode ser acessada por todas as classes que pertencem ao mesmo pacote. Pacotes são agrupamentos de classes. Veja que no nosso exemplo Classe3 pertence ao pacote meuPacote. Assim todas as demais classes que pertençam ao pacote meuPacote poderão acessar a variável z.

As restrições de acesso desempenham um papel fundamental em uma boa programação orientada a objeto. Permitir que qualquer objeto tenha acesso a todas as variáveis de qualquer outro objeto, não é recomendável. Isto resulta em um código pouco robusto. É recomendável, desta forma, limitar o acesso a qualquer variável o quanto for possível. Esta restrição deve ser feita em função do papel desta variável em questão. Existem programadores que recomendam que não existam variáveis públicas. Como fazer então? Uma maneira mais segura de permitir acesso a variável consiste em declarar a variável como privada, mas incluir na classe métodos especiais controlando o acesso à variável. São métodos designados em inglês como *getter* e *setter*. Vemos na classe de exemplo, Classe4, que a variável *var* é declarada privada. Porém ela pode ser acessada usando *getVar* e *setVar*. Como para atribuir um valor é necessário chamar um método - *setVar* - podemos realizar qualquer operação sobre o valor passado, por exemplo verificando os limites.

```
public class Classe4 {

    private tipo var;
    ...
    public tipo getVar() { return var; }
    public void setVar( tipo valor ) {
        /* podemos colocar aqui testes para determinar se "valor" é
           aceitável */
        var = valor;
    }
}
```

4.6 Constantes em Java

Em Java constantes são na verdade variáveis cujo valor não pode mudar durante a execução do programa. As constantes são portanto criadas utilizando um modificador de variável. Este modificador é feito pela palavra-chave `final`. Por convenção, as constantes tem seus nomes escritos inteiramente em maiúsculas. A API do Java tem diversas constantes. Por exemplo na classe `java.lang.Math`, encontramos duas constantes:

- `public static final double PI` : que contém o valor de $\pi \approx 3,14159265358979323$. Veja mais sobre PI em <https://pt.wikipedia.org/wiki/Pi>.
- `public static final double E` : que contém o número de Euler ou Napier, $e \approx 2,718281828$, que é a base dos logaritmos naturais. Veja mais sobre e em https://pt.wikipedia.org/wiki/N%C3%BAmero_de_Euler.

4.7 Estruturas de controle

4.7.1 Condicional `if..then..else`

A forma mais simples de controle de fluxo é o comando `if ... else`. Esta estrutura é empregada para executar seletivamente ou condicionalmente um bloco ou outro bloco de comandos, mediante um critério de seleção. Esse critério é dado por uma expressão booleana, cujo valor resultante deve ser `true` ou `false`. Se esse valor for `true`, então o comando seguinte ao `if` é executado.

O comando pode ser escrito sem o `else`. Neste caso se a expressão booleana for falsa, a execução é passada para o primeiro comando depois do bloco `if`. A sintaxe para esse comando é

```
if (expressão booleana)
    [comando]    // Executado se a "expressão booleana" for true
```

Se houve o `else` e o valor da expressão condicional que define o critério de seleção for `false`, então o segundo bloco de comandos (aquele depois do `else`) é executado.

```
if(expressão booleana)
    [comando 1]    // Executado se a "expressão booleana" for true
else
    [comando 2]    // Executado se a "expressão booleana" for false
```

Esta estrutura pode ser encadeada para gerar uma estrutura condicional maior. Por exemplo podemos criar:

```
if (expressão booleana 1)
    comando 1
else if (expressão booleana 2)
    comando 1
else if (expressão booleana 3)
    comando 3
....
else
    comando n
```

4.7.2 Switch case

Como no caso execução seletiva de múltiplos comandos, há situações em que se sabe de antemão que as condições assumem valores pré definidos. Neste caso podemos utilizar a estrutura condicional *switch ... case* fornece uma forma de controle de fluxo bastante poderoso. Sua sintaxe é:

```
switch(expressão switch) {
    case [constante 1]:
        [comando 1]
        break;
    case [constante 2]:
        [comando 2]
        break;
    .
    .
    .
    case [constante n]:
        [de comando n]
        break;
    default:
        [comando]
}
```

A “expressão switch” pode ser qualquer expressão válida que retorna inteiro, boolean, char ou String. Esta é avaliada e o seu valor resultante é comparado com as constantes distintas [constante 1], [constante 2], ..., [constante n]. Caso esse valor seja igual a uma dessas constantes, o respectivo comando é executado (e todos os demais são saltados). Se o valor for diferente de todas essas constantes, então o comando presente sob o rótulo *default* é executado (e todos os demais são saltados), caso este esteja presente.

4.7.3 Operador ternário

O operador ternário funciona como um if-else compacto. Este operador faz uma avaliação seletiva de seus operandos, mediante o valor de uma expressão booleana semelhante à do comando if-else. Se essa expressão for *true*, então um primeiro operando é retornado. Se a expressão for *false*, então o segundo operando é retornado. A sua sintaxe é

```
[expressão booleana] ? [expressão 1] : [expressão 2]
```

Este operador é utilizado para reduzir o código escrito pelo programador. Por exemplo, consideremos o seguinte comando:

```
y = x > 2 ? 2-x : x*x;
```

Este comando é logicamente equivalente à seguinte sequência de comandos:

```
if (x > 2) y = 2-x;
else y = x*x;
```

4.8 Estruturas de repetição

4.8.1 Loop for

Um laço *for* é uma estrutura de controle de repetição que permite escrever de forma eficiente um laço que precisa executar um número específico de vezes. O *for* é útil quando você sabe quantas vezes a tarefa deve ser repetida. A sintaxe de um laço *for* é:


```
for (inicialização; expressão booleana; atualização) {  
    ...  
    comando  
    ...  
}
```

4.8.2 While e Do ... While

Temos duas variações em Java: *while ...* e *do .. while*.

Um laço *while* é uma estrutura de controle que permite que você repetir uma tarefa um determinado número de vezes. A sintaxe do *while* é:

```
while (expressão booleana) {  
    ...  
    comando  
    ...  
}
```

Ao executar, se o resultado da *expressão booleana* é verdadeira, então os comandos dentro do laço serão executados. Isto irá continuar enquanto o resultado da expressão for verdadeiro. O ponto-chave deste laço é que o laço não pode nunca ser executado. Quando a expressão é testada e o resultado é falso, o corpo do laço será ignorado e o primeiro comando após o *while* será executado.

Um laço *do ... while* é semelhante a um laço *while*, exceto que as instruções dentro do *do ... while* irão executar pelo menos uma vez. A sintaxe de um *do ... while* é:

```
do {  
    ...  
    comando  
    ...  
} while (expressão booleana);
```

Neste loop a *expressão booleana* aparece no final do ciclo. Deste modo as instruções dentro do loop serão executadas uma vez antes da *expressão booleana* ser testada. Se a *expressão booleana* é verdadeira, o fluxo de controle salta de volta para o primeiro comando (logo após o *do*) e as instruções no loop são executadas novamente. Esse processo se repete até que a expressão seja falsa.

4.9 break e continue

O comando *break* é usado para interromper a execução de um dos laços de iteração vistos acima ou de um comando *switch*. Este comando é comumente utilizado para produzir a parada de um laço mediante a ocorrência de alguma condição específica, antes da chegada do final natural do laço.

O comando *continue* tem a função de pular direto para final do laço, mas em vez de interromper o laço como no *break*, ele continua executando o próximo passo do laço.

4.10 Criando um fatorial diferente

A API java permite utilizar inteiros com precisão muito superior àquela em *int*. No Java encontramos a classe *java.math.BigInteger* que fornece operações análogas a todos os operadores inteiros que vimos no início deste capítulo. Também temos acesso a muitos métodos similares aos encontrados na biblioteca *java.lang.Math*.

Vamos ver abaixo um exemplo simples de algo que facilmente estoura a capacidade dos inteiros - o cálculo de fatorial. Se quiser saber mais sobre fatorial, dê uma olhada na Wikipédia. Basicamente queremos calcular o seguinte produtório:

$$n! = \begin{cases} 1, n = 0 \\ \prod_{k=1}^n k, n > 0 \end{cases}$$

A procedimento para executar o fatorial consiste em

1. verificar se $n == 0$, se for retornar 1
2. se $n > 0$, o resultado será $n * \text{fatorial}(n - 1)$

Com *BigInteger* não podemos utilizar os sinais de comparação e de operação aritmética. Temos que fazer as comparações e operações usando métodos da classe. Note que utilizamos *BigInteger.compareTo()* e *BigInteger.multiply()* para fazer a comparação dos números e a multiplicação, respectivamente. Usamos o procedimento recursivo pois ele é mais fácil de entender, mas poderia ser um loop *for*. O que você acha que é melhor - o procedimento recursivo ou o loop? Acertou que disse o loop! O loop usa menos espaço de memória, pois não tem que criar a pilha das chamadas recursivas. O loop também é mais rápido, pois não tem que fazer uma chamada de função.

Listagem 4.3: CalculaFatorial.java

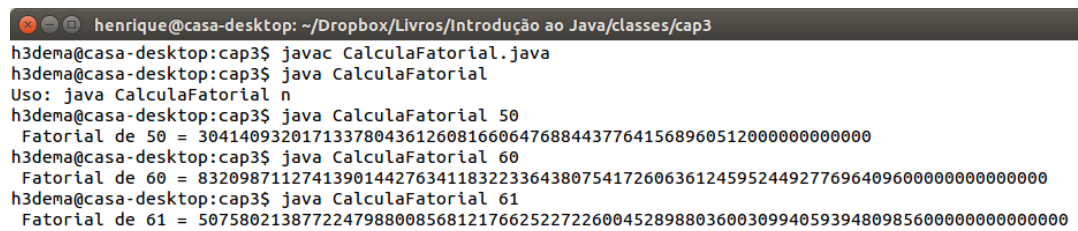
```

1 import java.math.BigInteger;
2
3 public class CalculaFatorial {
4
5     // funcao que calcula o fatorial de um numero n
6     // n tem que ser maior do que ou igual a 0
7     public static BigInteger fatorial( BigInteger n ) {
8         if ( n.compareTo( BigInteger.ONE ) <= 0 ) {
9             return BigInteger.ONE;
10        } else {
11            return n.multiply( fatorial( n.subtract(BigInteger.ONE) ) ); // n = n *
12                               f(n-1) --> recursao
13        }
14    }
15
16    // recebe como parâmetros o numero que iremos calcular o fatorial
17    public static void main( String[] args ) {
18        if ( args.length < 1 ) {
19            System.out.println("Uso: java CalculaFatorial n");
20            System.exit(0); // sair
21        } else {
22            BigInteger n = new BigInteger( args[0] );
23
24            System.out.printf("Fatorial de %d = %d\n", n, fatorial(n) );
25        }
26    }
27 }
28
29 }
```

Note que a classe *BigInteger* possui algumas constantes numéricas úteis:

- *BigInteger.ZERO* - constante *BigInteger* que representa o número 0
- *BigInteger.ONE* - constante *BigInteger* que representa o número 1
- *BigInteger.TEN* - constante *BigInteger* que representa o número 10

A figura 4.3 mostra a compilação e execução da classe *CalculaFatorial*.



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap3
h3dema@casa-desktop:cap3$ javac CalculaFatorial.java
h3dema@casa-desktop:cap3$ java CalculaFatorial
Uso: java CalculaFatorial n
h3dema@casa-desktop:cap3$ java CalculaFatorial 50
Fatorial de 50 = 30414093201713378043612608166064768844377641568960512000000000000
h3dema@casa-desktop:cap3$ java CalculaFatorial 60
Fatorial de 60 = 8320987112741390144276341183223364380754172606361245952449277696409600000000000000
h3dema@casa-desktop:cap3$ java CalculaFatorial 61
Fatorial de 61 = 50758021387722479880085681217662522722600452898803600309940593948098560000000000000
```

Figura 4.3: Rodando CalculaFatorial.java

Capítulo 5

Arrays e Matrizes em Java

5.1 Arrays

Um array em Java é uma lista de tamanho fixo que permite conter objetos similares. O array é criado com um número de elementos fixo e depois disto esta quantidade não pode ser alterada. Cada item do array (chamado elemento do array) pode ser acessado por índice numérico. A numeração deste índice sempre começa em zero. Por exemplo podemos ter as seguintes declarações:

```
int[] anArrayInteiros;  
byte[] anArrayBytes;  
short[] anArraShorts;  
long[] anArraLongs;  
float[] anArrayFloats;  
double[] anArrayDoubles;  
boolean[] anArrayBooleans;  
char[] anArrayChars;  
String[] anArrayStrings;
```

Como vimos nas declarações acima, o tamanho do array não faz parte da declaração do tipo. A quantidade de elementos é definida na chamada a new. Vamos ver um exemplo em 5.1.

Listagem 5.1: ExemploArray1.java

```
1 class ExemploArray1 {  
2  
3     public static void main(String[] args) {  
4         // cria um array para 10 inteiros  
5         int[] arrayDezInteiros = new int[10];  
6  
7         // preenche o array  
8  
9         for(int i = 0, i < 10, i++)  
10            arrayDezInteiros[i] = i;  
11  
12        // imprime  
13  
14        for(int i = 0, i < 10, i++) {  
15            System.out.println("Elemento_" + Integer.toString(i+1) + "_no_" + indice_ + "  
16                Integer.toString(i) + ":" + anArray[i]);  
17        }  
18    }  
19 }
```

Outra forma de criar esta variável é através da criação e inicialização do array na forma apresentada abaixo:

```
|| int[] arrayDezInteiros = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

O tamanho do array é dado pela quantidade de elementos indicados entre .

É possível declarar um array multidimensional utilizando dois ou mais conjuntos de parênteses, como por exemplo para criar uma matriz retangular de 10x2 utilizamos:

```
|| int[][] arrayDezPorDoisInteiros = new int[10][2];
```

Neste caso o acesso aos elementos é feito por um índice compostos como `arrayDezPorDoisInteiros[1][0]`

5.1.1 Copiando elementos de um array

Podemos copiar os elementos utilizando um método em `java.lang.System` denominado `arraycopy` que tem o seguinte formato:

```
|| arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

src - é o array de origem

srcPos - indica a posição inicial (índice) do array de origem de onde começará a cópia

dest - é o array de destino

destPos - indica a posição inicial (índice) do array de destino onde começarão a serem inseridos os dados copiados

length - número de elementos que serão copiados

Este método pode gerar 3 tipos de exceção:

- `IndexOutOfBoundsException` - qualquer tentativa fora dos limites dos arrays
- `ArrayStoreException` - se os tipos forem incompatíveis
- `NullPointerException` - se algum dos arrays for nulo

Vamos ver o exemplo mostrado no código 5.2:

Listagem 5.2: ExemploCopiaArray.java

```
1 public class ExemploCopiaArray {
2
3     public static void main(String[] args) {
4         int[] intArray = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
5
6         // declara a cópia do mesmo tamanho
7         int[] intCopia = new int[intArray.length];
8
9         // copia os dados de um para outro
10        System.arraycopy(intArray, 0, intCopia, 0, intArray.length);
11
12        for(int i = 0; i < intCopia.length; i++)
13            System.out.println(intArray[i] + " = " + intCopia[i]);
14    }
15 }
```

5.2 Vector

A classe denominada `java.util.Vector` implementa um array de objetos que pode ter a quantidade de elementos alterada - itens podem ser inseridos ou apagados depois da criação do

objeto Vector. Da mesma forma que um Array, os elementos podem ser acessados via um índice inteiro.

Esta classe permite o uso de iterators para acessar os seus componentes. Um iterator depois de criado, ao tentar ler um objeto que foi removido ou adicionado por outro processo depois do iterator ter sido criado, gera uma exceção `ConcurrentModificationException`. Este comportamento é denominado fail-fast. Uma boa programação não utiliza esta característica, já que a exceção é gerada como best effort. Esta classe retorna também Enumerations, porém este não são fail-fast.

A declaração desta classe é

```
public class Vector<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable
```

Depois de criado podemos utilizar dois métodos para inserir um objeto no Vector: `.add(objeto)` ou `.add(indice, objeto)`. Veja no exemplo estas duas situações. `size()` retorna o número de elementos do Vector.

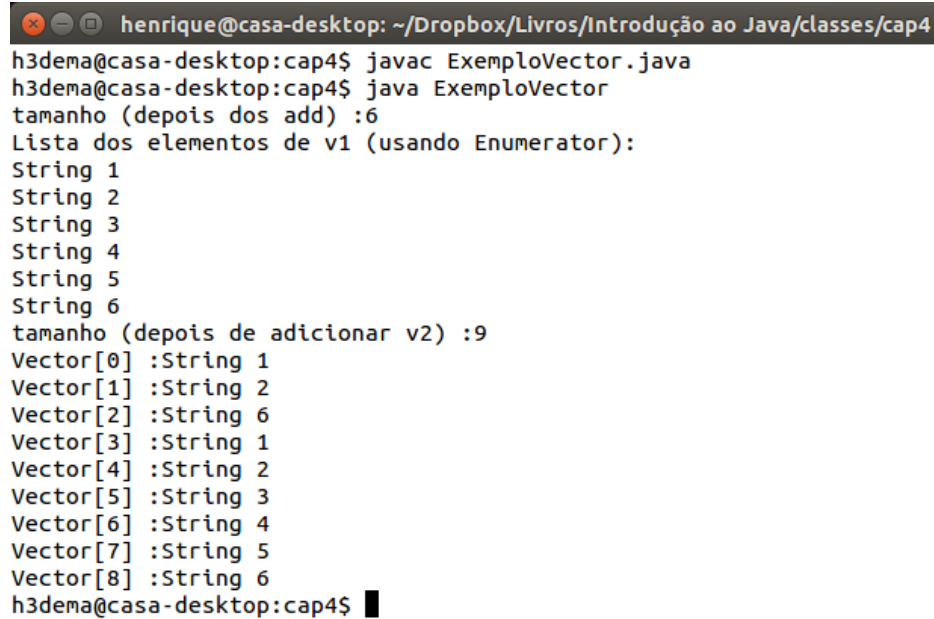
Vamos ver um exemplo de utilização do Vector criando um array de String que pode ser redimensionado. O código é mostrado em 5.3. Notem que poderia ser qualquer tipo de objeto.

Listagem 5.3: ExemploVector.java

```
1  import java.util.*;
2  import java.util.Vector;
3
4  public class ExemploVector {
5      public static void main(String[] args) {
6
7          Vector<String> v1=new Vector<String>();
8          Vector<String> v2=new Vector<String>();
9
10         // não temos que criar a quantidade de elementos
11         // eles podem ser adicionados um a um, utilizando .add
12         v1.add("String_1");
13         v1.add("String_2");
14         v1.add("String_3");
15         v1.add("String_5");
16         v1.add("String_6");
17
18         // podemos adicionar um elemento em uma posição determinada
19         v1.add(3, "String_4");
20
21         // número de elementos de v1
22         System.out.println("tamanho_(depois_dos_add)_: "+v1.size());
23
24         // uma forma de buscar os elementos de um Vector
25         Enumeration<String> vEnum = v1.elements();
26         System.out.println("Lista_dos_elementos_de_v1_(usando_Enumerator):");
27         while(vEnum.hasMoreElements())
28             System.out.println(vEnum.nextElement() + " ");
29
30         // vamos copiar os seis elementos de v1 para v2
31         for(int i=0; i<v1.size(); i++)
32             v2.add(v1.get(i));
33
34         // vamos remover os itens de índice 2 a 4
35         for(int i=4; i>1; i--)
36             v1.remove(i);
37
38         // adiciona os itens em v2 no Vector v1
39         v1.addAll(v2);
40
41         // número de elementos de v1
42         System.out.println("tamanho_(depois_de_adicionar_v2)_: "+v1.size());
```

```
43 // uma segunda forma de mostrar os elementos
44 for(int i=0; i<v1.size(); i++)
45     System.out.println("Vector["+i+"]_[]: "+v1.get(i));
46 }
47 }
48 }
```

Esta classe compilada e executada gera o seguinte resultado:



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap4
h3dema@casa-desktop:cap4$ javac ExemploVector.java
h3dema@casa-desktop:cap4$ java ExemploVector
tamanho (depois dos add) :6
Lista dos elementos de v1 (usando Enumerator):
String 1
String 2
String 3
String 4
String 5
String 6
tamanho (depois de adicionar v2) :9
Vector[0] :String 1
Vector[1] :String 2
Vector[2] :String 6
Vector[3] :String 1
Vector[4] :String 2
Vector[5] :String 3
Vector[6] :String 4
Vector[7] :String 5
Vector[8] :String 6
h3dema@casa-desktop:cap4$
```

Figura 5.1: Rodando ExemploVector.java

Capítulo 6

Datas em Java

A classe que representa datas na API Java é `java.lang.Date`. Anteriormente à especificação JDK 1.1, a classe **Date** tinha várias funções. Atualmente esta classe representa um instante no tempo contendo data e hora com precisão de até milissegundos. É necessário a utilização de duas classes adicionais **DateFormat** e **Calendar** para permitir a formatação da apresentação da data e para realizar a conversão de datas.

Os formatos dos construtores desta classe são, onde **data** é um inteiro que representa a quantidade de milissegundos contados a partir de 01/01/1970:

```
Date()  
Date(long data)
```

Os outros formatos para os construtores não devem ser utilizados (são considerados *deprecated*).

6.1 Formatando uma data com SimpleDateFormat

Vamos ver como mostrar uma data então. Para o nosso exemplo utilizarei a classe **SimpleDateFormat**. Esta classe utiliza um padrão de formatação que é montada como uma string. Existem diversas letras de formatação, que estão resumidas abaixo na tabela 6.1.

Para colocar um texto que não será interpretado, basta escrevê-lo entre aspas simples ('). Veja no exemplo abaixo (na listagem 6.1) como escrevemos as strings. A repetição das letras gera variações na interpretação do formato pelo Java. Por exemplo, se utilizamos MMM temos o mês em texto na forma de uma abreviatura, se forma MMMM temos por extenso completo, M temos o mês em número, e finalmente MM temos o mês no formato numérico com 2 dígitos.

Vamos ver em 6.1 um exemplo de uma classe simples que manipula data em Java.

Listagem 6.1: ExemploData.java

```
1 import java.util.Date;  
2 import java.text.SimpleDateFormat;  
3  
4  
5 public class ExemploData {  
6  
7     public static void main(String[] args) {  
8         Date Agora = new Date(); // obtem o datatstamp da hora da execução  
9  
10  
11         // impressão usando  
12         System.out.println("Usando toString = " + Agora.toString());  
13  
14
```

| Letra para formatação | Data ou horário | Forma de apresentação | Exemplo |
|-----------------------|--------------------------|-----------------------|---------------------------------------------|
| G | Era (AC ou DC) | Texto | AD |
| y | Year | Year | 1996; 96 |
| M | Mês in year | Mês | July; Jul; 07 |
| w | Semana no ano | Número | 27 |
| W | Semana no mês | Número | 2 |
| D | Dia no ano | Número | 189 |
| d | Dia no mês | Número | 10 |
| F | Dia da semana no ano | Número | 2 |
| E | Day in week | Text | Tuesday; Tue |
| a | Indicador Am/pm | Text | PM |
| H | Hora no dia (0-23) | Número | 0 |
| k | Hora no dia (1-24) | Número | 24 |
| K | Hora no dia (0-11) AM/PM | Número | 0 |
| h | Hora no dia (1-12) AM/PM | Número | 12 |
| m | Minuto na hora | Número | 30 |
| s | Segundo no minuto | Número | 55 |
| S | Milissegundo | Número | 978 |
| z | Time zone | General time zone | Pacific Standard Time; PST; GMT-08:00 |
| Z | Time zone | segundo RFC 822 | -0800 |

Tabela 6.1: Letras utilizadas na formatação de datas

```

15 // Use a SimpleDateFormat to print the date our way.
16 SimpleDateFormat fmt = new SimpleDateFormat ("E_dd/MM/yyyy'as' hh:mm:ss_a
   _zzz");
17 System.out.println("Usando_formatter=" + fmt.format(Agora));
18 fmt = new SimpleDateFormat ("MMM");
19 System.out.println("Mes_(abreviado)=" + fmt.format(Agora));
20 fmt = new SimpleDateFormat ("MMMM");
21 System.out.println("Mes_(completo)=" + fmt.format(Agora));
22 }
23 }

```

Se lembrarmos da definição de Data que representa o número de milissegundos que passaram deste 1º de Janeiro de 1970 00:00:00.000 GMT. Temos que as operações de adicionar ou subtrair tempos podem ser manuseadas como operações matemáticas com inteiros. Vamos ver estas operações em 6.2.

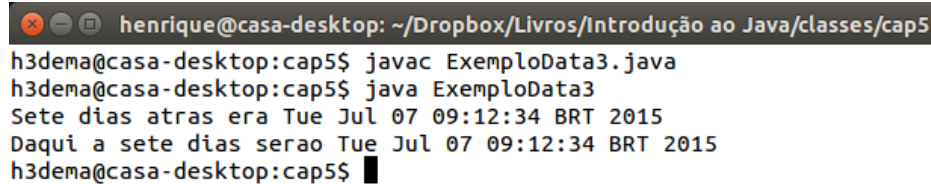
Listagem 6.2: ExemploData3.java

```

1 import java.util.Date;
2
3
4 public class ExemploData3 {
5     public static void main(String[] args) {
6         Date agora = new Date();
7         long t = agora.getTime();
8
9         // 7 dias atrás:
10        // equivale a achar t menos
11        // 7 dias x 24 horas x 60 minutos x 60 segundos x 1000 milissegundos
12        Date seteDias = new Date( t - 7 * 24 * 60 * 60 * 1000);
13        System.out.println("Sete_dias_atras_era_" + seteDias);

```

```
14
15     // daqui a sete dias
16     seteDias = new Date( t - 7 * 24 * 60 * 60 * 1000);
17     System.out.println("Daqui a sete dias serao " + seteDias);
18 }
19 }
```



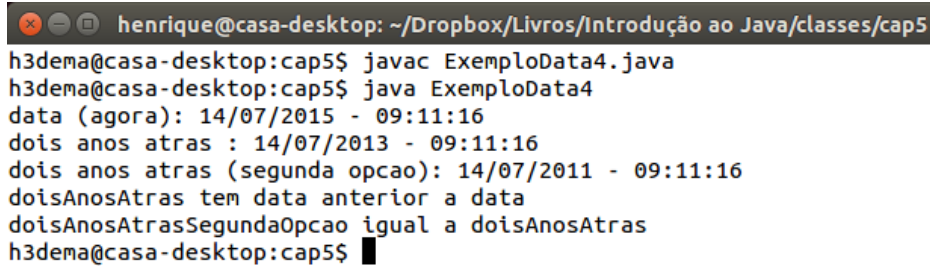
```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap5
h3dema@casa-desktop:cap5$ javac ExemploData3.java
h3dema@casa-desktop:cap5$ java ExemploData3
Sete dias atras era Tue Jul 07 09:12:34 BRT 2015
Daqui a sete dias serao Tue Jul 07 09:12:34 BRT 2015
h3dema@casa-desktop:cap5$
```

Figura 6.1: Saída para ExemploData3.java

Esta é uma classe funcional mas não é a melhor forma de realizar operações sobre datas. O Java possui uma classe denominada `java.util.Calendar` que permite trabalhar melhor com estes valores:

Listagem 6.3: ExemploData4.java

```
1 import java.util.Calendar;
2 import java.text.SimpleDateFormat;
3
4 public class ExemploData4 {
5
6     public static void main(String[] args) {
7         Calendar data = Calendar.getInstance();
8         // formatação a ser utilizada para apresentação das datas
9         SimpleDateFormat sfmt = new SimpleDateFormat("dd/MM/yyyy'-_'hh:mm:ss");
10        System.out.println("data(agora):" + sfmt.format(data.getTime()));
11
12        // obter 2 anos atras
13        Calendar doisAnosAtras = data;
14        doisAnosAtras.add(Calendar.DAY_OF_YEAR, -(365*2));
15        System.out.println("dois anos atras:" +
16            sfmt.format(doisAnosAtras.getTime()));
17
18        Calendar doisAnosAtrasSegundaOpcao = data;
19        doisAnosAtrasSegundaOpcao.add(Calendar.YEAR, -2);
20        System.out.println("dois anos atras (segunda opcao):" +
21            sfmt.format(doisAnosAtrasSegundaOpcao.getTime()));
22
23        // compara as datas
24        // existe um método Calendar.before(data) veja na documentação
25        if (doisAnosAtras.after(data)) {
26            System.out.println("doisAnosAtras tem data posterior a data");
27        } else {
28            System.out.println("doisAnosAtras tem data anterior a data");
29        }
30
31        if (doisAnosAtrasSegundaOpcao.equals(doisAnosAtras)) {
32            System.out.println("doisAnosAtrasSegundaOpcao igual a doisAnosAtras");
33        } else {
34            System.out.println("doisAnosAtrasSegundaOpcao diferente de doisAnosAtras");
35        }
36    }
37 }
```



```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap5
h3dema@casa-desktop:cap5$ javac ExemploData4.java
h3dema@casa-desktop:cap5$ java ExemploData4
data (agora): 14/07/2015 - 09:11:16
dois anos atras : 14/07/2013 - 09:11:16
dois anos atras (segunda opcao): 14/07/2011 - 09:11:16
doisAnosAtras tem data anterior a data
doisAnosAtrasSegundaOpcao igual a doisAnosAtras
h3dema@casa-desktop:cap5$

```

Figura 6.2: Saída para ExemploData4.java

6.2 Operações com data usando DateUtils

A classe *org.apache.commons.lang.time.DateUtils* contém uma grande quantidade de métodos para manipulações de datas ou calendários. *DateUtils* não faz parte da API Java padrão. Ela faz parte do pacote Apache Commons Lang. Você pode fazer o download no site da Apache Commons.

Vamos ver um método interessante para decodificar uma data que está no formato String. O método é *DateUtils.parseDate*. O analisador de *parseDate* vai tentar cada padrão de análise. A análise só é considerada bem sucedida se consegue verificar toda a cadeia de entrada. Se não existirem padrões de análise correspondentes à entrada, uma exceção *ParseException* é lançada. O formato da chamada é apresentado a seguir. Notem que podemos ter uma lista de padrões.

```
Date parseDate(String str, String... parsePatterns)
        throws ParseException
```

Outro método interessante é *isSameDay()*. Com ele conseguimos comparar dois objetos do tipo *Date*, porém a comparação considera somente a data, desconsiderando os valores de horas, minutos, etc. O formato da chamada é apresentado a seguir.

```
boolean isSameDay(Date date1, Date date2)
```

DateUtils possui ainda um conjunto de métodos que nos auxiliam a fazer operações com *Date*. Vemos nos métodos listados abaixo que podemos adicionar anos, semanas, dias, horas, minutos, segundos e milissegundos a uma data.

```
Date addYears(Date date, int amount)
Date addWeeks(Date date, int amount)
Date addMonths(Date date, int amount)
Date addDays(Date date, int amount)
Date addHours(Date date, int amount)
Date addMinutes(Date date, int amount)
Date addSeconds(Date date, int amount)
Date addMilliseconds(Date date, int amount)
```

A classe *DateUtils* da biblioteca Apache Commons possui ainda três métodos de arredondamento de datas, mostrados abaixo. O primeiro parâmetro a ser fornecido é o objeto do tipo *Date*. *ceiling* arredonda o valor de *Date* para cima, *round* arredonda o valor e *truncate* trunca o valor. Estes três métodos possuem um segundo parâmetro que informa sobre qual dos campos será feita a operação. O valor deste campo é definido em *Calendar*. Os valores são por exemplo *Calendar.HOUR*, *Calendar.MINUTE*, *Calendar.SECOND*, *Calendar.MILLISECOND*, *Calendar.DAY_OF_MONTH*, *Calendar.DATE*, *Calendar.MONTH*, *Calendar.YEAR*.

```
Date ceiling(Date date, int field)
Date round(Object date, int field)
```

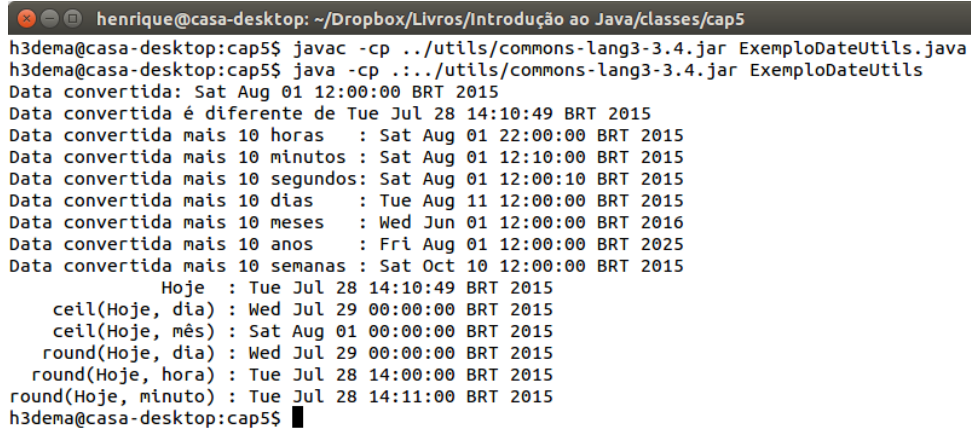
```
Date truncate(Date date, int field)
```

No código da listagem 6.4 vemos a utilização destes métodos fornecidos pela classe *DateUtils* da biblioteca Apache Commons. Note como fizemos a compilação e execução da classe na figura 6.3. É necessário utilizar a diretiva “-cp” para indicar o caminho onde está localizado o arquivo *.jar* que contém a biblioteca Apache Commons Lang.

Listagem 6.4: ExemploDateUtils4.java

```
1 import java.util.Date;
2 import java.util.Calendar;
3 import java.text.ParseException;
4 import org.apache.commons.lang3.time.DateUtils;
5 import org.apache.commons.lang3.exception.ExceptionUtils;
6
7 class ExemploDateUtils {
8
9     /**
10      * transforma uma string formatada com a data
11      * em um objeto Date
12      * @param date string no formato yyyy:MM:dd HH:mm:ss
13      * @return objeto Date ou null se não conseguiu converter
14      */
15     private static Date parseDateTimeString(String date){
16         try {
17             return DateUtils.parseDate(date,new String[]{"yyyy-MM-dd_HH:mm:ss", "dd
18                 -MM-yyyy_HH:mm:ss",});
19         }
20         catch (ParseException e) {
21             System.err.println("Erro na conversão da data."+ExceptionUtils.
22                 getMessage(e));
23             return null;
24         }
25     }
26
27     public static void main(String[] args) {
28         Date agora = new Date();
29         Date data1 = parseDateTimeString("2015-08-01_12:00:00");
30         System.out.println("Data convertida: "+data1);
31
32         if (DateUtils.isSameDay(data1, agora)) {
33             System.out.println("Datas são iguais");
34         } else {
35             System.out.println("Data convertida é diferente de "+agora);
36         }
37
38         System.out.println("Data convertida mais 10 horas: "+DateUtils.
39             addHours(data1,10));
40         System.out.println("Data convertida mais 10 minutos: "+DateUtils.
41             addMinutes(data1,10));
42         System.out.println("Data convertida mais 10 segundos: "+DateUtils.
43             addSeconds(data1,10));
44         System.out.println("Data convertida mais 10 dias: "+DateUtils.addDays
45             (data1,10));
46         System.out.println("Data convertida mais 10 meses: "+DateUtils.
47             addMonths(data1,10));
48         System.out.println("Data convertida mais 10 anos: "+DateUtils.
49             addYears(data1,10));
50         System.out.println("Data convertida mais 10 semanas: "+DateUtils.
51             addWeeks(data1,10));
52
53         System.out.println("Hoje: "+agora);
54         System.out.println("Ceil(Hoje, dia): "+DateUtils.ceiling(agora,
55             Calendar.DAY_OF_MONTH));
56         System.out.println("Ceil(Hoje, mês): "+DateUtils.ceiling(agora,
57             Calendar.MONTH));
```

```
46     System.out.println("round(Hoje, dia): "+DateUtils.round( agora,
47         Calendar.DATE));
48     System.out.println("round(Hoje, hora): "+DateUtils.round( agora,
49         Calendar.HOUR));
50     System.out.println("round(Hoje, minuto): "+DateUtils.round( agora,
51         Calendar.MINUTE));
52 }
```



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap5
h3dema@casa-desktop:cap5$ javac -cp ../utils/commons-lang3-3.4.jar ExemploDateUtils.java
h3dema@casa-desktop:cap5$ java -cp ../utils/commons-lang3-3.4.jar ExemploDateUtils
Data convertida: Sat Aug 01 12:00:00 BRT 2015
Data convertida é diferente de Tue Jul 28 14:10:49 BRT 2015
Data convertida mais 10 horas : Sat Aug 01 22:00:00 BRT 2015
Data convertida mais 10 minutos : Sat Aug 01 12:10:00 BRT 2015
Data convertida mais 10 segundos: Sat Aug 01 12:00:10 BRT 2015
Data convertida mais 10 dias : Tue Aug 11 12:00:00 BRT 2015
Data convertida mais 10 meses : Wed Jun 01 12:00:00 BRT 2016
Data convertida mais 10 anos : Fri Aug 01 12:00:00 BRT 2025
Data convertida mais 10 semanas : Sat Oct 10 12:00:00 BRT 2015
Hoje : Tue Jul 28 14:10:49 BRT 2015
ceil(Hoje, dia) : Wed Jul 29 00:00:00 BRT 2015
ceil(Hoje, mês) : Sat Aug 01 00:00:00 BRT 2015
round(Hoje, dia) : Wed Jul 29 00:00:00 BRT 2015
round(Hoje, hora) : Tue Jul 28 14:00:00 BRT 2015
round(Hoje, minuto) : Tue Jul 28 14:11:00 BRT 2015
h3dema@casa-desktop:cap5$
```

Figura 6.3: Saída para ExemploDateUtils.java

Existem ainda outros métodos úteis das classes *DateUtils* e *Date* que não falamos neste capítulo. Sugiro que deem uma olhada na documentação disponível na Internet.

Capítulo 7

Usando Arquivos

Vamos supor que desejamos fazer um programa para ler um log de registro de ponto de um coletor simples. As informações do registro de ponto que são gravadas no arquivo texto são:

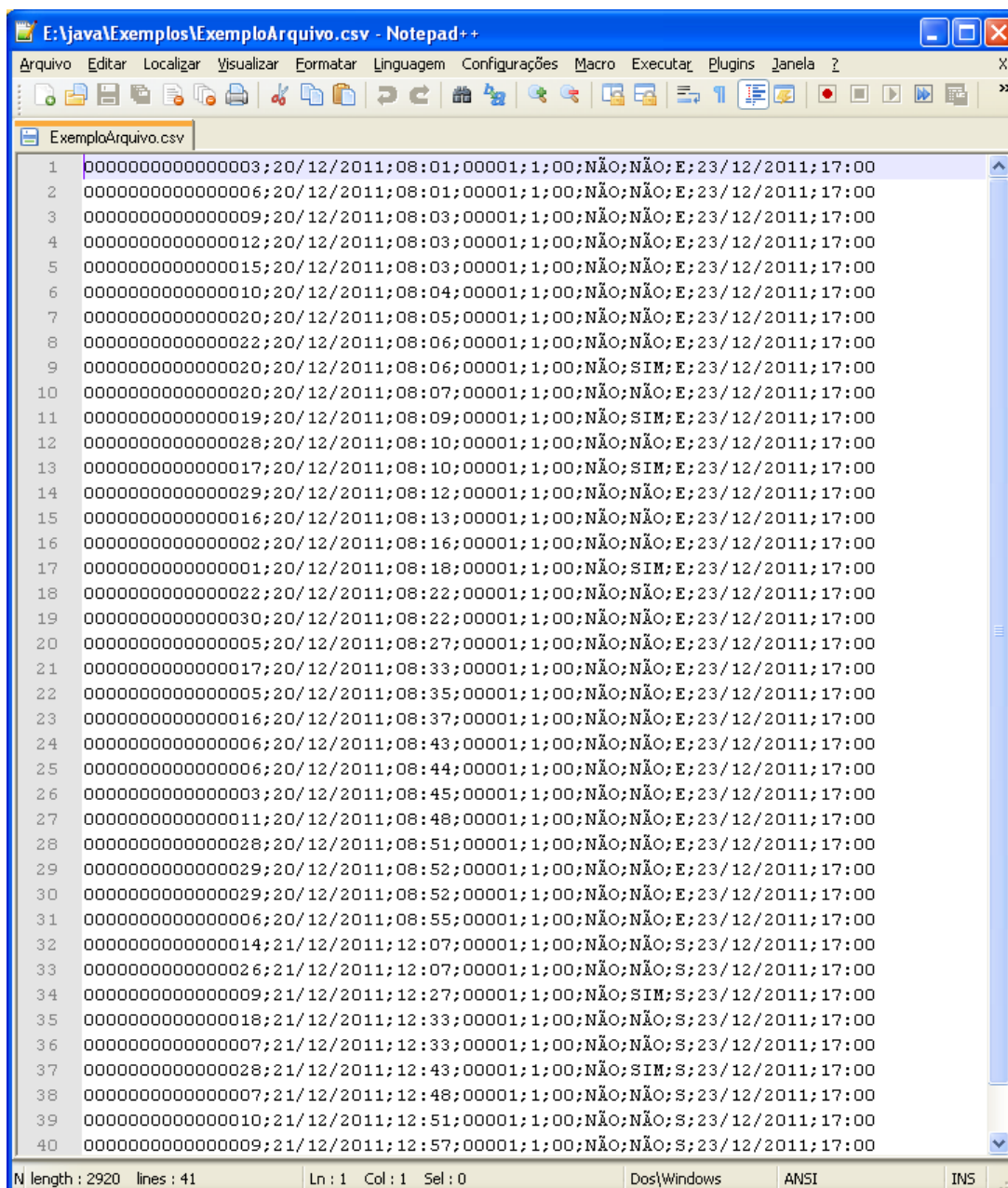
número do crachá,
data e hora do registro,
número do relógio,
indicação do método de registro (crachá, teclado, biometria,
biometria + crachá ou biometria + teclado),
número da função selecionada pelo colaborador,
indicação do registro efetuado pelo mestre,
indicação do registro efetuado pelo teclado,
sentido de passagem (entrada ou saída),
data e hora da coleta do registro.

Utilizaremos um arquivo de exemplo contendo as seguintes linhas:

A classe mostrada na figura 7.1 lê um arquivo de strings e que os registros.

Listagem 7.1: ExemploArquivo1.java

```
1 import java.io.*;
2 import java.util.*;
3 import java.util.ArrayList;
4
5
6 /*
7  campos do arquivo de batidas do relógio:
8  1) número do crachá
9  2) data
10 3) hora do registro
11 4) número do relógio
12 5) indicação do método de registro (crachá, teclado, biometria, biometria
    + crachá ou biometria + teclado)
13 6) número da função selecionada pelo colaborador
14 7) indicação do registro efetuado pelo mestre
15 8) indicação do registro efetuado pelo teclado
16 9) sentido de passagem (entrada ou saída)
17 10) data da coleta do registro
18 11) hora da coleta do registro
19 */
20 public class ExemploArquivo1 {
21     String nomeArquivo; // guarda o nome do arquivo CSV
22
23
24     public ExemploArquivo1(String nom) {
25         nomeArquivo = nom;
```



```
E:\java\Exemplos\ExemploArquivo.csv - Notepad++
Arquivo Editar Localizar Visualizar Formatar Linguagem Configurações Macro Executar Plugins Janela ?
ExemploArquivo.csv
1 0000000000000003;20/12/2011;08:01;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
2 0000000000000006;20/12/2011;08:01;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
3 0000000000000009;20/12/2011;08:03;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
4 0000000000000012;20/12/2011;08:03;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
5 0000000000000015;20/12/2011;08:03;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
6 0000000000000010;20/12/2011;08:04;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
7 0000000000000020;20/12/2011;08:05;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
8 0000000000000022;20/12/2011;08:06;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
9 0000000000000020;20/12/2011;08:06;00001;1;00;NÃO;SIM;E;23/12/2011;17:00
10 0000000000000020;20/12/2011;08:07;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
11 0000000000000019;20/12/2011;08:09;00001;1;00;NÃO;SIM;E;23/12/2011;17:00
12 0000000000000028;20/12/2011;08:10;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
13 0000000000000017;20/12/2011;08:10;00001;1;00;NÃO;SIM;E;23/12/2011;17:00
14 0000000000000029;20/12/2011;08:12;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
15 0000000000000016;20/12/2011;08:13;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
16 0000000000000002;20/12/2011;08:16;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
17 0000000000000001;20/12/2011;08:18;00001;1;00;NÃO;SIM;E;23/12/2011;17:00
18 0000000000000022;20/12/2011;08:22;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
19 0000000000000030;20/12/2011;08:22;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
20 0000000000000005;20/12/2011;08:27;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
21 0000000000000017;20/12/2011;08:33;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
22 0000000000000005;20/12/2011;08:35;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
23 0000000000000016;20/12/2011;08:37;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
24 0000000000000006;20/12/2011;08:43;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
25 0000000000000006;20/12/2011;08:44;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
26 0000000000000003;20/12/2011;08:45;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
27 0000000000000011;20/12/2011;08:48;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
28 0000000000000028;20/12/2011;08:51;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
29 0000000000000029;20/12/2011;08:52;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
30 0000000000000029;20/12/2011;08:52;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
31 0000000000000006;20/12/2011;08:55;00001;1;00;NÃO;NÃO;E;23/12/2011;17:00
32 0000000000000014;21/12/2011;12:07;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
33 0000000000000026;21/12/2011;12:07;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
34 0000000000000009;21/12/2011;12:27;00001;1;00;NÃO;SIM;S;23/12/2011;17:00
35 0000000000000018;21/12/2011;12:33;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
36 0000000000000007;21/12/2011;12:33;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
37 0000000000000028;21/12/2011;12:43;00001;1;00;NÃO;SIM;S;23/12/2011;17:00
38 0000000000000007;21/12/2011;12:48;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
39 0000000000000010;21/12/2011;12:51;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
40 0000000000000009;21/12/2011;12:57;00001;1;00;NÃO;NÃO;S;23/12/2011;17:00
N length : 2920 lines : 41 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS
```

Figura 7.1: Arquivo de dados para o exemplo de leitura da classe ExemploArquivo1.java


```
26 }
27
28
29 // carrega o arquivo para memória, armazenando na lista
30 private ArrayList<String> LeDadosCSV() {
31     String linhaLida;
32     ArrayList<String> lista = new ArrayList<String>();
33     boolean eof = false;
34     try {
35         FileReader arq = new FileReader(nomeArquivo);
36         BufferedReader buf = new BufferedReader(arq);
37         while (!eof) {
38             linhaLida = buf.readLine();
39             if (linhaLida == null) {
40                 eof = true;
41             }
42             else {
43                 lista.add(linhaLida);
44             }
45         }
46         buf.close();
47         arq.close();
48     } catch (IOException e) {
49         System.out.println("Erro de acesso a arquivo [" + nomeArquivo + "] " + e.
50             toString());
51     }
52     return lista; // retorna em lista o arquivo lido
53 }
54
55 private void ApresentaLog(String log) {
56     StringTokenizer tokens = new StringTokenizer(log, ";");
57     String[] campos = new String[11]; // são 11 campos
58     // preenche os campos
59     for(int i=0; tokens.hasMoreTokens(); i++) {
60         campos[i] = tokens.nextToken();
61     }
62     System.out.println("cracha: " + campos[0]);
63     System.out.println("registro [" + campos[8] + "]: " + campos[1] + " " + campos
64         [2]);
65 }
66
67 public static void main(String[] args) {
68     // verifica se foi passado o parâmetro com o nome do arquivo
69     if (args.length < 1) {
70         System.out.println("Uso: java ExemploArquivoLe arquivo.csv");
71         System.exit(0);
72     }
73
74     // verifica se o arquivo existe
75     if (!(new File(args[0])).exists()) {
76         System.out.println("Arquivo [" + args[0] + "] não encontrado!");
77         System.exit(0);
78     }
79
80
81     // chama a classe para processamento
82     ExemploArquivo1 batidas = new ExemploArquivo1(args[0]);
83     ArrayList<String> logs = batidas.LeDadosCSV();
84     for(String s : logs) {
85         batidas.ApresentaLog(s);
86     }
87 }
88 }
89 }
```

Para a leitura do arquivo de logs utilizamos duas classes : `FileReader` e `BufferedReader`

A classe `FileReader` está em `java.io` e sua função é permitir a leitura de arquivos de caracteres. Esta classe assume a codificação de caracteres default e utiliza um tamanho de buffer também default. Isto é suficiente para nós neste exemplo. Caso fosse necessário alterar algum destes parâmetros, deveríamos utilizar `InputStreamReader` ou `FileInputStream`.

Esta classe possui 3 construtores:

```
FileReader(File arquivo)
FileReader(String nomeDoArquivo)
FileReader(FileDescriptor arqDesc)
```

Note que utilizamos o 2º formato. Veja que utilizamos no método `main()` a construção `new File(args[0]).exists()` para testar se o arquivo existe. Poderíamos ter gerado uma variação da classe com as seguintes alterações (note que retirei as linhas da classe original que permaneceram sem alteração) como mostrado em 7.2.

Listagem 7.2: `ExemploArquivo2.java`

```

1 public class ExemploArquivo2 {
2     static File arquivo;
3
4     public ExemploArquivo2(File arq) {
5         arquivo = arq;
6     }
7
8     // carrega o arquivo para memória, armazenando na lista
9     private ArrayList<String> LeDadosCSV() {
10         ...
11         FileReader arq = new FileReader(arquivo);
12         ...
13         System.out.println("Erro de acesso a " + arquivo.getName() + " " +
14             e.toString());
15     }
16
17     public static void main(String[] args) {
18         ...
19         arquivo = new File(args[0]);
20         if (!arquivo.exists()) {
21             ...
22             ExemploArquivo2 batidas = new ExemploArquivo2(arquivo);
23         }
24     }
25 }
```

A classe `FileReader` é utilizada como `Reader`. Para ler as linhas utilizamos uma segunda classe denominada `BufferedReader`. O construtor tem o formato `BufferedReader(Reader arquivo)`. Esta classe tem um método muito interessante para nós que é `BufferedReader.readLine()`. Este método retorna:

- se existir a linha no arquivo: uma string contendo o conteúdo , sem os terminadores de linha
- se não existir, retorna `null`.

O método gera uma exceção denominada `IOException` e por isto temos que colocar o `try...catch`.

7.0.1 StringTokenizer

Vamos ver como funciona a classe **StringTokenizer** utilizada no programa 7.1. Esta classe permite que você separe palavras (*tokens*) em qualquer formato. Por exemplo, nos registros armazenados cada campo estava separado por um `”;`.

Esta classe está disponível em `java.util.StringTokenizer`. Possui três construtores:

```
StringTokenizer(String str)
StringTokenizer(String str, String delim)
StringTokenizer(String str, String delim, boolean returnDelims)
```

onde:

`str` – é a string que desejamos vasculhar
`delim` – indica o delimitador que iremos procurar, por exemplo ";"
`returnDelims` – uma flag que indice se os delimitadores devem ser retornados como tokens. Nos dois primeiros construtores, onde esta flag não é indicada, os delimitadores NÃO são tratados como tokens.

Os métodos que utilizamos foram:

- `countTokens()` - indica quantos elementos foram separados
- `hasMoreTokens()` - retorna true se ainda existem elementos na lista para serem recuperados com `nextToken`
- `nextToken()` - retorna o próximo token. Notem que foi este método que nos permitiu recuperar os campos do registro do arquivo texto.

7.1 Comparar datas de arquivos

Muitas vezes quando desejamos efetuar operações sobre arquivos, como por exemplo, copiar um arquivo de um diretório para outro, queremos ter certeza que não estamos sobrescrevendo o novo pelo antigo. Para saber isto temos que comparar as datas do arquivo de origem com o destino. Vamos ver uma classe que faz esta comparação em 7.3.

Listagem 7.3: `ComparaDatasArquivos.java`

```
1 import java.util.*;
2 import java.io.File;
3
4
5 /**
6  * compara a data de dois arquivos
7  * @param arg[0] arquivo de origem
8  * @param arg[1] arquivo de destino
9  */
10 public class ComparaDatasArquivos {
11     public static void main(String[] args) {
12         if (args.length < 2) {
13             System.out.println("Uso: java ComparaDatasArquivos arquivo1 arquivo2");
14             System.exit(0);
15         }
16
17         // pega os dois arquivos
18         File f1 = new File(args[0]);
19         File f2 = new File(args[1]);
20
21
22         if (!(f1.exists() && f2.exists())) {
23             System.out.println("Arquivo inexistente! Operação cancelada");
24             System.exit(0);
25         }
26     }
27 }
```

```

28
29 // pega as datas dos arquivos
30 long d1 = f1.lastModified();
31 long d2 = f2.lastModified();
32 String situacao;
33 if (d1 == d2)
34     situacao = "tem_mesma_data_que";
35 else if (d1 < d2)
36     situacao = "eh_mais_antigo_que";
37 else
38     situacao = "eh_mais_novo_que";
39 System.out.println(f1.getName() + " " + situacao + " " + f2.getName());
40 }
41 }

```

Na figura 7.2 vemos o resultado da execução da classe `ComparaDatasArquivos.java`. Notem a construção do `if .. else if .. else` do final da classe. Quando se tem um comando, não é necessário criar um bloco com `{ .. }`.

```

CA H3dema
E:\java\Exemplos>javac ComparaDatasArquivos.java
E:\java\Exemplos>dir ZipJ*
0 volume na unidade E é UBOX_Blog
0 número de série do volume é 0000-0811

Pasta de E:\java\Exemplos
03/01/2012 00:50 4.940 ZipJava.java
03/01/2012 00:57 2.467 ZipJava.zip
2 arquivo(s) 7.407 bytes
0 pasta(s) 557.022.306.304 bytes disponíveis

E:\java\Exemplos>java ComparaDatasArquivos ZipJava.java ZipJava.zip
ZipJava.java eh mais antigo que ZipJava.zip

E:\java\Exemplos>java ComparaDatasArquivos
Uso: java ComparaDatasArquivos arquivo1 arquivo2

E:\java\Exemplos>

```

Figura 7.2: Arquivo de dados para o exemplo de leitura da classe `ComparaDatasArquivos.java`

7.2 Renomear arquivos

Vamos dizer que você baixou um torrent. Os arquivos do torrent foram todos acrescidos de um sufixo com o nome do uploader e os espaços foram trocados por "+". Queremos retirar este sufixo e trocar os "+" por espaços. Vamos construir uma classe 7.4 que permite renomear um arquivo, trocando o nome antigo para o nome desejado.

Listagem 7.4: `RenomeiaArquivos.java`

```

1 import java.io.File;
2 import java.io.IOException;
3
4 /**
5  * renomeia um arquivo, remove um padrão e substitui por outro
6  * @param arg[0] arquivo ou diretorio
7  * @param arg[1] padrao a ser procurado
8  * @param arg[2] novo valor
9  * @param arg[3] opcional. se igual a "sub" procura nos subdiretórios
10 */

```

```
11 public class RenomeiaArquivos {
12     String padrao;
13     String novoPadrao;
14     boolean subDiretorios = false;
15
16     public RenomeiaArquivos(String seq, String novaSeq, boolean sub) {
17         padrao = seq;
18         novoPadrao = novaSeq;
19         subDiretorios = sub;
20     }
21
22     private String adequaString(String original) {
23         return original.replace(padrao, novoPadrao);
24     }
25
26     private boolean renomeiaArquivo(File f) throws IOException {
27         if (f.getName().contains(padrao)) {
28             String novoNome = adequaString(f.getAbsolutePath());
29             return f.renameTo(new File(novoNome));
30         }
31         return false;
32     }
33
34     private void renomeiaDiretorio(File f) throws IOException {
35         String[] lista = f.list();
36         File arq;
37         for(int i=0; i < lista.length; i++) {
38             arq = new File(lista[i]);
39             if (arq.isFile()) {
40                 renomeiaArquivo(arq);
41             } else if (subDiretorios) {
42                 System.out.println("Dir>"+arq.getName());
43                 renomeiaDiretorio(arq);
44             }
45         }
46     }
47
48     private void executar(String nomeArquivo) throws IOException {
49         File f = new File(nomeArquivo);
50         System.out.println("Renomeando_" + f.getName());
51         if (f.isFile()) {
52             renomeiaArquivo(f);
53         } else {
54             renomeiaDiretorio(f);
55         }
56     }
57
58     public static void main(String[] args) {
59
60         if (args.length < 3) {
61             System.out.println("Uso: _java_RenomeiaArquivos_{nomeArquivo|
62                 nomeDiretorio}_\"padrao_original\"_\"novo_padrao\"_\"[sub]");
63             System.exit(0); // sair
64         }
65
66         boolean sub = (args.length > 3) && (args[3].equalsIgnoreCase("sub"));
67         RenomeiaArquivos ren = new RenomeiaArquivos(args[1], args[2], sub);
68         try {
69             ren.executar(args[0]);
70         } catch (IOException e) {
71             System.out.println(e.toString());
72         }
73     }
74 }
```

Na figura 7.2 vemos o resultado da execução da classe `RenomeiaArquivos.java`.

```

henrique@casa-desktop: ~/Blog/java/Exemplos/20111231
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ javac RenomeiaArquivos.java
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ ls
RenomeiaArquivo.java  RenomeiaArquivos.class  RenomeiaArquivos.java
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ java RenomeiaArquivos RenomeiaArquivo.java -l
Renomeando RenomeiaArquivo.java-
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ ls
RenomeiaArquivo.java_11_NovoPadrao  RenomeiaArquivos.class  RenomeiaArquivos.java
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ java RenomeiaArquivos RenomeiaArquivo.java_11_NovoPadrao
Renomeando RenomeiaArquivo.java_11
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$ ls
RenomeiaArquivo.java_NovoPadrao  RenomeiaArquivos.class  RenomeiaArquivos.java
henrique@casa-desktop:~/Blog/java/Exemplos/20111231$

```

Figura 7.3: Arquivo de dados para o exemplo de leitura da classe `RenomeiaArquivos.java`

Nós vimos na postagem anterior que `java.io.File` permite renomear arquivos do sistema operacional. A nova versão `java.nio` também possui este recurso que vamos explorar aqui. Primeiro a nossa classe de exemplo. Lembrem que só vai compilar se seu Java SE for java7+.

Listagem 7.5: `Renomear.java`

```

1  import java.nio.file.*;
2  import java.io.IOException;
3
4
5  /**
6   * @param arg[0] padrao a ser procurado
7   * @param arg[1] novo valor
8   * @param arg[2] arquivo ou diretorio
9   * @param arg[3] arquivo ou diretorio
10  * ....
11  */
12  class Renomear {
13
14
15      String padrao;
16      String novoPadrao;
17
18
19      public Renomear(String seq, String novaSeq) {
20          padrao = seq;
21          novoPadrao = novaSeq;
22      }
23
24
25
26
27      public void executar(String arquivo) {
28          Path f = FileSystems.getDefault().getPath(arquivo);
29          // getFileName() retorna Path por isto temos que encadear
30          // com .toString() para procurar pelo padrao
31          String nomeArquivo = f.getFileName().toString();
32          if (nomeArquivo.contains(padrao)) {
33              // move() funciona para mover um arquivo ou para renomea-lo
34              try {
35                  Files.move(f, f.resolveSibling( nomeArquivo.replace(padrao,
36                      novoPadrao) ));

```

```

36         } catch (IOException e) {
37             System.out.println(e.toString());
38         }
39     }
40 }
41
42
43 public static void main(String[] args) {
44
45     if (args.length < 3) {
46         System.out.println("Uso: java Renomear \"padrao original\" \"novo\"
47             padrao \"[{nomeArquivo|nomeDiretorio}] [{nomeArquivo|nomeDiretorio}]\"
48             ");
49         System.exit(0); // sair
50     }
51
52     Renomear r = new Renomear(args[0], args[1]);
53     for(int i = 2; i < args.length; i++)
54         r.executar(args[i]); // renomeia cada arquivo da lista
55
56 }
57 }

```

Compilamos, rodamos e (acreditem) funciona! Veja o resultado em 7.4.

```

H3dema
E:\java\Exemplos\20120104>javac Renomear.java
E:\java\Exemplos\20120104>dir
0 volume na unidade E é USBXX_Blog
0 número de série do volume é 0000-0011

Pasta de E:\java\Exemplos\20120104
13/01/2012 23:35      2.028 Achador.class
13/01/2012 23:35      4.162 Achador.java
13/01/2012 23:34      4.162 Achador.java~
13/01/2012 23:35      2.025 FiltroDePesquisa.class
13/01/2012 23:35      7.368 GetOpt.class
13/01/2012 23:34     40.505 GetOpt.java
13/01/2012 23:29     40.505 GetOpt.java~
13/01/2012 23:13     40.490 Getopt.java
13/01/2012 23:17      1.685 LongOpt.class
13/01/2012 23:17      5.724 LongOpt.java
13/01/2012 23:15      6.530 LongOpt.java.original~
13/01/2012 23:16      5.745 LongOpt.java~
13/01/2012 23:12      1.338 RemoveNumeros.class
13/01/2012 23:12      962 RemoveNumeros.java
13/01/2012 23:11      982 RemoveNumeros.java~
14/01/2012 23:54      1.758 Renomear.class
14/01/2012 23:52      1.329 Renomear.java
14/01/2012 23:37      1.215 Renomear.java~
18 arquivo(s)          192.513 bytes
0 pasta(s) 541.520.265.216 bytes disponíveis

E:\java\Exemplos\20120104>java Renomear ~ 111 Achador.java~ GetOpt.java~
E:\java\Exemplos\20120104>

```

Figura 7.4: Arquivo de dados para o exemplo de leitura da classe Renomear.java

O padrão procurado foi " " que será trocado por "111". Um dos arquivos selecionado na lista está marcado com a seta. Na figura 7.5 mostra o resultado.

Vamos ver o que foi utilizado:

- `FileSystems.getDefault()` retorna o objeto `FileSystem` padrão que utilizaremos para obter

```

C:\ H3dema
14/01/2012 23:37 1.215 Renomear.java~
18 arquivo(s) 192.513 bytes
0 pasta(s) 541.520.265.216 bytes disponíveis

E:\java\Exemplos\20120104>java Renomear ~ lll Achador.java~ GetOpt.java~
E:\java\Exemplos\20120104>dir
0 volume na unidade E é UBOX_Blog
0 número de série do volume é 0000-0811

Pasta de E:\java\Exemplos\20120104

13/01/2012 23:35 2.028 Achador.class
13/01/2012 23:35 4.162 Achador.java
13/01/2012 23:34 4.162 Achador.javalll
13/01/2012 23:35 2.025 FiltroDePesquisa.class
13/01/2012 23:35 7.368 GetOpt.class
13/01/2012 23:34 48.505 GetOpt.java
13/01/2012 23:29 48.505 GetOpt.javalll
13/01/2012 23:13 48.490 GetOpt.java~
13/01/2012 23:17 1.685 LongOpt.class
13/01/2012 23:17 5.724 LongOpt.java
13/01/2012 23:15 6.530 LongOpt.javalll
13/01/2012 23:16 5.745 LongOpt.java~
13/01/2012 23:12 1.338 RemoveNumeros.class
13/01/2012 23:12 962 RemoveNumeros.java
13/01/2012 23:11 982 RemoveNumeros.javalll
14/01/2012 23:54 1.758 Renomear.class
14/01/2012 23:52 1.329 Renomear.java
14/01/2012 23:37 1.215 Renomear.java~
14/01/2012 23:55 17.375 Renomear01.png
19 arquivo(s) 209.888 bytes
0 pasta(s) 541.520.244.736 bytes disponíveis

E:\java\Exemplos\20120104>

```

Figura 7.5: Arquivo de dados para o exemplo de leitura da classe Renomear.java

o path para o arquivo. Este método retorna *FileSystem* e por isto podemos utilizar o método abaixo.

- *FileSystems.getDefault().getPath(String s)* converte uma String com um caminho e retorna um objeto do tipo Path que pode ser utilizado para localizar e/ou acessar um arquivo.
- *Path.getFileName()* retorna um Path com o nome do arquivo (ou diretório) representado pelo objeto Path.
- *Path.toString()* retorna uma String com o caminho do arquivo/diretório. Em função disto podemos usar uma chamada com *Path.getFileName().toString()* para obter o nome do arquivo como String.
- *Files.move()* tem o seguinte formato *move(Path origem, Path destino, CopyOption opcoes)*. Este terceiro parâmetro é opcional e nós não utilizamos. *move()* permite mover ou renomear o arquivo da mesma maneira que o comando *mv* do linux.
- *Path.resolveSibling(String s)* converte uma string em um caminho, utilizando o caminho original. Por exemplo:
 - se temos um Path p com `"/var/log/system"`
 - `Path p1 = p.resolveSibling("system.1");`
 - p1 vai ser `"/var/log/system.1"`

7.3 Apagando arquivos com Java

Continuando a utilizar a classe `File` temos um outro exemplo bacana e simples... para apagar arquivos.

Listagem 7.6: `ApagaArquivos.java`

```
1 import java.io.File;
2 /**
3  * remove os arquivos listados
4  * @param arg[0] arquivo ou diretorio a ser removido
5  * @param arg[1] igual ...
6  */
7 public class ApagaArquivos {
8     public static void main(String[] args) {
9         if (args.length < 1) {
10             System.out.println("Uso: java ApagaArquivos {nomeArquivo|nomeDiretorio}
11                 ...");
12             System.exit(0); // sair
13         }
14
15         for(int i=0; i<args.length; i++) {
16             File f = new File(args[i]);
17             f.delete();
18         }
19     }
20 }
```

Na figura 7.2 vemos o resultado da execução da classe `ApagaArquivos.java`.

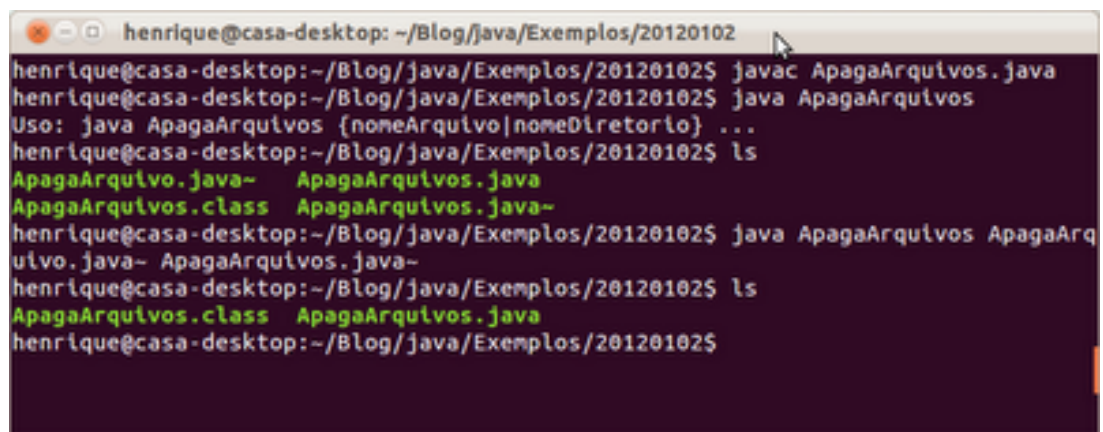


Figura 7.6: Arquivo de dados para o exemplo de leitura da classe `ApagaArquivos.java`

7.4 Criando diretórios

Na classe `java.io.File` existem dois métodos para criar um novo diretório:

`mkdir()`: cria um diretório de acordo com o pathname que foi passado
`mkdir()`: cria um diretório de acordo com o pathname que foi passado, incluindo todos os subdiretórios necessários

Ambos os métodos retornam **true** se conseguiram criar o diretório.

Vamos ver um exemplo. Utilizando `mkdir()` conseguimos criar um diretório a partir do que estamos ("."), mas não conseguimos criar um diretório e seu subdiretório em um único comando. Já com `mkdirs()` esta segunda opção também funciona.

A partir da versão 7 do Java existe um novo pacote denominado `java.nio` que implementa e melhora várias coisas que já existiam no `java.io`. Vamos ver o exemplo 7.7 para criar diretórios:

Listagem 7.7: CriaDirectories.java

```

1  import java.nio.file.*;
2  import java.io.IOException;
3
4  public class CriaDirectories {
5
6      public static void main(String[] args) {
7          // usa o args[0] para indicar o que queremos criar
8          if (args.length == 0) {
9              System.out.println("Uso: java CriaDirectories <nome do diretório>");
10             return;
11         }
12         Path dir = FileSystems.getDefault().getPath(args[0]);
13         try {
14             Path p = Files.createDirectories(dir);
15             System.out.println(p.toAbsolutePath());
16         } catch (IOException e) { // createDirectories() gera IOException
17             System.out.println("Erro na criação do diretório: " + dir.getFileName());
18             System.out.println(e.toString());
19         }
20     }
21 }

```

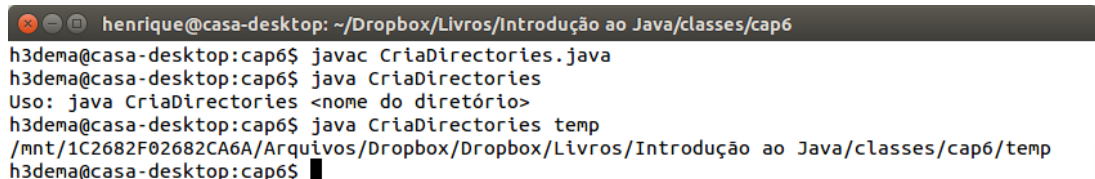
Vejam que ele ficou muito diferente do outro. Utilizamos duas novas classes `Path` e `Files`. Esta segunda é estática. Existe também o método `createDirectory()` que realiza função equivalente a `java.io.File.mkdir()`. As funções têm a forma genérica de:

```

createDirectory(Path d, FileAttributes <?> attr)
createDirectories(Path d, FileAttributes <?> attr)

```

Ambas retornam `Path` (como no exemplo da listagem 7.7) indicando o diretório. O parâmetro `d` é obviamente o diretório que queremos criar (notem que é um objeto do tipo `Path`). No exemplo 7.7 criamos o objeto `dir` (do tipo `Path`) a partir de `args[0]` para poder chamar o método `createDirectory`. `attr` é uma lista opcional composta pelo par (String nomeDoAtributo, T valorDoAtributo) que desejamos setar para o diretório.



```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/cap6
h3dema@casa-desktop:cap6$ javac CriaDirectories.java
h3dema@casa-desktop:cap6$ java CriaDirectories
Uso: java CriaDirectories <nome do diretório>
h3dema@casa-desktop:cap6$ java CriaDirectories temp
/mnt/1C2682F02682CA6A/Arquivos/Dropbox/Dropbox/Livros/Introdução ao Java/classes/cap6/temp
h3dema@casa-desktop:cap6$

```

Figura 7.7: Criando diretórios usando CriaDirectories.java

Capítulo 8

Tratamento de Exceção

As exceções são a maneira habitual para indicar em Java para um método que uma condição anormal ocorreu. Quando um método encontra uma condição anormal (também chamada condição de exceção) que o método não pode lidar, o método pode lançar uma exceção. Lançar uma exceção é indica que há um problema que não pode ser tratado no local em que ocorreu. Em algum lugar, caso tenhamos preparado nosso programa para isto, esta exceção vai ser capturada e que o problema será resolvido.

Exceções são capturados por manipuladores posicionados ao longo da pilha de invocação do método na *thread*, como podemos ver na figura 8.1. Se o método de chamada não está preparado para capturar a exceção, ele lança a exceção até o seu método de chamada, e assim por diante. Se uma das *threads* do programa lança uma exceção que não é capturada por qualquer método ao longo da pilha de invocação, esta *thread* irá expirar. Quando programamos em Java, devemos posicionar manipuladores de exceção estrategicamente, para que o programa consiga capturar pegar e tratar todas as exceções, a partir da qual o programa poderá se recuperar.

Em Java, exceção é um evento que interrompe o fluxo normal do programa. Exceção é também um objeto que é lançado em tempo de execução indicando um condição anormal.

8.1 Classes de Exceções

Em Java, as exceções são objetos. Quando lançamos uma exceção, um objeto é criado. Não podemos lançar qualquer objeto como uma exceção. Somente os objetos cujas classes descendem de da classe *Throwable* podem ser lançados como exceção. *Throwable* serve como a classe base para uma família inteira de exceções. Ela é declarada em *java.lang*. O programa pode instanciá-la e lançá-la. Uma pequena parte desta família é mostrado na figura 8.2.

Como podemos ver na figura 8.2, *Throwable* tem duas subclasses diretas, *Exception* e *Error*. Os membros da família *Exception* são lançados para sinalizar condições anormais. Estas condições podem muitas vezes ser manipuladas por algum coletor. Contudo é possível que elas não possam ser capturados e, portanto, resultem em uma *thread* morta. Já os membros da família de *Error* são normalmente lançados por problemas mais graves, como por exemplo *OutOfMemoryError*. Este problemas podem não ser tão fáceis de manusear. O código que escrevemos deve lançar, normalmente, exceções e não erros. Os erros são normalmente lançados pelos métodos da API de Java ou pela própria máquina virtual Java.

O Java considera que existem três tipos de exceções:

1. Exceção verificada: As classes que estendem classe *Throwable*, com exceção de *RuntimeException* e *Error*, são conhecidas como exceções verificadas.

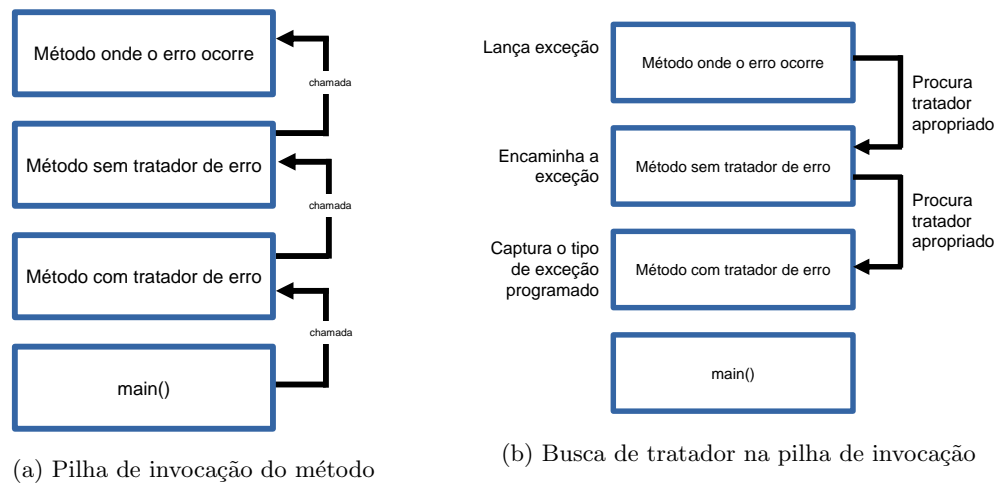


Figura 8.1: Tratando exceções na pilha de invocação

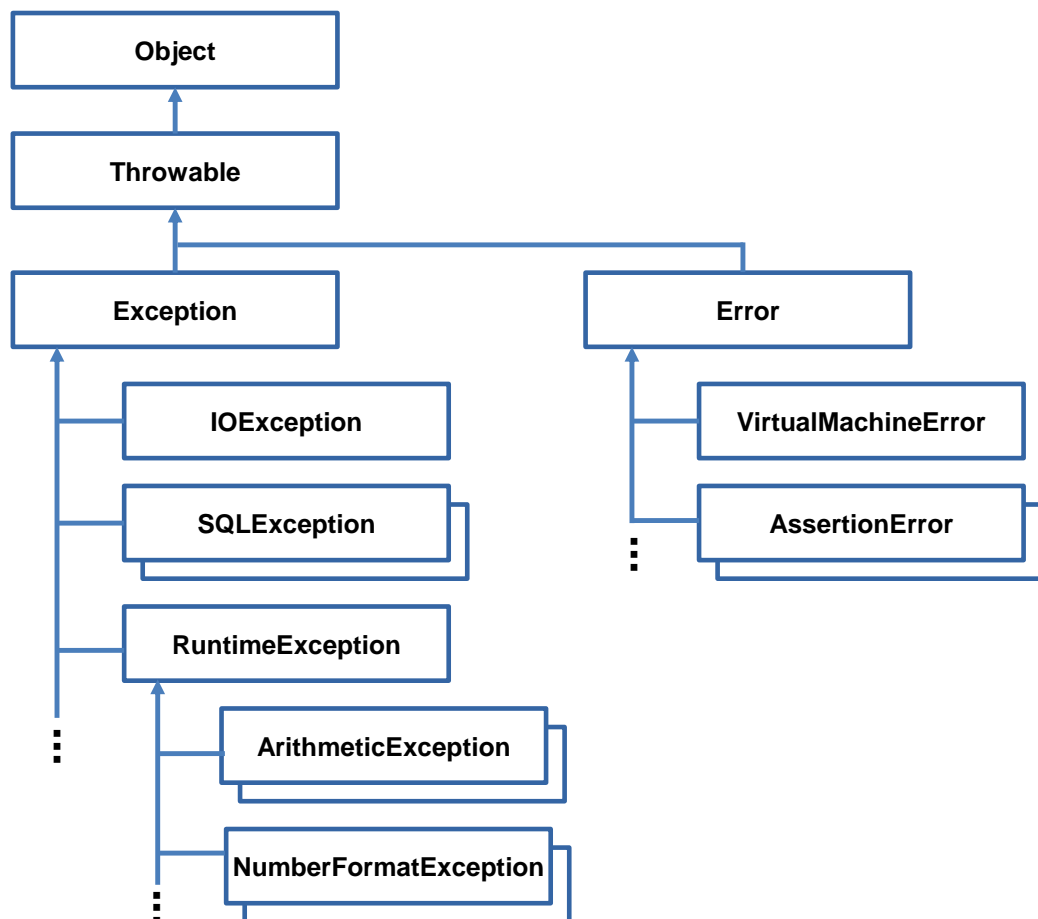


Figura 8.2: Hierarquia (parcial) das classes de exceção em Java

- Exceção não verificada ou exceção em tempo de execução: As classes que estendem *RuntimeException* são conhecidas como exceções não verificadas

3. Erro: São as exceções do tipo *Error* ou suas subclasses. Um *Error* é irrecuperável, por exemplo *OutOfMemoryError*, *VirtualMachineError*, *AssertionError* etc.

Além de lançar objetos cujas classes são declaradas em *java.lang*, podemos lançar objetos que nós mesmos criamos, estendendo uma classe de *Throwable*. Para criar nossa própria classe de objetos *Throwable*, só precisamos declará-la como uma subclasse de algum membro da família *Throwable*. Em geral nas classes *Throwable* que definimos, devemos estender a classe *Exception*.

Dependendo da situação, usamos uma classe de exceção existente de *java.lang* ou criamos uma. Em alguns casos, uma classe de *java.lang* vai funcionar muito bem. Por exemplo, se um método é chamado com um argumento inválido, podemos lançar *IllegalArgumentException*, sem precisar criar uma classe de exceção customizada. *IllegalArgumentException* é uma subclasse de *RuntimeException* em *java.lang*.

Outras vezes, no entanto, precisamos transmitir mais informações sobre a condição anormal que ocorre. Desta forma não podemos utilizar uma classe de *java.lang*. Normalmente, a própria classe do objeto de exceção indica o tipo de condição anormal que foi encontrado. Por exemplo, se o objeto de exceção lançado tem como classe *IllegalArgumentException*, sabemos que o método recebeu um argumento ilegal.

8.2 Lançando exceções

Para lançar uma exceção, podemos usar a palavra-chave *throw* com uma referência de objeto. O tipo da referência deve ser *Throwable* ou uma das suas subclasses. Desta forma o formato mostrado a seguir, onde *ThrowableClass* é a classe *Throwable* ou uma das suas subclasses.

```
throw new ThrowableClass();
```

Vamos ver como isto funciona com um exemplo. Vamos fazer um programa que calcula a raiz quadrada por um método de aproximação chamado método babilônico.

Listagem 8.1: RaizQuadrada.java

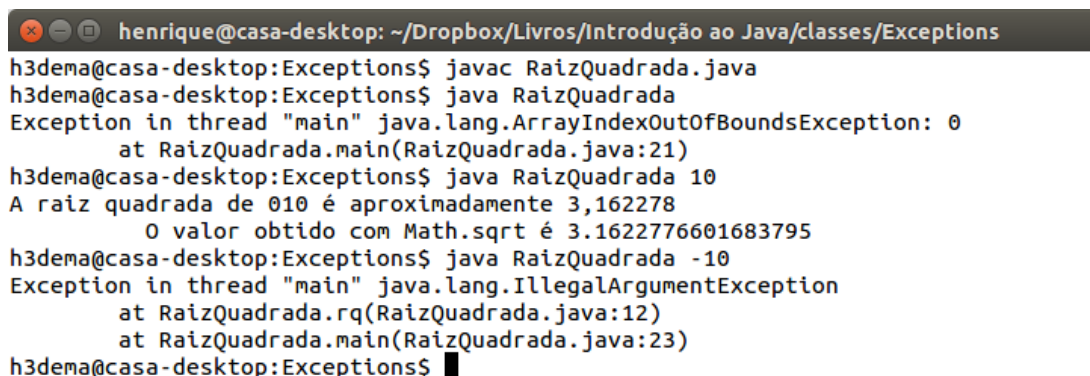
```
1 class RaizQuadrada {
2
3     private static int NUM_ITER = 50;
4
5     /**
6      * calcula a raiz quadrada usando Babylonian method
7      * veja mais em https://en.wikipedia.org/wiki/
8      *     Methods_of_computing_square_roots
9      * S deve ser um inteiro positivo
10    */
11    public static double rq(int S) throws IllegalArgumentException {
12        if (S < 0) throw new IllegalArgumentException();
13        double x = S / 2;
14        for(int i = 0; i < NUM_ITER; i++) {
15            x = (x + S / x) / 2;
16        }
17        return x;
18    }
19
20    public static void main(String[] args) {
21        int S = Integer.valueOf(args[0]);
22
23        double raiz = rq(S);
24        System.out.printf("A raiz quadrada de %03d é aproximadamente %f\n", S,
25                           raiz);
26        System.out.println("O valor obtido com Math.sqrt é " + Math.sqrt(S));
27    }
28 }
```

```

26     }
27 }

```

Vemos na figura 8.3 como ocorre a execução da classe definida na listagem 8.1. Note que quando não passamos nenhum parâmetro na saída mostrada na figura 8.3, ocorre um erro denominado *ArrayIndexOutOfBoundsException*. Isto ocorre porque na linha 23 tentamos converter o primeiro argumento de passagem para inteiro. Como não este argumento não foi passada não tem como o Java acessar `args[0]`. Quando tentamos chamar o programa para calcular a raiz quadrada de -10, a condição na linha 12 é verdadeira. Desta forma o nosso programa lança uma exceção *IllegalArgumentException*. Como esta exceção não é tratada em `main()`, aparece uma mensagem de erro do Java com resultado.



```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Exceptions
h3dema@casa-desktop:Exceptions$ javac RaizQuadrada.java
h3dema@casa-desktop:Exceptions$ java RaizQuadrada
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at RaizQuadrada.main(RaizQuadrada.java:21)
h3dema@casa-desktop:Exceptions$ java RaizQuadrada 10
A raiz quadrada de 010 é aproximadamente 3,162278
    O valor obtido com Math.sqrt é 3.1622776601683795
h3dema@casa-desktop:Exceptions$ java RaizQuadrada -10
Exception in thread "main" java.lang.IllegalArgumentException
    at RaizQuadrada.rq(RaizQuadrada.java:12)
    at RaizQuadrada.main(RaizQuadrada.java:23)
h3dema@casa-desktop:Exceptions$ █

```

Figura 8.3: Execução da classe RaizQuadrada

A tabela 8.1 destaca as diferenças entre *throw* e *throws*.

| # | throw | throws |
|---|---------------------------------------------------------------------------------|--------------------------------------------------------------------|
| 1 | A palavra chave <i>throw</i> é utilizada para lançar explicitamente uma exceção | A palavra chave <i>throws</i> é usada para declarar uma exceção. |
| 2 | Uma exceção verificada não pode ser propagada utilizando somente <i>throw</i> | Uma exceção verificada pode ser propagada utilizando <i>throws</i> |
| 3 | <i>throw</i> é seguido por uma instância de classe (um objeto) | <i>throws</i> é seguido por uma classe |
| 4 | <i>throw</i> é utilizada dentro de um método | <i>throws</i> é utilizada na assinatura de um método |
| 5 | Não podemos lançar múltiplas exceções com <i>throw</i> | Podemos declarar múltiplas exceções em um método |

Tabela 8.1: Diferença entre *throw* e *throws*

8.3 Tratando exceções

Para capturar uma exceção em Java, escrevemos um bloco *try* com uma ou mais cláusulas de captura, chamada cláusula *catch*. Cada cláusula *catch* especifica o tipo de exceção que está preparada para lidar.

```

try {
    // código que será executado
    // e cujas exceções serão capturadas pelas cláusulas catch

```

```
} catch (ThrowableClass1 tc1) {  
    // realiza algum tratamento da exceção  
}  
} catch (ThrowableClassn tcn) {  
    // realiza algum tratamento da exceção  
}
```

O bloco *try* delimita a parte do código que está sendo verificado. Sob este bloco funcionarão as cláusulas de captura. Se o pedaço de código delimitado pelo bloco *try* gera uma exceção, as cláusulas de captura associadas serão examinadas pela máquina virtual Java. Se a máquina virtual encontra uma cláusula *catch* que está preparado para lidar com a exceção lançada, o programa continua a execução começando com a primeira declaração de que cláusula *catch*. As cláusulas *catch* indicam o tipo de condição anormal elas lidam mediante a declaração do tipo de referência de exceção que tratam.

Vamos melhorar o nosso programa para que ele capture agora os erros que identificamos em *RaizQuadrada.java*. Este novo programa é mostrado na listagem 8.2. Note que agora acrescentamos 2 blocos *try*. O primeiro bloco iniciado na linha 22 serve para tratar os erros resultantes do argumento de entrada. Note que o bloco possui duas cláusulas de captura - uma para tratar o erro que ocorre não já inserido um argumento na linha de chamada do programa e outro para a conversão para inteiro. Na linha 33 temos outro bloco *try*. Este bloco trata a exceção que é gerada quando passamos um valor negativo para o método *rq()*. Os resultados da execução estão na figura 8.4.

Listagem 8.2: *RaizQuadrada2.java*

```
1 class RaizQuadrada2 {  
2  
3     private static int NUM_ITER = 50;  
4  
5     /**  
6      * calcula a raiz quadrada usando Babylonian method  
7      * veja mais em https://en.wikipedia.org/wiki/  
8          Methods\_of\_computing\_square\_roots  
9      * S deve ser um inteiro positivo  
10     */  
11     public static double rq(int S) throws IllegalArgumentException {  
12         if (S < 0) throw new IllegalArgumentException();  
13         double x = S / 2;  
14         for(int i = 0; i < NUM_ITER; i++) {  
15             x = (x + S / x) / 2;  
16         }  
17         return x;  
18     }  
19  
20     public static void main(String[] args) {  
21         int S = 0;  
22         try {  
23             S = Integer.valueOf(args[0]);  
24         } catch (ArrayIndexOutOfBoundsException e) {  
25             System.out.println("Erro: você deve passar como parâmetro o número de  
26                 deseja calcular a raiz quadrada.");  
27             return;  
28         } catch (NumberFormatException e) {  
29             System.out.println("Erro: deve ser fornecido um número inteiro.");  
30             return;  
31         }  
32  
33         double raiz;  
34         try {  
35             raiz = rq(S);  
36         } catch (IllegalArgumentException e) {
```

```

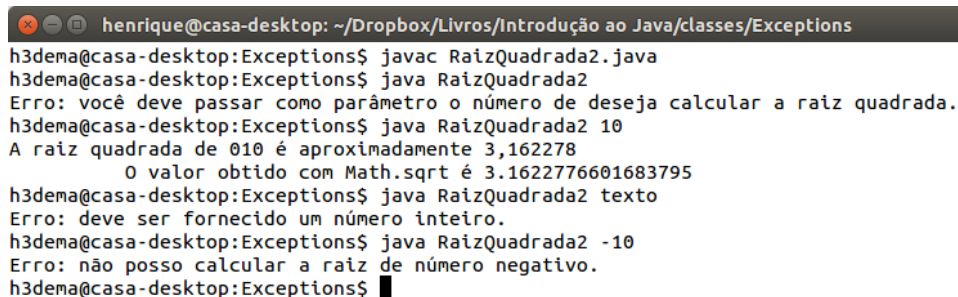
36         System.out.println("Erro: não posso calcular a raiz de número negativo.
37         ");
37         return;
38     }
39     System.out.printf("A raiz quadrada de %03d é aproximadamente %f\n", S,
40         raiz);
40     System.out.println("O valor obtido com Math.sqrt é " + Math.sqrt(S));
41 }
42
43 }

```

No código da listagem 8.2 vemos, por exemplo, na cláusula *catch* da linha 27, o caractere minúsculo “e”. Ele é uma referência para o objeto da *NumberFormatException* que é lançado pela instrução *Integer.valueOf()*. Esta referência poderia ter sido usada dentro da cláusula *catch*, embora, neste caso, não seja.

Se algum código dentro de um bloco *try* gera uma exceção, suas cláusulas de captura são examinados em sua ordem de codificação no arquivo de origem (*.java*). Por exemplo, se o bloco *try* da linha 22 no exemplo da listagem 8.2 gera uma exceção, a cláusula *catch* para *ArrayIndexOutOfBoundsException* será examinada em primeiro lugar, em seguida, a cláusula *catch* para *NumberFormatException*. A ordenação do exame das cláusulas *catch* é importante porque é possível que várias cláusulas de captura de um bloco *try* possam lidar com a mesma exceção. Reveja a figura 8.2. Vemos que *ArithmeticException* é uma classe derivada de *RuntimeException*. Suponha que um método lance uma exceção *ArithmeticException*. No bloco tratador de exceções, temos primeiro *RuntimeException* e depois *ArithmeticException*. Neste caso, a cláusula *catch* para *ArithmeticException* nunca será acionada, pois sempre a exceção sempre será tratada pela cláusula *catch* de *RuntimeException*.

A regra geral é: cláusulas *catch* de subclasses deve preceder cláusulas *catch* da sua superclasse.



```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Exceptions
h3dema@casa-desktop:Exceptions$ javac RaizQuadrada2.java
h3dema@casa-desktop:Exceptions$ java RaizQuadrada2
Erro: você deve passar como parâmetro o número de deseja calcular a raiz quadrada.
h3dema@casa-desktop:Exceptions$ java RaizQuadrada2 10
A raiz quadrada de 010 é aproximadamente 3,162278
O valor obtido com Math.sqrt é 3.1622776601683795
h3dema@casa-desktop:Exceptions$ java RaizQuadrada2 texto
Erro: deve ser fornecido um número inteiro.
h3dema@casa-desktop:Exceptions$ java RaizQuadrada2 -10
Erro: não posso calcular a raiz de número negativo.
h3dema@casa-desktop:Exceptions$

```

Figura 8.4: Execução da classe *RaizQuadrada2.java*

A partir do Java SE 7, uma única cláusula *catch* pode lidar com mais de um tipo de exceção. Esse recurso pode reduzir a duplicação de código e diminuir a tentação de capturar uma exceção excessivamente ampla. Na cláusula *catch*, podemos especificar os tipos de exceções que o bloco pode capturar e separar cada tipo de exceção com uma barra vertical. No exemplo abaixo remodelamos o bloco *try* da linha 22 no exemplo da listagem 8.2 para tratar simultaneamente das duas exceções.

```

try {
    S = Integer.valueOf(args[0]);
}
catch (ArrayIndexOutOfBoundsException | NumberFormatException e){
    System.out.println(

```



```
        "Erro: Parâmetro inválido. Deve ser inteiro não negativo.");  
    return;  
}
```


Capítulo 9

Estruturas de dados

9.1 Listas encadeadas

Uma lista encadeada (do inglês *linked list*) é uma representação de uma sequência de objetos, todos do mesmo tipo, na memória do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda e assim por diante. Cada célula contém um objeto de algum tipo e o endereço da célula seguinte. Cada célula é chamada de nó da lista. A lista é representada por um ponteiro para o primeiro nó e o ponteiro do último elemento é nulo.

```
public class No<T> {  
  
    No prox;  
    T valor;  
  
    public No(T valor) {  
        prox = null;  
        this.valor = valor;  
    }  
}
```

O Java possui duas implementações

`java.util.LinkedList`

9.2 Pilhas e Filas

9.3 Sets

9.4 Maps

9.5 Hash Tables

9.6 Binary Search Trees

9.7 Binary Tree Traversal

9.8 Priority Queues

9.9 Heaps

9.10 O algoritmo Heapsort

Capítulo 10

Algoritmos de ordenacao

10.1 Bubble Sort

10.2 Selection Sort

10.3 Merge Sort

10.4 Quick Sort

10.5 Pesquisa

10.6 Busca Binária

Capítulo 11

Interface gráfica

11.1 Criando janelas

11.1.1 Janelas

11.1.2 Paineis

11.1.3 Mensagens

11.2 Interagindo com o usuários

11.2.1 Botões

11.2.2 Entradas de seleção

11.2.3 Áreas de texto

11.2.4 Menus

Capítulo 12

Threads

Os programas são normalmente escritos que forma sequencial. Em Java todos os aplicativos começam sua execução a partir da primeira instrução do método *main()* da classe principal. A partir daí o programa é executado linha a linha, com a execução de saltos em função de condições e com a chamada de subrotinas na própria classe ou em outras classes. Assim em um dado momento existe somente uma instrução sendo executada, ou seja, temos um único fluxo de execução.

Contudo é possível criar diversos fluxos de execução, criando uma aplicação *multithreading*. A cada *thread* é designada uma parte de um programa. Uma *thread* pode ser executada paralelamente a outras *thread*. Todo programa em execução tem pelo menos uma *thread* rodando, a *thread* principal. Desta forma vemos que *threads* são parte integrante do funcionamento de nossos programas.

Então, por quê usar *threads*? O uso de *thread* permite que um programa faça mais de uma coisa ao mesmo tempo. Em um ambiente com um único processador existe pelo menos para o usuário a ilusão que diversas coisas são realizadas ao mesmo tempo. A criação de várias *threads* permite o processamento de uma tarefa de forma paralela e em um ambiente com múltiplos processadores ocorre realmente a execução simultânea de diversas tarefas.

12.1 Threads

Uma *thread*, algumas vezes chamada de processo leve (do inglês, *Lightweight Process*) é a unidade de execução. Uma *thread* executa seu código em paralelo a de outras *threads* em execução no momento. Quando você tem apenas um processador e existem *threads* em execução em paralelo, o usuário só tem a impressão de simultaneidade (*concurrency*). Mas quando temos múltiplos processadores, o poder de *multithreading* é utilizado em sua plenitude, pois temos as *threads* distribuídas nos processadores de computador.

Em Java, uma *thread* é uma instância da classe *java.lang.Thread*. Esta *thread* pode ser gerenciada: (a) como *thread* diretamente mapeada para uma *thread* nativa do sistema operacional ou (b) como uma *thread* da máquina virtual Java sendo executada de forma preemptiva. Quer saber um pouco mais sobre preemptividade, veja neste link da wikipedia.

Um sistema preemptivo é um sistema no qual as *threads* são gerenciados por um escalonador. Uma *thread* pode ser interrompida a qualquer momento para permitir que outra *thread* utilize o processador. Quando programamos, na maioria das vezes, não é necessário prestar atenção qual tipo de *thread* é usada. O resultado será normalmente o mesmo. Contudo podem haver diferenças entre os sistemas operacionais.

Existem três conceitos importantes para programação *multithreading*:

1. *Atomicidade*: Uma operação atômica não pode ser interrompida. A grande maioria das operações em Java não é atômica. Existem formas de fazer com que um conjunto de

operações possa ser realizado de forma atômica, utilizando sincronização. Veremos neste capítulo como fazer isso.

2. *Visibilidade*: Isso ocorre quando uma *thread* deve observar ações de uma ou mais *threads*, o que implica em algum tipo de sincronização entre elas.
3. *Ordem de execução*: Em um programa normal, que tem somente a *thread* principal, todas as linhas de código são executado na mesma ordem a cada vez que você iniciar o aplicativo (desde que as mesmas condições sejam verificadas). Este não é o caso quando usamos programação concorrente. O sistema operacional (ou a VM do Java) definem quanto tempo será executada cada *thread*. Isso pode mudar toda vez que o aplicativo é iniciado. Portanto, a ordem de execução não é garantida!.

Uma *thread* em Java pode assumir diversos estados, como podemos ver na figura 12.1. Em qualquer ponto na linha de tempo uma *thread* só pode estar em apenas um dos estados da figura. O estado “nova thread” é o estado que uma *thread* estão quando já foi criada mas ainda não começou a rodar. O estado executável pode ser considerado como um estado composto com dois subestados: “Pronta” e “Rodando”. Quando uma *thread* é passada para o estado “executável” pela primeira vez, a *thread* primeiro vai para o subestado “Pronta”. O escalonador de *threads* decide se a *thread* realmente irá começar, continuar ou ser suspensa. “Espera temporizada” é o estado no qual uma *thread*, não está executando e espera por um período de tempo especificado. A *thread* é colocada neste estado por um método, como por exemplo via *Thread.sleep(milissegundos)*. Uma *thread* está no estado “esperando” em função da chamada para um método, como por exemplo *Object.wait()*, *Thread.join()* ou *LockSupport.park()*. Uma *thread* está no estado “bloqueada” enquanto espera para bloqueio do monitor para entrar em um bloco ou método sincronizado. Depois que *thread* conclui a execução do método *run()*, a *thread* é movida para o estado “terminada”.

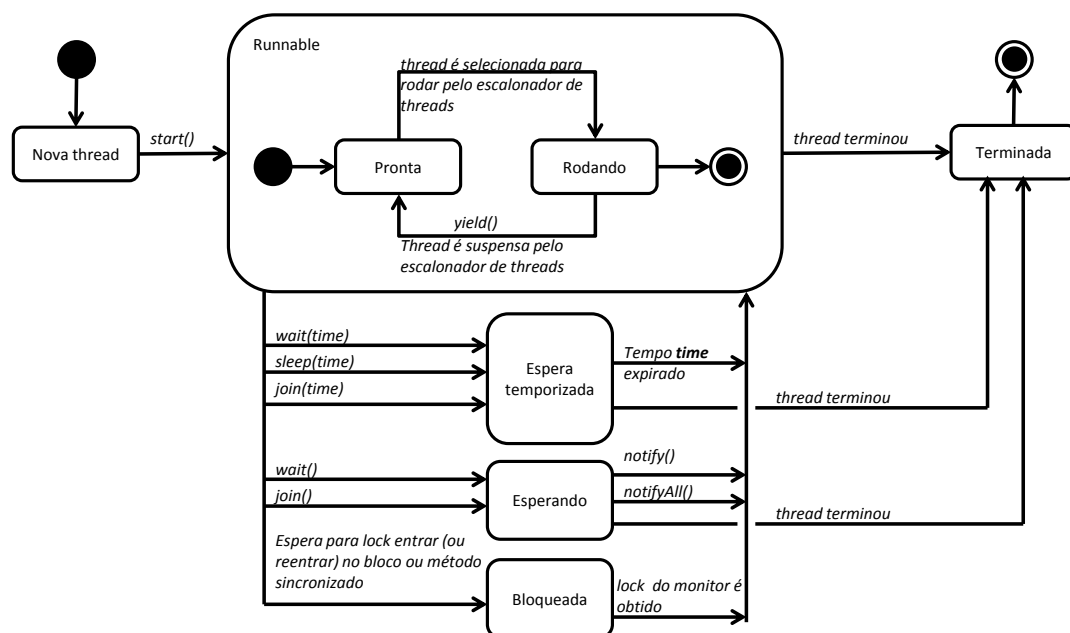


Figura 12.1: Estados possíveis para uma thread em Java

12.2 Criando *threads*

Podemos criar *threads* de duas maneiras: estendendo a classe *Thread* ou implementar a classe *Runnable* e enviar esta nova classe para um construtor de *Thread*. Podemos ainda criar uma classe *Thread* anônima e implementar o método *run()*.

A primeira solução não é considerada uma boa solução. Primeiro porque fere a ideia de extensão de classe, pois não estamos realmente criando uma nova *thread* especializado. Na verdade estamos somente implementando um conjunto de instruções. Esta é a ideia por trás da interface *Runnable*. A implementação usando *Runnable* é melhor também porque na verdade *Runnable* é uma interface e dessa forma podemos, para a nossa *thread*, realizar duas operações: estende uma classe que desejamos e ao mesmo tempo implementar a interface *Runnable*.

Vamos ver um exemplo simples. Na listagem abaixo criamos uma classe que implementa *Runnable*.

```
import java.lang.Runnable;

public class Exemplo1 implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 10; i++)
            System.out.println("Estamos em uma nova thread!");
    }
}
```

Para rodar a nossa *thread* vamos criar uma classe.

```
import java.lang.Thread;

class RodaExemplo1 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Exemplo1());
        thread.start();
        for(int i = 0; i < 10; i++)
            System.out.println("Estamos em na thread principal!");
    }
}
```

Podemos criar várias *thread* simultaneamente. Na classe *Exemplo2* utilizamos o atributo *id* para identificar qual a *thread*.

```
import java.lang.Runnable;

public class Exemplo2 implements Runnable {
    private int id;

    // construtor que recebe um número como identificador da thread
    Exemplo2(int id) { this.id = id; }

    @Override
    public void run() {
        for(int i = 0; i < 20; i++)
            System.out.println("Estamos em na thread "+id+ " !");
    }
}
```

Para rodar a nossa thread vamos criar uma classe RodaExemplo2. Esta classe cria agora 5 *threads* (além da principal).

```
import java.lang.Thread;

class RodaExemplo2 {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Thread thread = new Thread(new Exemplo2(i));
            thread.start();
        }
        for(int i = 0; i < 10; i++)
            System.out.println("Estamos em na thread principal!");
    }
}
```

Podemos fazer várias operações sobre uma Thread, como por exemplo: i) podemos fazer uma *thread* a dormir durante “x” milissegundos; ii) podemos espera que uma outra *thread* conclua seu processamento para continuarmos; iii) podemos gerenciar as prioridades das *threads* e pausar uma *thread* para dar oportunidade de outra *thread* executar iv) podemos interromper uma *thread*.

12.2.1 Thread.sleep()

Nós podemos fazer uma *thread* dormir por um certo número de milissegundos. Para isso, a classe Thread tem o método *sleep(long milissegundos)*. Este é um método permite fazer a *thread* corrente dormir. Não podemos escolher a *thread* que queremos fazer dormir.

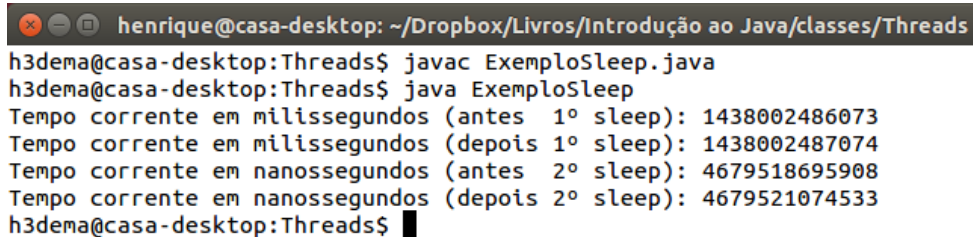
Vejamos o exemplo em 12.1. Na segunda parte da classe vemos que é possível fazer usar o método sleep com mais precisão, informando inclusive o período em nanossegundos. O formato deste método é método *sleep(long milissegundos, long nanossegundos)*. A saída para a execução de ExemploSleep é mostrada em 12.2.

Listagem 12.1: ExemploSleep.java

```
1 import java.lang.Thread;
2 import java.lang.InterruptedException;
3
4 public class ExemploSleep {
5
6     public static void main(String[] args) {
7         /**
8          * usando Thread.sleep(long milissegundos)
9          */
10        System.out.println("Tempo corrente em milissegundos (antes 1° sleep): "
11            + System.currentTimeMillis());
12        try {
13            Thread.sleep(1000); // dorme por 1000 milissegundos
14        } catch (InterruptedException e) {
15            e.printStackTrace();
16        }
17        System.out.println("Tempo corrente em milissegundos (depois 1° sleep): "
18            + System.currentTimeMillis());
19
20        /**
21         * usando Thread.sleep(long milissegundos, long nanossegundos)
22         */
23        System.out.println("Tempo corrente em nanossegundos (antes 2° sleep): "
24            + System.nanoTime());
25        try {
```

```
23         Thread.sleep(2, 5000); // dorme por 2 milissegundos e 5000
           nanossegundos
24     } catch (InterruptedException e) {
25         e.printStackTrace();
26     }
27     System.out.println("Tempo▯corrente▯em▯nanossegundos▯(depois▯2°▯sleep):▯"
           + System.nanoTime());
28 }
29 }
```

Notem que ao usarmos *Thread.sleep()* é necessário tratar a exceção *java.lang.InterruptedException*.



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Threads
h3dema@casa-desktop:Threads$ javac ExemploSleep.java
h3dema@casa-desktop:Threads$ java ExemploSleep
Tempo corrente em milissegundos (antes 1º sleep): 1438002486073
Tempo corrente em milissegundos (depois 1º sleep): 1438002487074
Tempo corrente em nanossegundos (antes 2º sleep): 4679518695908
Tempo corrente em nanossegundos (depois 2º sleep): 4679521074533
h3dema@casa-desktop:Threads$
```

Figura 12.2: Saída de ExemploSleep.java

12.2.2 Experando uma *thread* terminar para poder continuar

Pode fazer com uma *thread* espere por que outra *thread* termine. Por exemplo, podemos criar um conjunto de *threads* para realizar um cálculo complexo que pode ser desmembrado. Cada *thread* fará uma parte do cálculo. Contudo para achar o valor final é necessário ter em mãos o resultado de cada parte. Desta forma podemos criar o conjunto de *thread* e executá-las, mas precisamos esperar que todas tenham terminado de calcular seu pedaço para obter o resultado final.

Para fazer com que uma *thread* espere por outra, podemos usar o método *join()* da classe *Thread*. Este método permite a uma *thread* para esperar que a *thread* que está se unindo termine. Este método também lança *InterruptedException*, quando a *thread* for interrompida durante o processo de espera de outra *thread*.

Vamos ver um exemplo, onde esperamos que a *thread* *t* termine para podemos continuar. O código é apresentado em 12.2. O código fará com que a *thread* atual (no caso a *thread* principal) espere que a *thread* *t* termine. Note como a *thread* *t* foi criada.

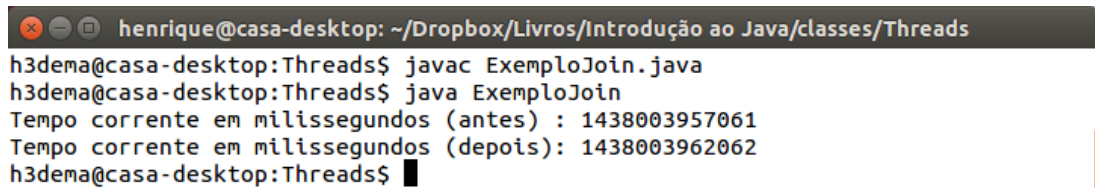
Listagem 12.2: ExemploJoin.java

```
1 import java.lang.Thread;
2 import java.lang.InterruptedException;
3
4 public class ExemploJoin {
5
6     public static void main(String[] args) {
7         Thread t = new Thread(){
8             @Override
9             public void run() {
10                 try {
11                     Thread.sleep(5000); // dorme por 5 segundos
12                 } catch (InterruptedException e) {
13                     e.printStackTrace();
14                 }
15             }
16         };
17         System.out.println("Tempo▯corrente▯em▯milissegundos▯(antes)▯:▯" + System.
           currentTimeMillis());
```

```

18     t.start(); // inicia a thread
19     try {
20         t.join();
21     } catch (InterruptedException e) {
22         e.printStackTrace();
23     }
24     System.out.println("Tempo corrente em milissegundos (depois): " + System.
        currentTimeMillis());
25 }
26 }

```



```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Threads
h3dema@casa-desktop:Threads$ javac ExemploJoin.java
h3dema@casa-desktop:Threads$ java ExemploJoin
Tempo corrente em milissegundos (antes) : 1438003957061
Tempo corrente em milissegundos (depois): 1438003962062
h3dema@casa-desktop:Threads$

```

Figura 12.3: Saída da execução de ExemploJoin.java

Pode adicionar um tempo limite em *join()*. Este tempo pode ser fornecido em milissegundos usando *join(long milissegundos)* ou com nanossegundos, usando *join(long milissegundos, long nanossegundos)*.

12.2.3 Priorizando threads

Quando trabalhamos com *threads* podemos alterar as prioridades de execução. O escalonador na máquina virtual Java, é baseado em prioridade. Portanto, se uma *thread* entrar em estado executável com uma prioridade maior do que uma outra *thread*, a nova *thread* será executado e a *thread* com menos prioridade aguardará sua vez. Esse comportamento não é garantido pois está depende da máquina virtual que o programa está rodando. Portanto, não é aconselhável depender das prioridades estabelecidas. O uso de prioridades deve ser feito pensando somente em melhorar a eficiência do seu programa. Normalmente, a gama de prioridade é um número inteiro que pode assumir valores de 0 a 10. Uma máquina virtual pode possuir faixas mais baixas ou mais elevadas para prioridade. Para conhecer a faixa de prioridade, podemos usar constantes da classe *Thread*: *Thread.MIN_PRIORITY*, *Thread.MAX_PRIORITY* e *Thread.NORM_PRIORITY*.

Para definir a prioridade de uma *thread*, usamos o método *setPriority(int prioridade)* da classe *Thread*. Se o valor informado for maior que a prioridade máxima, o valor máximo será usado. Se nada for informado, a prioridade utilizada será a prioridade da *thread* atual. Vemos um exemplo na listagem 12.3.

Listagem 12.3: ExemploPrioridade.java

```

1  import java.lang.InterruptedException;
2  import java.lang.Thread;
3  import java.lang.Runnable;
4
5  import java.util.Random;
6
7  public class ExemploPrioridade {
8
9      private static Random rnd = new Random();
10     static boolean rodar = true;
11
12     public static void main(String[] args) {
13         // cria conjunto de threads
14         Thread[] t = new Thread[10];

```

```
15     for(int i = 0; i < t.length; i++) {
16         t[i] = new Thread(new MinhaThread(i));
17     }
18
19     // sorteia qual thread tera maior prioridade
20     for(int i = 0; i < t.length; i++) {
21         if (rnd.nextInt(100) < 20 ) {
22             t[i].setPriority(Thread.MAX_PRIORITY);
23             break;
24         }
25     }
26     System.out.println("Priorizando threads");
27     for(int i = 0; i < t.length; i++) {
28         System.out.println("Thread "+i+": "+t[i].getPriority());
29     }
30     // inicia a execução
31     for(int i = 0; i < 10; i++) t[i].start();
32 }
33
34 static class MinhaThread implements Runnable {
35     int id;
36     MinhaThread(int id) { this.id = id; }
37
38     @Override
39     public void run() {
40         while(rodar) {
41             System.out.print(id);
42             rodar = rnd.nextInt(100) != 50; // critério de parada
43         }
44     }
45 }
46 }
```

12.3 Bloqueios intrínsecos

A sincronização em Java é uma maneira de fazer com que seu código seja seguro para *threads*. Um código que pode ser acessado por várias *threads* é dito *thread-safe*. Este código pode ser chamado de várias *threads* sem corromper o estado do objeto. O código é executado na ordem desejada. Por exemplo, vamos uma classe simples que permite recuperar um valor inteiro em sequencia, como no código abaixo:

```
public class Contador {
    private int valor = 0;

    public int obtemProximo() {
        return valor++;
    }
}
```

Este código simples pode apresentar problemas quando chamado por diversas *threads*. Isto acontece porque o “valor” pode ser atualizado primeira por duas *threads*, por exemplo. Só depois é retornado para as *threads*. Desta forma o valor pode ser retornado errado. Isto acontece porque as operações não são atômicas. Isto é *obtemProximo()* não é realizado completamente antes de outra *thread* chamá-lo. Em Java é simples resolver este problema.

```
public class Contador {
    private int valor = 0;

    public synchronized int obtemProximo() {
```



```
        synchronized (this) {  
            return valor++;  
        }  
    }  
}
```

Note que se você tiver em uma classe 2 métodos com *synchronized*, somente um dos métodos pode ser executado ao mesmo tempo. Isto ocorre porque o mesmo bloqueio é utilizada para os dois métodos. Quando usamos *synchronized*, se o método é *static*, o bloqueio está vinculado à classe que estamos criando. Neste caso mesmo que tenhamos diversos objetos desta classe, somente um poderá usar o método a cada momento.

É possível vincular o bloqueio a um outro objeto. Para isto não utilizamos o bloqueio intrínseco. O resultado obtido com o uso de *Lock* é semelhante ao obtido com *synchronized*. Vamos ver a classe *Contador* abaixo em um exemplo na listagem 12.4. Contudo neste caso podemos criar diversos bloqueios e permitir acesso a diversos métodos simultaneamente.

```
public class Contador {  
    private int valor = 0;  
  
    private final Object lock = new Object();  
  
    public int obtemProximo() {  
        synchronized (lock) {  
            return valor++;  
        }  
    }  
}
```

Listagem 12.4: TesteLock.java

```
1  class TesteLock {  
2  
3      public static void main(String[] args) {  
4  
5          System.out.println("Gerando numero com threads");  
6          for(int i = 0; i < 20; i++) {  
7              Thread t = new Thread() {  
8                  public void run() {  
9                      System.out.print(Contador.obtemProximo()+" ");  
10                 }  
11             };  
12             t.start();  
13         }  
14     }  
15  
16     public static class Contador {  
17         private static int valor = 0;  
18  
19         private static final Object lock = new Object();  
20  
21         public static int obtemProximo(){  
22             synchronized (lock) {  
23                 return valor++;  
24             }  
25         }  
26     }  
27 }
```

Há um outro problema quando utilizamos diversas *threads* que é a visibilidade das variáveis, ou seja, quando uma alteração feita por uma *thread*, esta seja visível por uma outra *thread*.

O compilador Java e suas máquinas virtuais utilizam algumas otimizações de desempenho, utilizando registros e cache. Desta forma, você não tem nenhuma garantia de que uma alteração feita por uma *thread* é visível por outra *thread*. Para fazer que uma mudança seja visível, devemos usar blocos sincronizados para assegurar a visibilidade da mudança. Devemos usar blocos sincronizados para a leitura e para a escrita dos valores compartilhados. Devemos fazer isso para cada leitura / gravação de um valor compartilhado entre vários segmentos.

Podemos também usar a palavra-chave *volatile* na definição da variável para assegurar a visibilidade de leitura/gravação entre as várias *threads*. *volatile* garante apenas a visibilidade, não a atomicidade. Os blocos sincronizados asseguram a visibilidade e a atomicidade. Assim podemos usar a palavra-chave volátil em campos que não necessitam de atomicidade.

É importante ficar atento ao utilizar estes recursos de bloqueio pois o uso indevido pode causar deadlocks.

12.4 Semáforos em Java

Semáforo é um conceito inventado Edsger Dijkstra. Basicamente um semáforo é um contador (inteiro) que permite que uma *thread* entre em uma região crítica se o valor deste contador é superior a 0. Em Java, um semáforo é criado usando a classe *java.util.concurrent.Semaphore*.

O semáforo é utilizado para sincronizar *threads*. O semáforo possui 3 operações: i) Inicialização: onde o semáforo recebe um valor inteiro indicando a quantidade de processos que podem acessar um determinado recurso; ii) : operação de *wait()*: também chamada por alguns autores de operação *P()*. Esta operação decrementa o valor do semáforo. Se o semáforo está com valor zero, o processo é posto para dormir; e iii) operação de *signal()*: também chamada por alguns autores de operação *V()*. Se o semáforo estiver com o valor zero e existir algum processo adormecido, um processo será acordado. Caso contrário, o valor do semáforo é incrementado.

Existe um semáforo especial, denominado mutex. Este semáforo funciona para evitar que dois processos ou *threads* tenham acesso simultaneamente a um recurso compartilhado. Este semáforo é binário, isto é, que só pode assumir dois valores distintos, 0 e 1. O travamento por semáforo deve ser feito antes de utilizar o recurso, e após o uso o recurso deve ser liberado. Enquanto o recurso estiver em uso, qualquer outro processo que o utilize deve esperar a liberação.

As operações *wait()* e *signal()* são realizadas utilizando os métodos *acquire()* e *release()*. O método *acquire()* pode ser interrompido se a *thread* é interrompido. Existe uma versão que não pode ser interrompida que utiliza o método *acquireUninterruptibly()*. Uma terceira versão deste método é *tryAcquire()* que adquirir acesso apenas se houver um disponível, caso contrário, este método irá retornar *false*. Todos os métodos de espera tem uma versão com passagem de um tempo limite.

Vamos ver na listagem 12.5, uma variação do nosso contador só que utilizando semáforos. Neste caso temos que obter o acesso à zona crítica (parte onde obtem e incrementa o valor do contador) usando a instrução *mutex.acquire()*. Se não existir uma *thread* na região crítica, o *mutex* estará em 1 e a *thread* atual pode continuar.

O mutex é criado usando *Semaphore mutex = new Semaphore(1)*.

Listagem 12.5: Variação da TesteLock.java usando mutex

```

1 import java.util.concurrent.Semaphore;
2 import java.lang.InterruptedException;
3
4 class TesteLock2 {
5
6     public static void main(String[] args) {
7
```

```
8      System.out.println("Gerando numero com threads");
9      for(int i = 0; i < 20; i++) {
10         Thread t = new Thread() {
11             public void run() {
12                 try {
13                     System.out.print(Contador.obtemProximo()+" ");
14                 } catch (InterruptedException e) {}
15             }
16         };
17         t.start();
18     }
19 }
20
21 public static class Contador {
22     private static int valor = 0;
23     /**
24      * cria um semáforo que funciona como um mutex
25      * a quantidade de acesso ao recurso é 1 (veja o new).
26      * isto garante que será um semáforo binário
27      */
28     private static final Semaphore mutex = new Semaphore(1);
29
30     public static int obtemProximo() throws InterruptedException {
31         try {
32             mutex.acquire(); // passo 1 - wait() que em Java é acquire
33             return valor++; // quando semáforo é 1, esta instrução é executada
34         } finally {
35             mutex.release(); // passo final - signal() que em Java é release
36         }
37     }
38 }
39 }
```

Um grande problema com semáforo é que podemos:

- i) esquecer de efetuar o `.release`;
- ii) inverter a sequência entre `.acquire` e `.release`
- iii) entrar em deadlock principalmente quando várias semáforos estão envolvidos.

Se você quiser saber mais sobre semáforos, recomendo o site [The Little Book of Semaphores](#).

12.5 Tipos atômicos em Java

A partir do Java versão 5 foram adicionados diversas variáveis atômicas. Estas classes permitem realizar operações atômicas para valores do tipo inteiro, longo, boolean e referência. Temos ainda versões em array para estes tipos. As classes em Java são : `AtomicBoolean`, `AtomicInteger`, `AtomicLong` e `AtomicReference`. Estas classes suportam o acesso *multi-threaded* e têm uma escalabilidade melhor do que a sincronização de todas as operações

As operações básicas são:

- `compareAndSet(T valorEsperado, T novoValor)`: define atômicamente o valor da variável do tipo T para o “novoValor”, se o valor atual for igual ao “valorEsperado”;
- `getAndSet(T novoValor)`: define atômicamente o valor da variável do tipo T para o “novoValor” e retorna o valor anterior;
- `get()`: obtém atômicamente o valor atual da variável; e
- `set(T novoValor)`: define atômicamente o valor da variável do tipo T para o “novoValor”.

Nas expressões acima *T* corresponde a uma das classes *AtomicBoolean*, *AtomicInteger*, *AtomicLong* ou *AtomicReference*. Existe operações específicas para cada tipo, por exemplo, *AtomicInteger* possui métodos para incrementar ou decrementar o valor atual.

Vamos ver como implementaríamos a classe *TesteLock* que usamos nos exemplos anteriores, só que usando *AtomicInteger*. A classe remodelada está na listagem 12.6. A primeira coisa que notamos é que não precisamos mais da classe interna *Contador*. A declaração da variável que contem o valor é similar a que faríamos para *java.lang.Integer*. No *new* passamos o valor inicial que é zero. Para obter o valor usamos um dos métodos de *AtomicInteger*, que é *getAndIncrement()*. O método *getAndIncrement()* incrementa atomicamente o valor da variável e retorna o valor anterior.

Listagem 12.6: Variação da *TesteLock.java* usando *AtomicInteger*

```

1  import java.util.concurrent.atomic.AtomicInteger;
2  import java.lang.InterruptedException;
3
4  class TesteLock3 {
5
6      private static AtomicInteger valor = new AtomicInteger( 0 );
7
8      public static void main(String[] args) {
9
10         System.out.println("Gerando numero com threads");
11         for(int i = 0; i < 20; i++) {
12             Thread t = new Thread() {
13                 public void run() {
14                     System.out.print(valor.getAndIncrement()+" ");
15                 }
16             };
17             t.start();
18         }
19     }
20 }
```

12.6 Monitores em Java ou o problema do Jantar dos Filósofos

Este problema é encontrado em muitos textos de computação. Você pode ler mais sobre ele na wikipedia. Mas eu fiquei curioso quando vi uma explicação no livro de Fundamentos de Sistemas Operacionais do Silberschatz. Ele apresenta o conceito de monitores.

Os monitores são mecanismo de programação concorrente. Um monitor é um mecanismo de mais alto nível do que os semáforos e também mais poderoso. Um monitor é uma instância de uma classe que pode ser usada com segurança por várias *threads*. Todos os métodos de um monitor são executados com exclusão mútua. Assim, no máximo, uma *thread* pode executar um método do monitor ao mesmo tempo - as demais *threads* tem que ficar esperando. Esta política de exclusão mútua torna mais fácil trabalhar com o monitor. Pois é muito mais fácil desenvolver o conteúdo do método do monitor.

Os monitores têm uma outra característica importante, podemos fazer com que uma *thread* fique à espera de uma condição. Durante o tempo de espera, a *thread* entra em espera (fica dormindo) e o seu acesso exclusivo é cedido para outra *thread*. A *thread* que está esperando readquire o acesso ao ponto onde ficou parada, depois que a condição foi cumprida. Você também pode sinalizar um ou mais tópicos que uma condição foi cumprida. Desta forma:

- Todo o código de sincronização é centralizado em um único local. Os usuários deste código não precisam saber como ele é implementado.

- O código não depende do número de processos. Ele funciona para tantos processos quantos você quiser.
- Você não precisa liberar algo como um *mutex*. Desta forma não há como você esquecer de fazê-lo.

Teoricamente e é assim que está no livro do Silberschatz, para criar um monitor, você simplesmente utiliza uma palavra chave *monitor*, no lugar onde normalmente colocaríamos *class*, e teríamos pronta a nossa classe como mostrado na listagem abaixo:

```
monitor MonitorTeorico {
    public method void procedimentoA () {
        //seu codigo
    }

    public method int procedimentoB () {
        int valor;
        //seu codigo
        return valor;
    }
}
```

Podemos criar uma variável de condição. Teoricamente usaríamos uma palavra-chave como *cond*. Uma variável de condição é uma espécie de fila com os processos que estão esperando na mesma condição. Você tem várias operações disponíveis em uma condição - os mais importantes são sinalizar para um processo que ele pode ser despertado ou deve esperar pela condição. Existem algumas semelhanças entre as operações de *signal/wait* dos semáforos. A operação *signal* da condição não faz nada se a fila está vazia. A operação de *wait* coloca a *thread* na fila de espera. A fila de processos é atendida usando First-come, first-served - FCFS. Quando a *thread* desperta depois de esperar em uma condição, ela readquire o bloqueio antes de continuar no código. Existem diversas formas de sinalização em teoria, contudo o Java só oferece um denominado Signal & Continue (SC). Nele o processo que sinalizar mantém a exclusão mútua e o processo que recebe o sinal será despertar mas precisa adquirir a exclusão mútua antes de continuar.

Em Java não há nenhuma palavra-chave para criar diretamente um monitor. Para implementar um monitor, você deve criar uma nova classe e usar classes de bloqueio e classe de condições. Estas classes estão na biblioteca *java.util.concurrent.locks* e são denominadas respectivamente *Lock* e *Condition*.

Lock é uma interface. Uma implementação desta interface (que usaremos no exemplo) é *ReentrantLock*. *ReentrantLock* possui dois construtores: *ReentrantLock()*, construtor padrão, e *ReentrantLock(boolean fair)*, construtor com um argumento booleano que indica se o bloqueio é justo ou não. Um bloqueio justo indica que as *threads* irão adquirir os bloqueios na ordem que eles solicitarem. Este efeito de justiça carrega bem mais o processamento. Para adquirir o bloqueio, você apenas tem que usar o método de *lock()* e *unlock()* para liberá-lo. Desta forma para implementar a listagem que mostramos antes em Java temos um código semelhante ao abaixo:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.lang.InterruptedException;

class MonitorEmJava {
    private final Lock lock = new ReentrantLock();
```

```

public method void procedimentoA() throws InterruptedException {
    lock.lock();
    try {
        //seu codigo
    } finally {
        lock.unlock();
    }
}

public method int procedimentoB() throws InterruptedException {
    lock.lock();
    try {
        int valor;
        //seu codigo
        return valor;
    } finally {
        lock.unlock();
    }
}
}

```

O código acima pode ser criado usando os recursos de *synchronized blocks*. Contudo *synchronized blocks* não permite criar as variáveis de condição. É possível criar condições usando o método *newCondition* da classe *Lock*. A condição criada desta forma é uma variável do tipo *Condition*. Você pode fazer a *thread* esperar pela condição utilizando o método *await()* e suas variantes com tempos limites. Você pode sinalizar usando *signal()*, para acordar uma única *thread* da fila, ou *signalAll()*, que acorda todas as *thread* em espera na variável de condição.

12.6.1 Implementando Dining Philosophers

Vamos ver então como fica a implementação do problema do jantar dos filósofos descrito no livro do Silberschatz.

Listagem 12.7: Enumeração

```

1 public enum EstadoFilosofo { THINKING, EATING, HUNGRY };

```

Listagem 12.8: DininigPhilosophers.java

```

1 import java.util.concurrent.locks.Condition;
2 import java.util.concurrent.locks.Lock;
3 import java.util.concurrent.locks.ReentrantLock;
4 import java.lang.InterruptedException;
5
6 class DininigPhilosophers {
7
8     private final Lock lock = new ReentrantLock();
9
10    private int num_filosofos;
11    private EstadoFilosofo[] estados;
12    private Condition[] cond;
13
14    private int esquerda(int i) { return (i + num_filosofos - 1) %
        num_filosofos; }
15    private int direita(int i) { return (i + + 1) % num_filosofos; }
16
17    DininigPhilosophers(int num) {
18        num_filosofos = num;

```

```
19     estados = new EstadoFilosofo[num];
20     cond = new Condition[num];
21     for(int i = 0; i < num; i++) {
22         cond[i] = lock.newCondition();
23         estados[i] = EstadoFilosofo.THINKING;
24     }
25 }
26
27 /**
28  * deseja comer
29  */
30 void pickup(int i) throws InterruptedException {
31     lock.lock();
32     try {
33         estados[i] = EstadoFilosofo.HUNGRY;
34         test(i);
35         if (estados[i] != EstadoFilosofo.EATING) {
36             cond[i].await();
37         }
38     } finally {
39         lock.unlock();
40     }
41 }
42
43 /**
44  * acabou de comer
45  */
46 void pulldown(int i) throws InterruptedException {
47     lock.lock();
48     try {
49         estados[i] = EstadoFilosofo.THINKING;
50         // verifica vizinhos
51         test( esquerda(i) );
52         test( direita(i) );
53     } finally {
54         lock.unlock();
55     }
56 }
57
58 void test(int i) throws InterruptedException {
59     lock.lock();
60     try {
61         if (estados[esquerda(i)] != EstadoFilosofo.EATING &&
62             estados[direita(i)] != EstadoFilosofo.EATING &&
63             estados[i] == EstadoFilosofo.HUNGRY) {
64             estados[i] = EstadoFilosofo.EATING;
65             cond[i].signal();
66         }
67     } finally {
68         lock.unlock();
69     }
70 }
71
72 public void estados(int rodada) {
73     lock.lock();
74     try {
75         System.out.printf("%3d┐┐", rodada);
76         for(int i=0; i < estados.length; i++) {
77             System.out.printf("%8s┐", estados[i].name());
78         }
79         System.out.println("");
80     } finally {
81         lock.unlock();
82     }
83 }
84
```

85 }

Listagem 12.9: Classe de exemplo - RunDP.java

```

1  import java.lang.Thread;
2  import java.lang.InterruptedException;
3  import java.util.concurrent.atomic.AtomicInteger;
4
5  class RunDP {
6
7      public static final int NUM_FILOSOFOS = 5;
8      boolean rodar = true;
9      DininigPhilosophers dp;
10
11      AtomicInteger rodada = new AtomicInteger(0);
12
13      public void init() {
14          dp = new DininigPhilosophers(NUM_FILOSOFOS);
15
16          for(int i = 0; i < NUM_FILOSOFOS; i++) {
17              Thread t = new Thread(new Filosofo(i));
18              t.start();
19          }
20      }
21
22      public static void main(String[] args) {
23          RunDP run = new RunDP();
24          run.init();
25      }
26
27      public class Filosofo implements Runnable {
28          private int id;
29
30          Filosofo(int id) { this.id = id; }
31
32          public void run() {
33              while (rodar) {
34                  try {
35                      dp.estados(rodada.get());
36                      dp.pickup(id);
37                  } catch (InterruptedException e) {}
38                  try {
39                      Thread.sleep(100);
40                  } catch (InterruptedException e) {}
41                  try {
42                      dp.estados(rodada.get());
43                      dp.pulldown(id);
44                  } catch (InterruptedException e) {}
45                  rodada.incrementAndGet();
46                  rodar = rodada.get() <= 50;
47              }
48          }
49      }
50
51  }

```

12.7 Grupos de *threads*

Grupos de *threads* de trabalho podem ser gerenciadas como *Thread pools*. Uma *thread pool* mantém uma fila com as *threads* cujas tarefas estão à espera de ser executados. A *thread pool* pode ser descrita como uma coleção de objetos do tipo *Runnable* e outra coleção de *threads* em


```

henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/DiningPhilosophers
h3dema@casa-desktop:DiningPhilosophers$ javac DininigPhilosophers.java RunDP.java EstadoFilosofo.java
h3dema@casa-desktop:DiningPhilosophers$ java RunDP
0 [ THINKING THINKING THINKING THINKING THINKING ]
0 [ THINKING EATING THINKING THINKING THINKING ]
0 [ THINKING EATING HUNGRY THINKING THINKING ]
0 [ HUNGRY EATING HUNGRY THINKING THINKING ]
0 [ HUNGRY EATING HUNGRY EATING THINKING ]
0 [ HUNGRY EATING HUNGRY EATING HUNGRY ]
1 [ EATING THINKING HUNGRY EATING HUNGRY ]
1 [ EATING HUNGRY HUNGRY EATING HUNGRY ]
2 [ EATING HUNGRY EATING THINKING HUNGRY ]
2 [ EATING HUNGRY EATING HUNGRY HUNGRY ]
3 [ THINKING HUNGRY EATING HUNGRY EATING ]
3 [ HUNGRY HUNGRY EATING HUNGRY EATING ]
4 [ HUNGRY EATING THINKING HUNGRY EATING ]
4 [ HUNGRY EATING HUNGRY HUNGRY EATING ]
5 [ HUNGRY EATING HUNGRY EATING THINKING ]
5 [ HUNGRY EATING HUNGRY EATING HUNGRY ]
6 [ EATING THINKING HUNGRY EATING HUNGRY ]
6 [ EATING HUNGRY HUNGRY EATING HUNGRY ]
7 [ EATING HUNGRY EATING THINKING HUNGRY ]
6 [ EATING HUNGRY EATING HUNGRY HUNGRY ]
8 [ THINKING HUNGRY EATING HUNGRY EATING ]

```

Figura 12.5: Saída paraRunDP.java

execução. Uma *thread pool* pode ser representada por uma instância de *ExecutorService*. Utilizando *ExecutorService* é possível submeter tarefas (*Runnable*) que são executadas no futuro.

A classe *java.util.concurrent.Executors* permite criar diversos tipos de *thread pools*. Os diferentes métodos para criação de *thread pools* mostrados abaixo retorna um *ExecutorService*. Um *Executor* fornece métodos para gerenciar a finalização de uma tarefa. Fornece ainda métodos que podem produzir objeto do tipo *Future* que nos permite acompanhar o progresso de uma ou mais tarefas assíncronas.

- Executor de *thread* única: O *pool* possui apenas uma *thread* de trabalho. Toda a tarefa submetida a este executor será rodada sequencialmente. O método que cria este tipo é *Executors.newSingleThreadExecutor()*.
- *Pool* de *threads* em cache: Este tipo de *pool* cria tantas *threads* quantas forem necessárias para executar as tarefas em paralelo. Estes pools normalmente melhoram o desempenho dos programas que executam muitas *threads* assíncronas e de curta duração. As *threads* disponíveis serão reutilizados para as novas tarefas. As chamadas para o método executar reutilizará *threads* previamente construídos, se disponíveis. Se nenhum *thread* está disponível, uma nova *thread* será criado e adicionada ao *pool*. *Threads* que não tenham sido utilizadas durante sessenta segundos são encerradas e removidas do cache. Assim, uma *pool* que permanece inativo por um tempo suficientemente grande para não consome recursos. Este *pool* pode ser criado chamando *Executors.newCachedThreadPool()*.
- *Pool* de *threads* fixas: Este *pool* possui um número fixo de *threads*. Se uma *thread* não está disponível para a tarefa, a tarefa é colocada na fila de espera, aguardando que uma *thread* fique disponível. A criação deste *pool* é feita chamando *Executors.newFixedThreadPool(n)*, onde “n” é o número de *threads*. As *threads* existirão no *pool* até que sejam explicitamente terminadas usando *shutdown()*.

- *Pool de threads programadas*: Este *pool* permite que as tarefas sejam agendadas para execução em um período futuro. Este *pool* é criado usando `Executors.newScheduledThreadPool()`.
- *Pool com thread única programada*: Este *pool* permite que as tarefas sejam agendadas para execução em um período futuro utilizando uma única *thread*. Este *pool* é criado usando `Executors.newSingleThreadScheduledExecutor()`.

Depois que o *pool* é criado, podemos enviar para ele tarefas, usando os diferentes métodos de enviar. Você pode enviar para o pool um objeto do tipo *Runnable* ou do tipo *Callable*. Vamos ver um exemplo de utilização de *Executors*. Na listagem 12.6 mostramos uma classe que invoca 10 threads. As threads não fazem nada exceto imprimir um texto na tela e retornar uma *String* para *Future*. Note que a execução das threads é iniciada imediatamente e as saídas de *main()* e das *threads* se misturam na tela (figura 12.6). A ordem das respostas está relacionada com a finalização da *thread* e não com a ordem de execução.

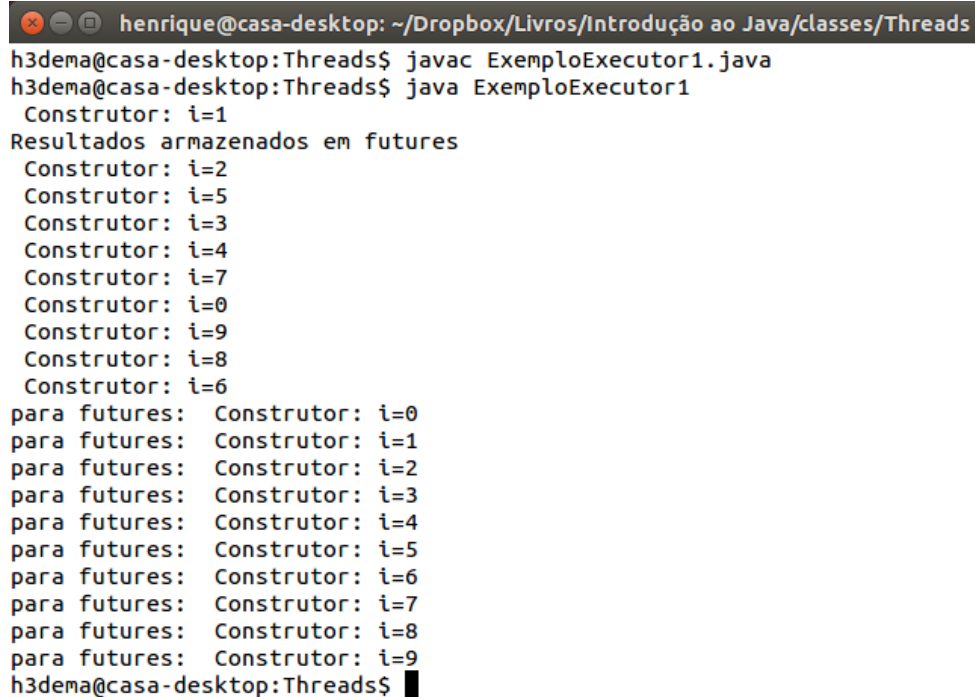
Listagem 12.10: Primeiro exemplo usando Executors

```

1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Future;
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.concurrent.Executors;
6  import java.util.concurrent.Callable;
7  import java.lang.InterruptedException;
8  import java.util.concurrent.ExecutionException;
9
10 class ExemploExecutor1 {
11
12     public static void main(String[] args) {
13
14         final int NUM_TAREFAS = 10;
15
16         ExecutorService pool = Executors.newFixedThreadPool(4); // cria pool com
17                               4 threads
18         List<Future<String>> futures = new ArrayList<Future<String>>(NUM_TAREFAS)
19                               ;
20         for(int i = 0; i < NUM_TAREFAS; i++){
21             Future<String> f = pool.submit(new MostraAlgumaCoisa<String>("i="+i));
22             futures.add(f);
23         }
24         pool.shutdown(); // para garantir que o pool será finalizado
25
26         System.out.println("Resultados armazenados em futures");
27         for(Future<String> f : futures){
28             try {
29                 String s = f.get();
30                 System.out.println(s);
31             }
32             catch(InterruptedException e) {}
33             catch(ExecutionException e) {}
34         }
35     }
36
37     private static final class MostraAlgumaCoisa<T> implements Callable<String>
38     {
39         private T v;
40
41         MostraAlgumaCoisa(T v) {
42             this.v = v;
43         }
44
45         public String call(){
46             String resultado = "Construtor: "+v;

```

```
45         System.out.println(resultado);
46         return "para_futures: "+resultado;
47     }
48 }
49
50 }
```



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Threads
h3dema@casa-desktop:Threads$ javac ExemploExecutor1.java
h3dema@casa-desktop:Threads$ java ExemploExecutor1
Construtor: i=1
Resultados armazenados em futures
Construtor: i=2
Construtor: i=5
Construtor: i=3
Construtor: i=4
Construtor: i=7
Construtor: i=0
Construtor: i=9
Construtor: i=8
Construtor: i=6
para futures: Construtor: i=0
para futures: Construtor: i=1
para futures: Construtor: i=2
para futures: Construtor: i=3
para futures: Construtor: i=4
para futures: Construtor: i=5
para futures: Construtor: i=6
para futures: Construtor: i=7
para futures: Construtor: i=8
para futures: Construtor: i=9
h3dema@casa-desktop:Threads$
```

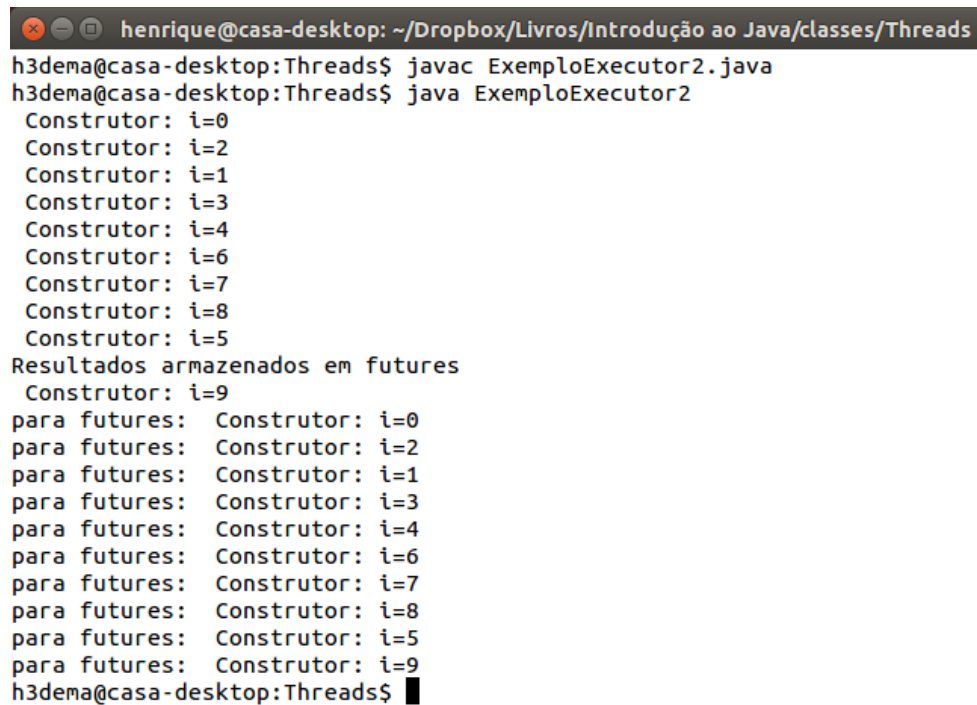
Figura 12.6: Saída ExemploExecutor1.java

O Java fornece uma classe adicional denominada `CompletableFuture`. Esta classe usa um `Executor` fornecido para executar tarefas. Esta classe arranja para que as tarefas submetidas sejam, após a conclusão, colocadas em uma fila acessível usando `take`. Com ela podemos reimplementar o nosso exemplo da listagem 12.10, como mostramos na listagem 12.11. Nesta nova implementação temos o resultado na ordem em que as tarefas são concluídas e não precisamos manter uma coleção de *Future*.

Listagem 12.11: Variação do exemplo usando Executors

```
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3 import java.util.concurrent.Callable;
4 import java.util.concurrent.CompletableFuture;
5 import java.util.concurrent.ExecutorCompletionService;
6 import java.lang.InterruptedException;
7 import java.util.concurrent.ExecutionException;
8
9 class ExemploExecutor2 {
10
11     public static void main(String[] args) {
12
13         final int NUM_TAREFAS = 10;
14
15         ExecutorService threadPool = Executors.newFixedThreadPool(4); // cria
            pool com 4 threads
```

```
16      CompletionService<String> pool = new ExecutorCompletionService<String>(
17          threadPool);
18      for(int i = 0; i < NUM_TAREFAS; i++){
19          pool.submit(new MostraAlgumaCoisa<String>("i="+i));
20      }
21      threadPool.shutdown(); // para garantir que o pool será finalizado
22
23      System.out.println("Resultados armazenados em futures");
24      for(int i = 0; i < NUM_TAREFAS; i++){
25          try {
26              String s = pool.take().get();
27              System.out.println(s);
28          }
29          catch(InterruptedException e) {}
30          catch(ExecutionException e) {}
31      }
32  }
33
34  private static final class MostraAlgumaCoisa<T> implements Callable<String>
35  {
36      private T v;
37
38      MostraAlgumaCoisa(T v) {
39          this.v = v;
40      }
41
42      public String call(){
43          String resultado = "Construtor:" + v;
44          System.out.println(resultado);
45          return "para futures:" + resultado;
46      }
47  }
48 }
```



```
henrique@casa-desktop: ~/Dropbox/Livros/Introdução ao Java/classes/Threads
h3dema@casa-desktop:Threads$ javac ExemploExecutor2.java
h3dema@casa-desktop:Threads$ java ExemploExecutor2
Construtor: i=0
Construtor: i=2
Construtor: i=1
Construtor: i=3
Construtor: i=4
Construtor: i=6
Construtor: i=7
Construtor: i=8
Construtor: i=5
Resultados armazenados em futures
Construtor: i=9
para futures: Construtor: i=0
para futures: Construtor: i=2
para futures: Construtor: i=1
para futures: Construtor: i=3
para futures: Construtor: i=4
para futures: Construtor: i=6
para futures: Construtor: i=7
para futures: Construtor: i=8
para futures: Construtor: i=5
para futures: Construtor: i=9
h3dema@casa-desktop:Threads$
```

Figura 12.7: Saída ExemploExecutor2.java