LINGUAGENS DE PROGRAMAÇÃO

AUTOR
FABIANO SANTOS

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2015

Estácio

Conselho editorial REGIANE BURGER; ROBERTO PAES; GLADIS LINHARES; KAREN BORTOLOTI; HELCIMARA AFFONSO DE SOUZA

Autor do original FABIANO GONÇALVES DOS SANTOS

Projeto editorial ROBERTO PAES

Coordenação de produção GLADIS LINHARES

Coordenação de produção EaD KAREN FERNANDA BORTOLOTI

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística AMANDA CARLA DUARTE AGUIAR

Imagem de capa KINETICIMAGERY | DREAMSTIME.COM

Dados Internacionais de Catalogação na Publicação (CIP)

S237L SANTOS, FABIANO

Linguagem de programação / Fabiano Santos.

Rio de Janeiro : SESES, 2015.

216 P. : IL.

ISBN: 978-85-5548-001-0

1. Algoritmos. 2. Lógica de programação. 3. Introdução à programação. 4. Linguagem. I. SESES. II. Estácio.

CDD 005.1

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

A área de desenvolvimento de sistemas está em constante atualização visando sempre o aumento da qualidade de aplicações e a velocidade de desenvolvimento sem perda de qualidade.

O desenvolvimento de soluções baseadas nos princípios da orientação a objetos é a base para que estes dois objetivos sejam atingidos. Este modelo de programação ganha espaço a cada dia por refletir a forma como nos relacionamos com o mundo real permitindo soluções mais intuitivas.

Essa realidade aumenta a importância da disciplina de Linguagem de Programação pois será o primeiro contato com este paradigma.

Nesta disciplina estudaremos o paradigma de programação orientado a objeto relacionado com a estrutura de programação orientada a objetos usando a linguagem Java. Nesta linguagem aprenderemos os principais elementos de construção/utilização de aplicativos com interface gráfica.

Como objetivos, esta disciplina visa apresentar você ao paradigma de programação orientado a objetos, tornando-o apto a se aprofundar em qualquer linguagem que utilize este tipo de programação. A partir dos conhecimentos aqui adquiridos, você estará mais preparado para o mercado de trabalho, onde a grande maioria dos sistemas é desenvolvida neste paradigma.

Além disso, você irá compreender os conceitos programação orientada a objetos e poderá desenvolver aplicativos utilizando este paradigma.

Bons estudos!

1

Introdução à Linguagem de Programação

Estudar uma linguagem de programação nova é como aprender um novo idioma: temos que entender seu vocabulário, sintaxe e semântica. E com a linguagem Java não é diferente.

A linguagem Java possui várias formas de realizar as tarefas e vamos apresentar neste capítulo algumas das maneiras principais. Conhecer os blocos de estruturação e controle de programas é fundamental e esperamos que este capítulo te ajude nisso.

Porém é importante que você aprimore seus conhecimentos com muita prática e estudos em outras fontes de informação. Só assim você vai se tornar fluente na linguagem.

Vamos lá?

OBJETIVOS

Neste capítulo vamos estudar os seguintes assuntos:

- Introdução à linguagem Java;
- tipos de dados primitivos;
- Estruturas de decisão e repetição
- Strings;
- Vetores;
- Conversão simples de tipos.

1.1 Introdução à linguagem de programação

Antes de tratarmos sobre a linguagem Java, é importante abordar sobre as linguagens de programação em geral.

Sabemos que os computadores atuais são dispositivos digitais e internamente só entendem duas coisas: 0 e 1 (bits). Os bits formam o que é chamado de linguagem de máquina e cada computador possui seu conjunto próprio de instruções. Portanto, computadores diferentes possuem **linguagens de máquina** diferentes.

Programar um computador em linguagem de máquina é muito complexo, ainda mais nos dias atuais devido à grande quantidade de diferentes tipos de computadores. Veja como seria um antigo programa em linguagem de máquina o qual faz a soma do ganho em horas extras no salário base e depois armazena o resultado no salário bruto:

```
+1300042774
+1400593419
+1200274027
```

Não dá pra entender, não é? O trecho é apenas uma parte de um programa que depois é convertido em bits para a execução pelo computador. Imagine um programa maior como seria! Ainda, imagine que este programa deve rodar em outro computador diferente, ele teria que ser totalmente modificado pois os códigos de máquina seriam diferentes! Quer dizer, é improdutivo demais! Precisamos então de uma linguagem mais próxima ao nosso entendimento.

Surgiu então a ***linguagem assembly***. A linguagem *assembly* é composta por símbolos abreviados, chamados de mnemônicos, os quais eram usados para compor a programação. Veja o código a seguir:

```
load   salbase
add    horaextra
store  salbruto
```

O entendimento do código fica mais fácil. Mas ainda assim não é o ideal pois em programas muito grandes, você pode imaginar a quantidade de linhas que devem ser escritas e no caso de erros na programação, o tempo que é perdido tentando encontrar onde está o problema. E ainda tem um detalhe: é necessário um **programa tradutor** o qual lê o código *assembly* e o transforma na linguagem de máquina do computador no qual está executando o programa. Logo, para cada computador, há uma linguagem *assembly* diferente.

O melhor seria termos uma linguagem de programação que se aproxima da forma na qual conversamos e escrevemos. Melhor ainda se ela entendesse a nossa fala e já programasse o computador (isso é chamado de linguagem natural). Desta necessidade, surgiram as **linguagens de alto nível**. Elas apareceram devido à grande popularização dos computadores e devido à grande necessidade de executar tarefas simples com instruções mais simples. As linguagens Java, C, C++, C#, PHP, Visual Basic, Python e tantas outras que conhecemos atuais são linguagens de alto nível.

Assim como o *assembly*, é preciso que um programa tradutor leia o código fonte (escrito em linguagem de alto nível) e o converta para a linguagem de máquina. Este programa é chamado de compilador e cada linguagem de programação possui o seu próprio **compilador**. As linguagens de alto nível permitem que os programadores escrevam programas com palavras que entendemos. Por exemplo:

```
salarioBruto = salarioBase + horasExtras;
```

Bem mais fácil, não é? Porém o processo de compilação de um programa em linguagem de alto nível pode demorar muito e consumir muito recurso da máquina. Para isso, foram desenvolvidos os programas **interpretadores** os quais executam as linguagens de alto nível diretamente, porém mais lentamente. Os interpretadores são muito usados em ambientes de desenvolvimento (IDE – *Integrated Development Environment*) para poder corrigir eventuais erros a medida que o programa é executado. Posteriormente uma versão compilada é produzida.

1.2 Introdução à linguagem Java

A linguagem Java começou em 1991 com um projeto mantido pela Sun Microsystems chamado *Green*. Este projeto tinha como objetivo integrar vários dispositivos eletrônicos, entre eles os computadores, por meio de uma mesma linguagem de programação. Uma vez que os dispositivos eletrônicos são compostos por microprocessadores, a ideia era desenvolver uma linguagem na qual um mesmo programa pudesse ser executado em diferentes dispositivos.

Com o passar do tempo, o projeto recebeu outro nome, *Oak* (carvalho, em inglês) e depois Java. O mercado de dispositivos eletrônicos não evoluiu como a Sun esperava e o projeto correu o risco de ser abortado. Porém, com o advento da *internet*, e a necessidade de gerar conteúdo dinâmico para as páginas *web*, o projeto tomou outro rumo e o Java foi lançado oficialmente em 1995 e chamou a atenção pelas várias facilidades que possuía para desenvolver aplicativos para a *internet*. Além disso, o Java possibilitava desenvolver também para aplicações em computadores desktop bem como outros dispositivos como os pagers e celulares.

Usos e aplicações do Java: A plataforma Java é composta por várias tecnologias. Cada uma delas trata de uma parte diferente de todo o ambiente de desenvolvimento e execução de software. A interação com os usuários é feita pela máquina virtual Java (Java Virtual Machine, ou JVM) e um conjunto padrão de bibliotecas de classe. Uma aplicação Java pode ser usada de várias maneiras: *applets* que são aplicações embutidas em páginas web, aplicativos para *desktops*, aplicativos para aparelhos celulares e em servidores de aplicações para internet.

1.3 Funcionamento de um programa em Java

Basicamente um programa em Java para ser executado passa por 5 fases:

EDIÇÃO:	o programa é feito em um editor de texto e armazenado no disco com a extensão *.java*

COMPILAÇÃO:	por meio do programa *javac.exe* (no Windows), o arquivo .java é compilado em *bytecodes* e armazenado no disco com a extensão *.class*
CARREGAMENTO:	o carregador de classes lê o arquivo *.class* com os *bytecodes* e aloca-os na memória
VERIFICAÇÃO:	outro programa faz a verificação dos *bytecodes* para garantir a validação dos códigos gerados e a segurança do Java
EXECUÇÃO:	A execução do programa se dá pela JVM por meio da tradução dos *bytecodes* na linguagem de máquina do computador onde está sendo executado. Isto é feito pelo programa *java.exe* (no Windows).

1.4 Ambiente de programação e tipos de programas

Basicamente, é possível desenvolver em Java usando apenas dois componentes principais: um editor de texto comum e o *Java Development Kit* (JDK – Kit de desenvolvimento em Java). Existem várias versões do compilador para desenvolvimento em Java, a mais básica e comum é a *Java Standard Edition*, ou Java SE. Esta é a versão que usaremos neste material.

Porém existem ambientes de desenvolvimento para Java. Entre eles não podemos deixar de mencionar o *Netbeans* e o Eclipse, ambos muito usados pela comunidade Java por possuírem código aberto e serem gratuitos, são produtos livres, sem restrições à sua forma de utilização. Vamos usar o *Netbeans* como exemplo.

O *Netbeans* é usado tanto para desenvolvimento de aplicações simples até grandes aplicações empresariais e assim como o Eclipse, suporta outras linguagens de programação como C, C++, PHP, etc. O *Netbeans* é um ambiente de desenvolvimento, ou seja, uma ferramenta para programadores o qual permite escrever, compilar, depurar e instalar programas. O IDE é completamente escrito em Java, mas pode suportar qualquer linguagem de programação. Existe também um grande número de módulos para extender as funcionalidades do *NetBeans*.

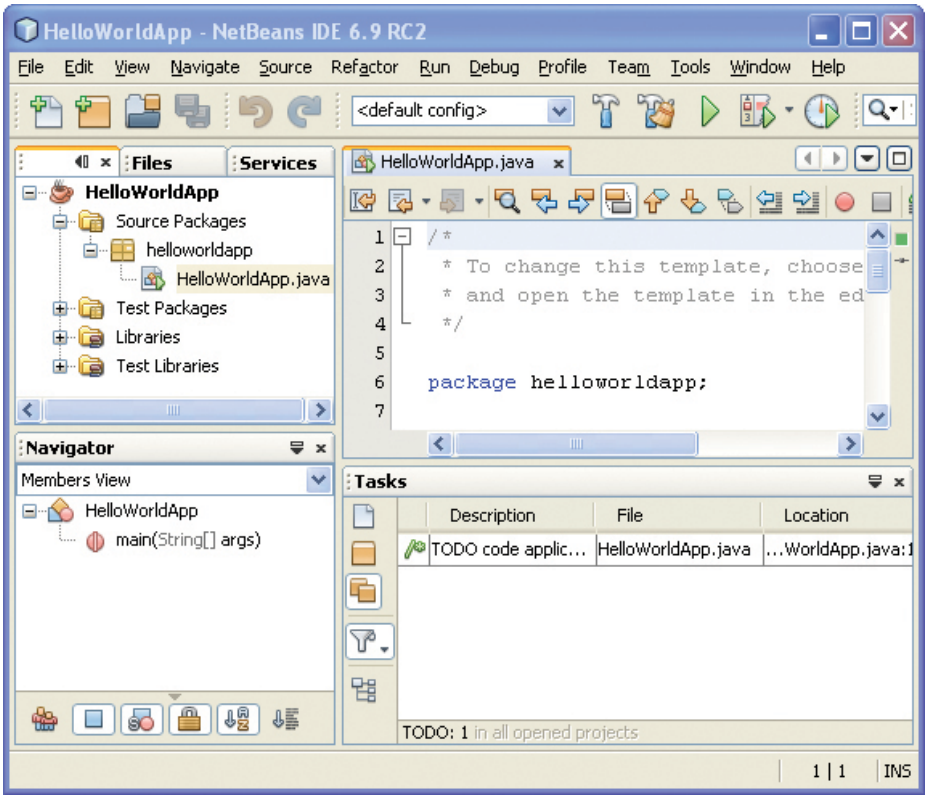


Figura 1.1 – Tela típica do Netbeans. Fonte: www.netbeans.org.

Com o Netbeans podemos desenvolver vários tipos de programas em Java como por exemplo:

- Aplicativos em Java para console;
- Java para *desktop* com os pacotes *Swing e JavaFX;*

• Java para *web*, com o uso de vários *frameworks* como o *Struts, Java Server Faces* e outros, integrado a servidores de aplicações como o *JBoss, Tomcat* e bancos de dados;

• Aplicativos para dispositivos móveis;

• *Applets;*

• Aplicativos para *web* em geral com HTML5;

• E outros.

1.5 Escrevendo e executando o primeiro programa

Agora vamos precisar que você instale o *Netbeans* e o JDK. No site do *Netbeans* existe uma versão do programa que inclui o JDK (chamada *bundled version*). Se você não possui o JDK instalado na sua máquina é interessante que baixe esta versão integrada. No próprio site do *Netbeans* tem instruções de instalação detalhadas de acordo com o sistema operacional.

Nosso primeiro programa será bem simples e para usar o *Netbeans* pela primeira vez é necessário que você entenda alguns termos que são usados no programa a fim de melhor se acostumar com o *software*. Para isso recomendamos a leitura e execução das atividades do seguinte link:

https://netbeans.org/kb/docs/java/quickstart_pt_BR.html

Neste link você encontrará informações a respeito de como criar um projeto e executar o primeiro programa em Java. Conforme avançarmos pelas aulas, você terá condições de entender o programa que você acabou de digitar. Vamos aprender sobre a linguagem Java?

1.6 Tipos de dados primitivos

Quando vamos escrever um programa de computador, em qualquer linguagem, vamos ter que usar variáveis. As variáveis servem para guardar valores dos mais variados tipos: podem guardar números inteiros ou decimais, valores alfanuméricos, somente alfabéticos, valores lógicos (como verdadeiro ou falso) e muitos outros. Existem linguagens que permitem que o programador crie seus próprios tipos.

Vamos usar um algoritmo bem simples para ilustrar:

```
algoritmo Soma
numero1, numero2, soma: inteiro
inicio
    soma ← 0
    escreval("Digite o primeiro número inteiro:")
    leia(numero1)
    escreval("Digite o segundo número inteiro:")
    leia(numero2)
    soma ← numero1 + numero2
    escreval("A soma é",soma)
fim
```

Algoritmo 1.1 – Programa que soma dois números inteiros.

Como podemos ver no algoritmo 1.1, é necessário criar 3 variáveis para guardar os dados que vamos ler do usuário e para armazenar o valor da soma dos dados lidos. Estas variáveis são declaradas antes de serem usadas e o seu tipo não é alterado durante a execução do programa. Em algumas linguagens é **obrigatório** declarar o tipo e a variável **antes** de serem usadas pelo programa. Em outras linguagens isso não é obrigatório.

Linguagem fortemente tipada: é aquela que a declaração do tipo da variável é obrigatória. Exemplo: Java, C, C++, ect..
Linguagem fracamente tipada: é aquela que pode alterar o tipo durante a execução do programa. Exemplo: PHP, Python, Ruby, Javascript.

Linguagens não tipada: é aquela em que só existe um tipo genérico para todo o programa ou nem existe. Exemplo: Perl

Linguagem de tipo estático: neste tipo de linguagem, o compilador deve conhecer o tipo antes da execução do programa. Exemplo: Java, C, C++, etc.

Linguagem de tipo dinâmico: o tipo da variável é conhecido somente na execução do programa. Exemplo: PHP, Ruby, Python

Como podemos ver no Box Explicativo, a linguagem Java é fortemente tipada e possui tipos estáticos, ou seja, antes de usar qualquer variável será obrigatório declarar a variável e seu tipo.

A linguagem Java, assim como outras linguagens, possui um conjunto de tipos necessários para as construções básicas da linguagem. Este conjunto é chamado de **tipos primitivos** e na prática são implementados por palavras-chave. Cada tipo primitivo possui um tamanho de memória (em bits) que é usado para armazenar o seu valor. Além disso, ele possui uma escala de valores, ou seja, possui um conjunto de valores específicos. Veja a tabela 1.1 para conhecer os tipos primitivos da linguagem Java e demais informações.

TIPO	TAMANHO (BITS)	ESCALA
boolean	-	true ou false
char	16	'\u0000' a '\uFFFF' (0 a 65535 – caracteres Unicode ISO)
byte	8	-128 a +127
short	16	-32768 a 32767
int	32	-2.147.483.648 e 2.147.483.647

TIPO	TAMANHO (BITS)	ESCALA
long	64	9223372036854775808 a 9223372036854775807
float	32	1,40129846432481707e-45 a 3,40282346638528860e+38
double	64	4,94065645841246544e-324d a 1,79769313486231570e+308d

Tabela 1.1 – tipos primitivos da linguagem Java.

Vamos explicar um pouco sobre a aplicação destes tipos:

• **boolean:** O *boolean* é fácil, ele só pode armazenar dois valores: true ou false. Java só aceita estes dois valores. Outras linguagens aceitam 0 ou 1, V ou F, mas Java não.

• **char:** este é tranquilo também. O *char* guarda qualquer caractere Unicode [1].

• **byte:** o *byte* aceita números compreendidos entre -127 e 128. É possível usar um modificador para usarmos somente números positivos também.

• **short:** o propósito do *short* é o mesmo do *byte,* porém ele guarda o dobro de valores do *byte*.

• **int:** este tipo guarda somente valores inteiros compreendidos no intervalo mostrado na tabela 1.1.

• **float e double:** estes dois tipos fazem a mesma coisa: guardam números decimais (também chamados de ponto flutuante). A diferença é que o double, naturalmente, guarda valores maiores que o *float*. Eles são usados em programas que requerem grande precisão de contas e resultados.

1 O Unicode é um padrão para representar caracteres usando números inteiros de 16 bits. O caractere "A" é representado como '/u0041' e o caractere "a" representado por '/u0061'

Quando declaramos uma variável e não atribuímos nenhum valor a ela, o Java atribui um valor automaticamente a ela de acordo com alguns padrões conforme mostra a tabela 1.2.

TIPO	VALOR PADRÃO
BYTE	0
SHORT	0
INT	0
LONG	0L
FLOAT	0.0f
DOUBLE	0.0d
CHAR	'\u0000'
BOOLEAN	false

Tabela 1.2 – Valores padrões para os tipos em Java.

1.6.1 Literais

Um literal é uma representação de algum valor fixo que o programador literalmente estabelece. Cada tipo primitivo possui um literal correspondente e na tabela 1.2 estão representados os literais padrões para cada tipo. Observe os exemplos a seguir. São todos exemplos de literais.

```
boolean resultado = true;
char cMaiusculo = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

E se pegássemos como exemplo o literal 100? Ele é um *int* ou um *long*? Você percebeu na tabela 1.2 que os valores *long, float e double* tem letras no final do valor? Estas letras servem para distinguir o tipo do literal em algumas situações.

Portanto, se 100 é um *long*, podemos representar como 100L ou 100l (é preferível usar o "L" maiúsculo). Se o 100 não tiver uma letra no final, será considerado um *int*.

Os valores literais para *byte, short, int e long* podem ser criados a partir de um literal *int*. Valores do tipo *long* que excedem a capacidade de um *int* podem ser criados por meio de literais do tipo *long*. Os literais do tipo *int* podem ser decimais, binários ou hexadecimais. Esses dois últimos sistemas de numeração são usados em programas que lidam e manipulam endereços de memória por exemplo. Veja os exemplos a seguir:

```
// O número 26 em decimal
int decimal = 26;
// Em hexadecimal
int hexa = 0x1a;
// E em binário
int binario = 0b11010;
```

1.6.2 Literais de ponto flutuante

Um literal de ponto flutuante é do tipo *float* se terminar com a letra "F" ou "f", caso contrário, seu tipo será *double* e pode terminar com "D" ou "d" opcionalmente.

Os tipos de ponto flutuante (*float e double*) também podem ser expressos na notação científica usando a letra "E" ou "e", "F" ou "f" para um literal *float* e "D" ou "d" para um literal *double*. Observe os exemplos:

```
double d1 = 123.4;
// é o mesmo valor que d1 mas em notação científica
double d2 = 1.234e2;
float f1  = 123.4f;
```

1.6.3 Literais caracteres e Strings

Os literais dos tipos char e String podem conter qualquer caractere Unicode. Dependendo do editor de texto que você estiver usando para programar você pode digitar diretamente caracteres como "Ç", "ã" e outros ou usar uma sequência chamada "escape para Unicode" para poder gera-los. Esta sequência são os caracteres \u. Veja os exemplos:

```
//literal char
char c = 'a';  //observe que o literal char tem que estar entre aspas simples!

char n = '\u004E';   //corresponde à letra E
```

As strings em Java na verdade são instâncias de uma classe Java chamada *String*. Existem várias maneiras de se representar uma *string* em Java e todas elas usam aspas duplas para a string desejada.

Existe também o literal nulo, representado pela palavra-chave *null*. O literal nulo representa ausência de valor e dado. O literal nulo pode ser atribuído à qualquer variável exceto variáveis de tipos primitivos.

```
//exemplo de literal String
//observe que a classe String começa com S maiúsculo!

String s = “Exemplo de uma string”;
```

1.7 Estruturas de decisão e repetição

As estruturas de decisão e repetição, assim como as estruturas sequenciais são chamadas de estruturas de controle em uma linguagem de programação. Em Java só existem estes três tipos de estruturas de controle.

As estruturas de decisão também são chamadas de estruturas de seleção e possuem três tipos e as estruturas de repetição possuem três tipos em Java.

Um programa de computador é formado pela combinação das instruções de sequência, de repetição e de seleção conforme descrito no algoritmo que o programa está implementando.

1.7.1 Estruturas de decisão

1.7.1.1 IF-THEN

Para poder mostrar o funcionamento dos comandos em Java vamos usar o diagrama de atividades da UML. Desta forma você pode entender visualmente como o comando é executado e compará-lo com o código em Java correspondente.

A figura 1.2 mostra o diagrama de atividade de uma instrução de seleção básica e o código correspondente em Java.

Por enquanto vamos adotar que todas as saídas para a tela dos nossos programas serão exibidas no console de comandos do sistema operacional. O comando para gerar uma saída no console em Java é:

```
System.out.println(String s);
//o println mostra a string s e coloca o cursor na próxima linha

System.out.print(String s);
//o print mostra a string s e deixa o cursor na mesma linha
```

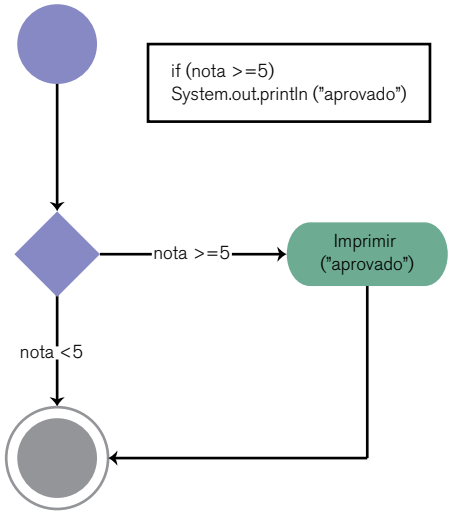


Figura 1.2 – O comando IF

Observe que neste caso temos apenas uma instrução que é executada se a condição do *if* for verdadeira. É claro que é possível ter várias instruções sendo executadas porém neste caso teremos que abrir um bloco, como no exemplo a seguir onde temos 2 instruções dentro de um bloco:

```
if (nota>=5) {       // início do bloco. Use o { para abrir
   System.out.println("Aprovado!");
   System.out.println("Você pode agora imprimir o seu boletim.");
}                    // fim do bloco. Use o } para fechar o bloco
```

Outra observação a ser feita é que obrigatoriamente a condição do if tem que estar entre parênteses.

1.7.1.2 IF-THEN-ELSE

O IF-THEN-ELSE em Java pode ser representado como mostra a figura 1.3.

Novamente como pode ser observado na figura 1.3 só foi colocada uma

instrução após a condição verdadeira e uma após a condição falsa. E assim como ocorre no IF-THEN, é possível ter um bloco após cada condição.

Uma boa prática de programação que você já deve ter aprendido anteriormente é a identação do código. Identar significa deslocar o código interno de um bloco para melhor legibilidade do código.

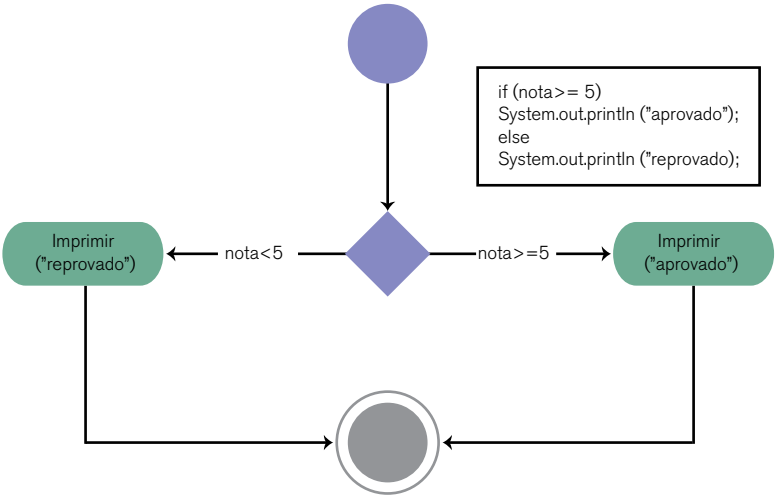


Figura 1.3 – O IF-THEN-ELSE em Java

No Java existe um operador condicional (?:) que pode ser usado no lugar do IF-THEN-ELSE. Esse na verdade é o único operador ternário do Java, ou seja, ele recebe três operandos juntos.

```
    System.out.println(nota >=5 ? “Aprovado” : “Reprovado”);

    //podemos ler a instrução assim: “Nota é maior ou igual a 5? Caso verdadeiro,
// imprima “Aprovado”, em caso contrário (após o :), imprima “Reprovado”)
```

Outra possibilidade é aninhar várias instruções IF-THEN ou IF-THEN-ELSE, observe com atenção o código a seguir. O que ocorre se a nota for igual a 5?

Observe que foi criado um bloco para o último else (apenas como exemplo).

```
if (nota>=9)
   System.out.println(“Conceito A”);
else
   if (nota>=8)
      System.out.println(“Conceito B”);
   else
      if (nota>=7)
         System.out.println(“Conceito C”);
      else
         if (nota>=6)
            System.out.println(“Conceito D”);
         else {
            System.out.println(“Conceito E”);
            System.out.println(“Você está reprovado!”);
         }
```

1.7.2 SWITCH

Podemos perceber que o IF-THEN é uma estrutura do tipo seleção única, ou seja, se a condição existente for verdadeira, um bloco de código é executado e a estrutura é finalizada.

No IF-THEN-ELSE a estrutura possui dupla seleção: se a condição for verdadeira, um bloco é executado ou senão o bloco correspondente à condição falsa será executado. Porém o Java possui uma estrutura na qual uma instrução de seleção múltipla pode realizar diferentes ações baseadas nos possíveis valores de uma variável ou expressão. A instrução switch possui sua estrutura geral mostrada na figura 1.4.

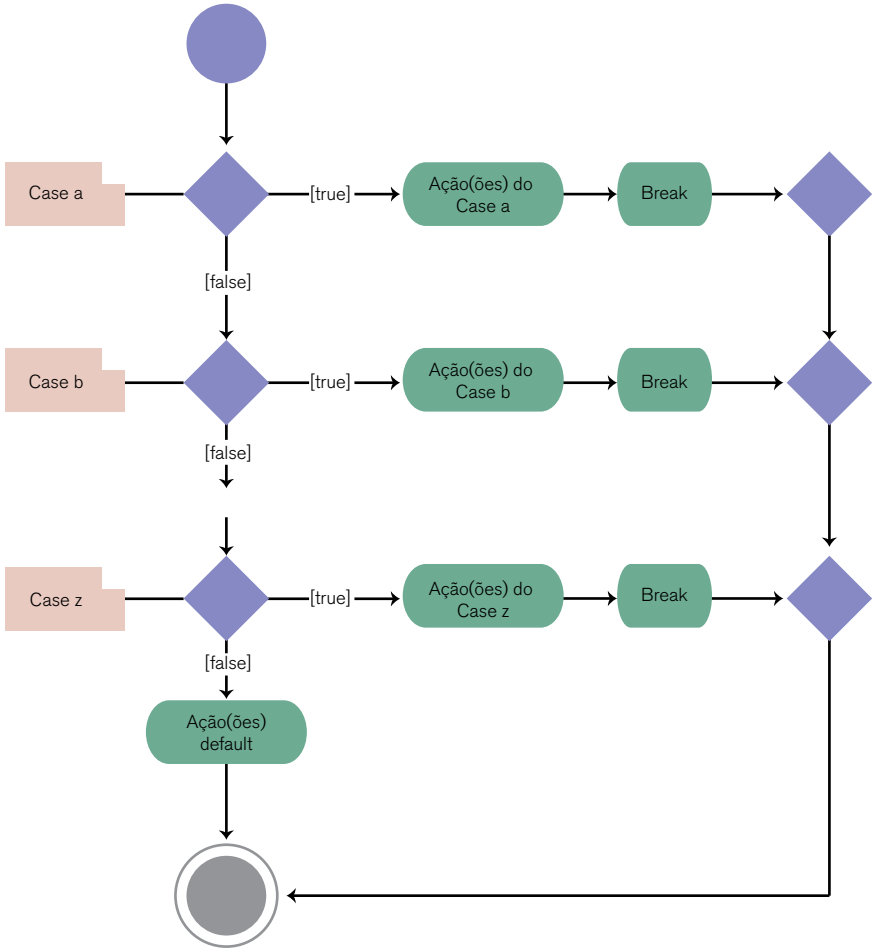


Figura 1.4 – Instrução de seleção múltipla (SWITCH) com instruções break.

O *switch* pode ser usado com os tipos *byte, short, char e int*, além de tipos enumerados, com a classe *String* e algumas classes especiais que encapsulam alguns tipos primitivos como *Character, Byte, Short e Integer* (veremos essas classes ao longo deste livro).

Para ilustrar esse comando, vamos usar um exemplo. No exemplo é declarada uma variável inteira que representa o valor do mês. O programa mostra o nome do mês de acordo com o valor numérico associado.

```
1  public class Selecao {
2      public static void main(String[] args) {
3
4          int mes = 8;
5          String nomeMes;
6          switch (mes) {       //inicio do bloco
7              case 1:nomeMes = "Janeiro";
8                          break;     //perceba o comando break no
                                      //final de cada condição
9              case 2:  nomeMes = "Fevereiro";
10                         break;
11             case 3:  nomeMes = "Março";
12                         break;
13             case 4:  nomeMes = "Abril";
14                         break;
15             case 5:  nomeMes = "Maio";
16                         break;
17             case 6:  nomeMes = "Junho";
18                         break;
19             case 7:  nomeMes = "Julho";
20                         break;
21             case 8:  nomeMes = "Agosto";
22                         break;
23             case 9:  nomeMes = "Setembro";
24                         break;
25             case 10: nomeMes = "Outubro";
26                         break;
27             case 11: nomeMes = "Novembro";
28                         break;
29             case 12: nomeMes = "Dezembro";
30                         break;
31             default: nomeMes = "Mês inválido";
32                         break;
33         }                        //fim do bloco
34     System.out.println(nomeMes);
35     }
36 }
```

Neste caso, o mês "Agosto" será impresso na tela.

O corpo de um comando switch é chamado de *bloco switch*. Uma declaração dentro do bloco switch pode ser rotulada com um ou mais comandos case (ou default). O comando switch avalia sua expressão e depois executa todas as declarações que "batem" com o rótulo case correspondente. No exemplo é mostrado na linha 21 e 22.

O mesmo código mostrado como exemplo pode ser convertido em vários comandos IF-THEN-ELSE:

```
int mes = 8;
if (mes == 1) {
   System.out.println("Janeiro");
}
37 else if (mes == 2) {
   System.out.println("Fevereiro");
}
...  // e assim por diante
```

A decisão em usar declarações IF-THEN-ELSE ou uma instrução switch é baseada na legibilidade do código e na expressão que a declaração está testando. Uma declaração IF-THEN-ELSE pode testar expressões com base em faixas de valores ou condições enquanto que um *switch* testa expressões baseadas somente em um inteiro, valor enumerado, ou *String*.

Outro ponto de observação é a instrução *break*. Cada instrução break termina a declaração de fechamento do *switch*. O fluxo de controle continua com a primeira declaração após o bloco switch. O *break* é necessário porque sem eles as declarações nos blocos ***switch*** seriam executadas em sequência e indistintamente até que algum *break* fosse encontrado.

Observe atentamente o código a seguir:

```
public class SwitchDemoFallThrough {
   public static void main(String[] args) {
      java.util.ArrayList<String > meses = new java.util.ArrayList<String>();
      int mes = 8;
```

```
        switch (mes) {
            case 1:   meses.add("Janeiro");
            case 2:   meses.add("Fevereiro");
            case 3:   meses.add("Março");
            case 4:   meses.add("Abril");
            case 5:   meses.add("Maio");
            case 6:   meses.add("Junho");
            case 7:   meses.add("Julho");
            case 8:   meses.add("Agosto");
            case 9:   meses.add("Setembro");
            case 10:  meses.add("Outubro");
            case 11:  meses.add("Novembro");
            case 12:  meses.add("Dezembro");
                      break;
            default:  break;
        }

        if (meses.isEmpty()) {
            System.out.println("Número de mês inválido");
        }
        else {
            for (String nomeMes : meses) {
                System.out.println(nomeMes);
            }
        }
    }
}
```

Embora existam comandos que nós não vimos ainda, dá para perceber que temos vários *case* sem o comando break para finalizar cada bloco. A saída na tela desse programa será:

```
Agosto
Setembro
Outubro
Novembro
Dezembro
```

Veja no exemplo que o programa busca o case que “bate” com o valor da variável mês e encontra no “case 8”. O programa adiciona a palavra “Agosto” na lista de meses e assim prossegue com o “case 9”, “case 10” até o “case 12” pois não há um comando break para parar a execução.

Assim, tecnicamente o break final acaba sendo desnecessário. Usar o *break* é recomendado para a legibilidade e deixar o código menos propenso a erros. O *default* trata todos os valores que não foram listados anteriormente nos case.

1.8 Estruturas de repetição

As estruturas de repetição também são chamadas de laços ou loops e permitem que o programador especifique que um programa deve repetir uma ação enquanto uma condição for verdadeira.

1.8.1 WHILE

O comando *while* executa repetidamente um bloco enquanto uma condição particular for verdadeira. A sintaxe do comando e o seu diagrama de atividade está representado na figura 1.5.

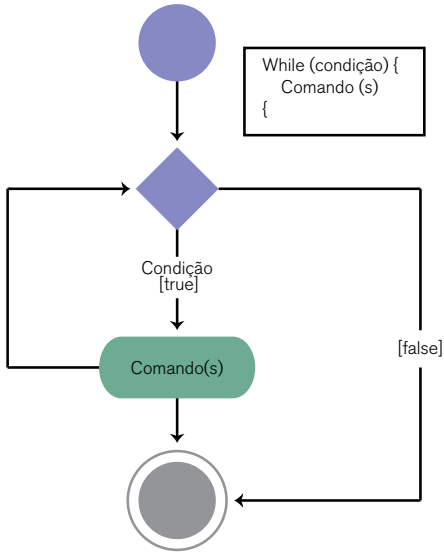


Figura 1.5 – O comando WHILE.

O comando *while* avalia a condição, a qual sempre retorna um valor *boolean (true ou false).* Se a condição retorna *true, o while* executa o(s) comando(s) no bloco. O *while* continua testando a condição e executando o bloco até que a condição seja *false*.

```
class Repete {
   public static void main(String[] args){
       int conta = 1;
       while (conta < 11) {
           System.out.println("Contando: " + conta);
           conta = conta+1;
       }
   }
}
```

A linguagem Java também possui um comando DO-WHILE, o qual possui a seguinte sintaxe:

```
do {
          comando(s)
} while (condição);
```

A diferença entre o DO-WHILE e WHILE é que o DO-WHILE analisa a expressão no final da repetição ao invés do início, ou seja, pelo menos uma vez o bloco de comandos será executado.

```
class Repete2 {
   public static void main(String[] args){
       int conta = 1;
       do {
           System.out.println("Contando: " + conta);
           conta = conta+1;
       } while (conta < 11);
   }
}
```

1.8.2 FOR

O comando for é um meio compacto de fazer uma repetição sobre um intervalo de valores. Na programação é comum usar o termo "*loop*" em repetições devido ao modo no qual o for, neste caso, repetidamente executa enquanto uma condição é verdadeira. A figura 1.6 mostra o diagrama de atividade e a sintaxe.

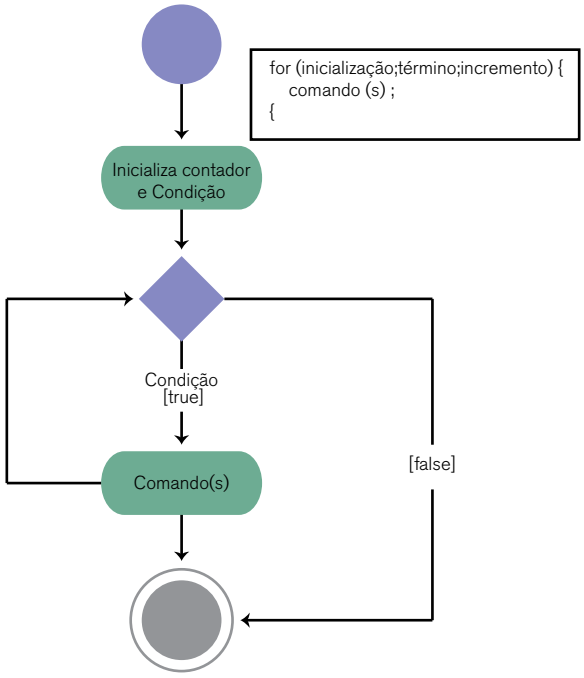


Figura 1.6 – O comando FOR.

Algumas observações devem ser feitas para o comando for:

• A expressão de inicialização começa o loop. É executada uma única vez assim que o loop começa.

• Quando a expressão de término é analisada e é falsa, o loop finaliza.

• A expressão de incremento é chamada a cada iteração dentro do loop. Normalmente é feito um incremento ou um decremento.

```
class Repete3 {
   public static void main(String[] args){
       for(int i=1; i<11; i++){
           System.out.println("Contando: " + i);
       }
   }
}
```

A saída do programa é:

```
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
Contando: 6
Contando: 7
Contando: 8
Contando: 9
Contando: 10
```

Observe como o código declara a variável dentro da expressão de inicialização. O escopo desta variável se estende desde sua declaração até o fim do bloco do for, de modo que pode ser usada nas expressões de terminação e de incremento. Se a variável que controla um for não é necessária fora do *loop*, é melhor declarar a variável na expressão de inicialização. Frequentemente e tradicionalmente usamos variáveis com nomes i, j e k para controlar os *loops for.* Declarando-as na inicialização faz com que sua vida útil seja limitada, assim como seu escopo e isso reduz erros. Veja o uso da variável i como exemplo no código anterior.

O *for* tem uma função estendida para vetores e classes *Collections*. Este tipo de *for* é chamado de *enhanced* for e pode ser usado para fazer iterações dentro dessas estruturas de dados, percorrendo-as. Por exemplo, considere o programa a seguir (vamos tratar sobre vetores em Java mais à frente):

```
class EnhancedFor {
   public static void main(String[] args){
      //declarando um vetor inteiro contendo uma lista de números
      int[] numeros = {1,2,3,4,5,6,7,8,9,10};
      //o for abaixo percorre o vetor e mostra na tela cada um dos números
      for (int item : numeros) {
         System.out.println("Contando: " + item);
      }
   }
}
```

No exemplo, a variável *item* guarda o valor atual dos números do vetor. A saída do programa está mostrada abaixo:

```
Contando: 1
Contando: 2
Contando: 3
Contando: 4
Contando: 5
Contando: 6
Contando: 7
Contando: 8
Contando: 9
Contando: 10
```

Esta forma de usar o for deve ser usada sempre que possível, ok?

1.9 Strings

As *strings* são muito usadas na linguagem Java e são na verdade sequências de caracteres. Na linguagem Java, as *strings* são objetos. A plataforma Java fornece uma classe chamada *String* (perceba a letra maiúscula inicial) para criar e manipular *strings*.

1.9.1 Criação de strings

A forma mais direta para criar uma string é:

```
String alo = “Alô mundo!”;
```

Neste caso, “Alô mundo!” é um literal *string*, ou seja, uma série de caracteres que são limitados por aspas duplas. Toda vez quem um literal *string* é encontrado no programa, o compilador cria um objeto *String* com seu valor, neste caso Alô Mundo!.

Como veremos mais adianta, assim como qualquer outro objeto, você pode criar objetos *String* usando a palavra chave *new* e um construtor. A classe *String* possui vários construtores que permitem você definir o valor inicial de uma *string* de diversas formas, como por exemplo:

```
char[] vetor = { 'a', 'l', 'o', '.' };
String aloString = new String(vetor);
System.out.println(aloString);
```

1.9.2 Tamanho da string

Como as *strings* em Java são objetos então existem métodos para obter informações destes objetos e são chamados de métodos assessores. Um dos métodos assessores para as *strings* é o *length()*. Este método retorna o número de caracteres de uma *string*. Veja o exemplo:

```
String palindromo = "A sacada da casa";
int tamanho = palindromo.length();
```

Um palíndromo é uma palavra ou frase que mantém o mesmo sentido quando lida de trás para a frente. Leia "A sacada da casa" de trás para frente: você perceberá que é a mesma frase!

No exemplo anterior, a variável tamanho, após a execução do método, conterá o valor 16, que é o tamanho da *string* palíndromo. Não esqueça que o espaço conta como um caractere normal!

Veja o programa a seguir:

```
public class TesteString {
    public static void main(String[] args) {
        String palindromo = "A sacada da casa";
        int tam = palindromo.length();
        char[] vetorCaracTemp = new char[tam];
        char[] vetorChar = new char[tam];

        // guardando a string original em um vetor de caracteres
        for (int i = 0; i < tam; i++) {
            vetorCaracTemp[i] = palindromo.charAt(i);
        }

        // invertendo o vetor de caracteres
        for (int j = 0; j < tam; j++) {
            vetorChar[j] = vetorCaracTemp[tam - 1 - j];
        }

        String palindromoInvertido = new String(vetorChar);
        System.out.println(palindromoInvertido);
    }
}
```

O programa não é muito útil pois ele inverte um palíndromo! Porém ele serve para exemplificar o uso de alguns métodos assessores das *strings (lenght() e o charAt())* e relembrar alguns tópicos vistos sobre repetição com o *for* (observe o uso das variáveis i e j).

Inicialmente o programa precisa converter a *string* em um vetor de caracteres (no primeiro *loop*), inverter o vetor em um segundo vetor (segundo *loop*) e depois converter novamente em uma *string*. A classe *String* contém um método

chamado *getChars()* que converte uma *string* (ou parte dela) em um vetor de caracteres e assim poderíamos trocar o primeiro loop do programa para a seguinte linha:

```
palindromo.getChars(0, tam, vetorCaracTemp, 0);
```

Foi usado outro método no programa chamado *charAt(x)*. Este método retorna o caractere encontrado na posição x do vetor que estamos usando para fazer a pesquisa.

1.9.3 Concatenando strings

A classe *String* contém um método para concatenar duas *strings*:

Isto retornará uma nova *string* começando com *string1 e string2* no final.

Também é possível concatenar strings de acordo com os exemplos a seguir, observe com atenção:

```
//com literias
"Meu nome é ".concat("Bond. James Bond");

//usando o operador +, que é a forma mais comum
String teste = "Olá " + " mundo" + "!"

//o que resultará em: Olá mundo !

String string1 = “sacada da";
System.out.println("A " + string1 + " casa");

//o que resultará em A sacada da casa
```

A concatenação pode ser uma mistura de qualquer objeto. Para cada objeto que não for uma *string*, é necessário usar o método *toString()* que converte o objeto em uma *string*.

CONEXÃO

Existem mais métodos muito úteis na classe String. Porém um bom desenvolvedor precisa conhecer a API da linguagem para poder usar os métodos eficientemente. A API da linguagem Java, especialmente da classe String pode ser encontrada no seguinte link: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

1.10 Vetores

Um vetor (ou *array*) é uma estrutura de dados que armazena uma sequência de dados, ou objetos, todos do mesmo tipo em posições consecutivas da memória.

O tamanho de um vetor é fixado assim que o vetor é criado.

Cada item em um vetor é chamado de elemento e cada elemento é acessado por meio de um valor chamado índice. Em Java, os índices dos vetores começam com 0 (zero).

Vamos ver um exemplo no qual é criado um vetor de inteiros, alguns valores são adicionados e depois impressos na tela.

```
class Array1 {
    public static void main(String[] args) {
        // declarando um vetor de inteiros
        int[] umVetor;

        // definindo o tamanho do vetor para 10 inteiros
        umVetor = new int[10];

        // definindo o primeiro elemento
        umVetor[0] = 100;
        // definindo o segundo elemento
        umVetor[1] = 200;
        // e assim por diante
        umVetor[2] = 300;
        umVetor[3] = 400;
        umVetor[4] = 500;
```

```
        umVetor[5] = 600;
        umVetor[6] = 700;
        umVetor[7] = 800;
        umVetor[8] = 900;
        umVetor[9] = 1000;

        System.out.println("Elemento com índice 0: " + umVetor[0]);
        System.out.println("Elemento com índice 1: " + umVetor[1]);
        System.out.println("Elemento com índice 2: " + umVetor[2]);
        System.out.println("Elemento com índice 3: " + umVetor[3]);
        System.out.println("Elemento com índice 4: " + umVetor[4]);
        System.out.println("Elemento com índice 5: " + umVetor[5]);
        System.out.println("Elemento com índice 6: " + umVetor6]);
        System.out.println("Elemento com índice 7: " + umVetor[7]);
        System.out.println("Elemento com índice 8: " + umVetor[8]);
        System.out.println("Elemento com índice 9: " + umVetor[9]);
    }
}
```

```
A saída do programa será:

Elemento com índice 0: 100
Elemento com índice 1: 200
Elemento com índice 2: 300
Elemento com índice 3: 400
Elemento com índice 4: 500
Elemento com índice 5: 600
Elemento com índice 6: 700
Elemento com índice 7: 800
Elemento com índice 8: 900
Elemento com índice 9: 1000
```

Em uma situação real você provavelmente usará as estruturas de repetição que já vimos neste capítulo para percorrer o vetor pois ficará muito mais legível e elegante do que escrever linha a linha como foi feito no exemplo.

1.10.1 Declarando uma variável como vetor

No programa anterior criamos um vetor de inteiros (chamado umVetor) com 10 posições.

Assim como nas declarações para variáveis, uma declaração de vetor tem dois componentes: o nome do tipo e o nome do vetor. O tipo do vetor é escrito como tipo[], onde o tipo é o tipo dos dados do conteúdo do vetor. Os colchetes indicam que o tipo declarado conterá um vetor. O tamanho do vetor não é obrigatório na declaração. Lembre-se declarar um vetor não significa que ele estará criado. A declaração conta ao compilador que a variável declarada conterá um vetor de um determinado tipo. A criação se dá com o comando *new*.

Você pode declarar vetores de outros tipos:

```
byte[] vetorBytes;
short[] vetorShorts;
long[] vetorLongs;
float[] vetorFloats;
double[] vetorDoubles;
boolean[] vetorBooleanos;
char[] vetorChars;
String[] vetorStrings;
```

1.10.2 Criando, acessando e manipulando um vetor

Uma forma de criar um vetor é usar o operador *new* como já falamos. A linha do programa *Array1*.java anterior cria um vetor com 10 posições.

```
// declarando um vetor de inteiros
int[] umVetor;
// definindo o tamanho do vetor para 10 inteiros
umVetor = new int[10];

//também poderia ser feito as seguinte forma:
int[] umVetor = new int[10];
```

Se o *new* for esquecido ou omitido, o compilador gerará um erro dizendo que a variável umVetor não foi inicializada.

As linhas abaixo atribuem um valor a cada posição do vetor:

```
umVetor[0] = 100; // inicializa o primeiro elemento
umVetor[1] = 200; // inicializa o segundo elemento
umVetor[2] = 300; // e assim por diante
```

Cada elemento do vetor é acessado pelo seu índice numérico:

```
System.out.println("Elemento 1 no índice 0: " + umVetor[0]);
System.out.println("Elemento 2 no índice 1: " + umVetor[1]);
System.out.println("Elemento 3 no índice 2: " + umVetor[2]);
```

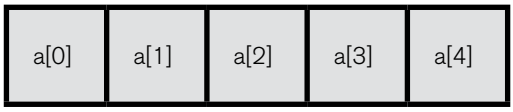
É possível usar a seguinte forma para criar e inicializar um vetor:

```
int[] umVetor = {100, 200, 300,400, 500, 600,700, 800, 900, 1000};

//nesta forma o tamanho do vetor é determinado pelo número de valores forne-
cidos // entre as chaves e separados por vírgulas
```

Também é possível usar vetores de vetores, ou seja, vetores com mais de uma dimensão, chamados de multidimensionais, usando dois ou mais conjuntos de colchetes como por exemplo *String [][]* valores. Cada elemento portanto deve ser acessado pelo valor do número do índice correspondente.

Na linguagem Java um vetor multidimensional é um vetor que contém vetores como componentes. Como consequência, as linhas podem ter tamanhos variados.



int a[] = new int[5];

Figura 1.7 – Um vetor unidimensional com 5 posições.

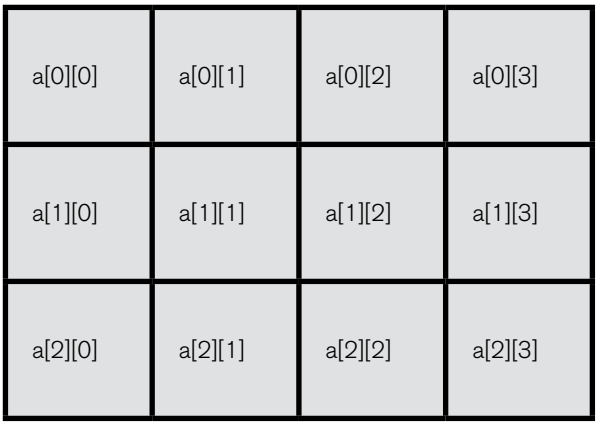


Figura 1.8 –Um vetor bidimensional com 3 linhas e 4 colunas.

Nas figuras 1.7 e 1.8 cada elemento é identificado por uma expressão de acesso ao vetor da forma a*[linha][coluna]*; a é o nome do vetor e linha e coluna são os índices unicamente identificados para cada elemento do vetor, por número de linha e coluna.

Vamos usar um programa como exemplo:

```
class MultiDim {
   public static void main(String[] args) {
       String[][] nomes = { {"Sr. ", "Sra. ", "Srta. "},
                            {"Silva", "Santos"}
       };
       // Sr. Silva
       System.out.println(nomes[0][0] + nomes[1][0]);
       // Srta. Santos
       System.out.println(nomes[0][2] + nomes[1][1]);
   }
}

A saída do programa será:
Sr. Silva
Srta. Santos
```

Você pode usar a propriedade length para determinar o tamanho de cada vetor. Veja o exemplo:

```
System.out.println(umVetor.length);

//esta linha imprime o tamanho do vetor umVetor
```

1.10.3 Copiando vetores

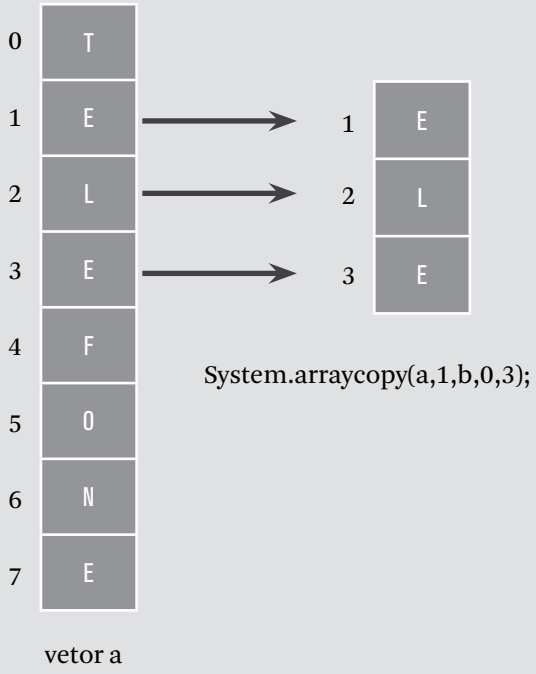
Como já dissemos, a linguagem Java possui muitas classes que auxiliam o desenvolvedor em tarefas como manipulação de vetores. Uma tarefa útil e muito usada é a cópia de vetores. A classe *System* possui um método *arraycopy* que você pode usar para fazer a cópia dos dados de um vetor para outro. O método possui a seguinte sintaxe:

```
public static void arraycopy(Object src, int srcPos,
              Object dest, int destPos, int length);
```

Os dois argumentos src e dest especificam o vetor de origem e destino respectivamente. Os três argumentos int especificam a posição inicial no vetor de origem, a posição inicial no vetor de destino e o número de elementos a serem copiados. Observe o programa a seguir:

```
public class CopiaParte {
   public static void main(String[] args) {
      //(1) cria o array "a" e o preenche com os caracteres da palavra
"telefone"
      char[] a = { 't','e','l','e','f','o','n','e'};
      //(2) Copia apenas a parte "ele" para o array b, usando “System.
arraycopy()”
      char [] b = new char[3];
      //primeiro é preciso reservar espaço para b
      System.arraycopy(a, 1, b, 0, 3);
```

```
        //agora copiamos a palavra "ele"
        //(3) exibe o conteúdo de "b"
        for (int i=0; i < b.length; i++) {
            System.out.println("b[" + i + "]=" + b[i]);
        }
    }
}
```



Existem outras abordagens de cópia de vetores: usando o loop for, o método *clone()* e o método *copyOf()* da classe *Arrays*. Nesta classe existem vários métodos muito úteis para a manipulação de vetores, como por exemplo:

- Procurar dentro de um vetor por um valor específico e obter o índice ao qual ele se refere (método *binarySearch*)
- Comparar dois vetores para determinar se eles são iguais ou não (método *equals*)
- Preencher um vetor com um determinado valor (método *fill*)
- Classificar um vetor em ordem crescente (método *sort*)
- Etc.

1.11 Conversões simples de tipos

Qualquer linguagem de programação possui várias formas de se fazer conversões entre seus tipos suportados e Java não é diferente. Porém, as conversões mais simples que podemos ter são as conversões entre números e *strings* e vamos mostrar algumas das formas. Não se esqueça que as outras formas você precisa investigar na API do Java para verificar outras mais apropriadas para o seu problema.

CONEXÃO

Os links a seguir mostram onde encontrar mais referências sobre a linguagem Java:
http://www.oracle.com/technetwork/java/api-141528.html: Especificações de todas as API Java
http://docs.oracle.com/javase/7/docs/api/: Especificação da API do Java 7

1.11.1 Convertendo strings para números

É muito comum armazenar dados numéricos em objetos *string*, por exemplo, o CPF é um caso. Todo CPF é um número com 11 dígitos e dependendo do programa que é desenvolvido é necessário fazer operações aritméticas com este dado. Porém dependendo do programa, ao obter o valor do CPF do usuário pelo teclado, o valor é guardado em uma *string* sendo necessária sua conversão posteriormente.

Para tratar este tipo de situação, a classe *Number* possui subclasses que envolvem os tipos primitivos numéricos (*Byte, Integer, Double, Float, Long e Short* – perceba que todas essas são classes, veja as iniciais maiúsculas, e não os tipos primitivos). Cada uma dessas classes possui um método de classe chamado *valueOf()* que converte uma *string* em um objeto daquele tipo. Veja o seguinte exemplo:

```
1  public class TesteValueOf {
2      public static void main(String[] args) {
3          if (args.length == 2) {
4              // converte strings em numeros
5              float a = (Float.valueOf(args[0])).floatValue();
```

```
6             float b = (Float.valueOf(args[1])).floatValue();
7             // algumas contas
8             System.out.println("a + b = " + (a+b));
9             System.out.println("a - b = " + (a-b));
10            System.out.println("a * b = " + (a*b));
11            System.out.println("a / b = " + (a/b));
12            System.out.println("a % b = " + (a%b));
13        }
14        else {
15             System.out.println("Digite dois números:");
16         }
17     }
18  }
```

Este programa possui várias lições e uso de conceitos que já aprendemos.

A linha 2 possui a declaração de um vetor sem tamanho definido do tipo *String* chamado *args*. Como já estudamos na seção anterior, é possível saber o tamanho do vetor por meio da propriedade *length* e a linha 3 usa esta propriedade para verificar se o tamanho do vetor é igual a 2 elementos. Em caso positivo, o resto do programa é executado. Lembra-se do *if*?

O método main de um programa em Java aceita como parâmetros os valores digitados na linha de comando. Por exemplo: se um programa Java que converte valores de graus *Celsius* para *Fahrenheit* chama converte.java, é possível estruturar o programa para ele executar pela linha de comando da seguinte forma:

```
C:\> java converte 100

//Neste caso, o programa vai converter 100 graus Celsius para Fahrenheit. A
saída seria:

212.0

C:\>
```

Como podemos ver, o 100 na linha de comando será o primeiro valor do índice 0 do vetor *args[]* usado no programa main, ou seja, *args[0]*=100. Se houvesse outros valores separados por espaço, eles seriam atribuídos aos próximos índices do vetor *args[]*.

As linhas 5 e 6 são semelhantes. Vamos analisar a linha 5 e você analise a linha 6 posteriormente.

```
5  float a = (Float.valueOf(args[0])).floatValue();
```

A variável a vai receber o resultado da execução de 2 métodos: o *valueOf()* em primeiro lugar, pois está dentro dos parênteses, e depois o resultado será passado para o método *floatValue()*. Como exemplo, vamos supor que o usuário executou o programa com os seguintes valores: 4.5 e 87.2 portanto, *args[0]*=4.5 e *args[1]*=87.2.

Na linha 5, o método *valueOf()* obtém a string "4.5" e a transforma para o objeto *Float*. Em seguida o método *floatValue()* transforma o valor do objeto *Float* para o *float* 4.5. Isto ocorre também na linha 6 para o *args[1]*.

Com as variáveis a e b contendo seus valores convertidos para *float*, seguem algumas operações aritméticas comuns nas linhas seguintes.

Uma observação a ser feita é que as subclasses da classe *Number* que implementam tipos numéricos primitivos também possuem um método chamado *parse____()*, por exemplo *Integer.parseInt(), Double.parseDouble(),* que podem ser usados para converter strings para os números primitivos. Desde que um tipo primitivo é retornado ao invés de um objeto, o *parseFloat()* é mais direto que o *valueOf()*. A linha 5 (e 6) poderia ser escrita de uma maneira mais direta assim:

```
float a = Float.parseFloat(args[0]);
```

1.11.2 Convertendo números para strings

Também é frequente converter um número para uma *string*. Há várias formas de se fazer isso e vamos mostrar uma delas.

```
int i;
// Concatena "i" com uma string vazia; a conversão é feita “na mão”, usando o “+”
String s1 = "" + i;

ou

// O método de classe valueOf().
String s2 = String.valueOf(i);
```

Cada uma das classes *Number* possui um método *toString()* para converter um número em uma *string*. Veja o exemplo a seguir:

```
public class TestaToString {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int ponto = s.indexOf('.');

        System.out.println(ponto + " dígitos " + "antes do ponto decimal.");
        System.out.println((s.length()-ponto-1)+"  dígitos  depois  do  ponto
decimal.");
    }
}
```

```
A saída do programa é:

3 dígitos antes do ponto decimal.
2 dígitos depois do ponto decimal.
```

ATIVIDADES

01. Verifique se os clientes de uma loja excederam o limite do cartão de crédito. Para cada cliente, temos os seguintes dados:

- número da conta corrente
- saldo no início do mês

- total de todos os itens comprados no cartão
- total de créditos aplicados ao cliente no mês
- limite de crédito autorizado

Todos esses dados são inteiros. O programa deve mostrar o novo saldo de acordo com a seguinte fórmula (saldo inicial + despesas – créditos) e determinar se o novo saldo excede o limite de crédito. Para aqueles clientes que o novo saldo excedeu, o programa deve mostrar a frase: "Limite de crédito excedido".

02. Considere o seguinte fragmento de código:

```
if (umNumero >= 0)
   if (umNumero == 0)
       System.out.println("Primeira string");
else System.out.println("Segunda string");
System.out.println("Terceira string");
```

a) O que você acha que será impresso se umNumero = 3?
b) Escreva um programa de teste contendo o código acima; assuma que umNumero = 3. Qual a saída do programa? Foi o que você respondeu na questão a? Explique a saída; em outras palavras, qual é o fluxo de controle do fragmento do código?
c) Usando somente espaços e quebras de linha, reformate o fragmento para torna-lo mais legível.
d) Use parênteses, colchetes, chaves e o que for necessário para deixar o código mais claro.

03. Vamos praticar um pouco de inglês. Considere a seguinte string:

```
String hannah = "Did Hannah see bees? Hannah did.";
```

a) Que valor é mostrado pela expressão: *hannah.lenght()* ?
b) Qual é o valor retornado pela chamada de método *hannah.charAt(12)*?
c) Escreva uma expressão para se referir à letra b na *string.*

04. No programa seguinte, qual é o valor do resultado do *valueOf* após cada linha numerada? Neste exercício você terá que pesquisar sobre a classe *StringBuilder* para responder.

```
public class ComputeResult {
   public static void main(String[] args) {
       String original = "software";
       StringBuilder resultado = new StringBuilder("olá");
       int indice = original.indexOf('a');

/*1*/   resultado.setCharAt(0, original.charAt(0));
/*2*/   resultado.setCharAt(1, original.charAt(original.length()-1));
/*3*/   resultado.insert(1, original.charAt(4));
/*4*/   resultado.append(original.substring(1,4));
/*5*/   resultado.insert(3, (original.substring(indice, indice+2) + " "));

       System.out.println(resultado);
   }
}
```

05. Um anagrama é uma palavra ou frase resultante do rearranjo das letras e produzindo outra palavra, usando as letras originais apenas uma vez. Por exemplo "Iracema" é um anagrama de "América", "Amor" tem os seguintes anagramas: "Roma", "Armo", "Mora", etc.. Escreva um programa que verifica se um *string* é um anagrama de outra *string*. O programa deve ignorar espaços e pontuações.

REFLEXÃO

O desenvolvimento em Java requer um esforço além dos livros didáticos. Aliás, qualquer linguagem é assim. Cabe ao estudante se esforçar em conhecer outros recursos da linguagem os quais não são explorados nos livros. Estudar a documentaçã, saber navegar dentro da documentação, saber pesquisar dentro dela é fundamental para o bom aprendizado de qualquer linguagem.

LEITURA

Java é uma linguagem com muito material publicado e na internet.

A seguir apresentamos alguns materiais que podem ajudá-lo no seu aprendizado:

- SIERRA, K. **Use a cabeça** Java. Rio de Janeiro: Altabooks, 2010.
- ECKEL, B. **Thinking in Java**. Disponível em http://www.mindview.net/Books/TIJ/.

- **Java Progressivo**. Disponível em: http://www.javaprogressivo.net/2013/02/Curso-Java-Progressivo-Tutorial-Inicial.html

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Bookman, 2002.

FLANAGAN, D. **Java o guia essencial**. Bookman, 2006.

GARY, C.; HORSTMANN, C. Core **Java 2:** Fundamentos. Makron Books, 2001.

GARY, C.; HORSTMANN, C. Core **Java 2:** Recursos Avançados. Makron Books, 2001.

HUBBARD, J. R. **Programação com Java**. Bookman, 2006.

2

Interfaces Gráficas e Orientação a Objetos

Todo mundo gosta de interagir com um programa de uma maneira agradável e objetiva. Com a explosão da internet, cada dia que passa exigimos que os programas tenham uma aparência e um comportamento parecido uns com os outros para aumentar a nossa produtividade e experiência no uso da informática. Com a popularização dos celulares isso aumenta ainda mais.

Porém ainda existe uma grande demanda de programas que rodam exclusivamente em computadores *desktop*, em redes e que não precisam estar na internet ou ter os mesmos recursos que a internet oferece. Muitos programas que usamos no dia a dia são assim como os editores de texto, planilhas eletrônicas, *softwares* para apresentações, jogos, entre outros.

Este capítulo vai mostrar os principais componentes para gerar programas para esta categoria de *software*. Bons estudos!

OBJETIVOS

O objetivo principal deste capítulo é estudar os seguintes tópicos:

- Introdução às IDEs, especialmente o Netbeans
- Criação de interfaces gráficas simples
- Classes e objetos
- Atributos e métodos

2.1 Introdução

Antes de estudarmos as interfaces gráficas vamos apresentar um ambiente de desenvolvimento muito conhecido e usado por vários desenvolvedores Java: o *Netbeans*.

O *Netbeans* foi criado em 1997 por um grupo de estudantes da antiga Tchecoslováquia e tinha como ideia principal ser um ambiente de desenvolvimento parecido com o *Delphi*. O *Delphi* na ocasião era um ambiente bastante utilizado por oferecer vários recursos de arrastar e soltar além de outras facilidades que não era encontrada em ambientes de desenvolvimento para Java. Ainda mais na questão de desenvolvimento de interfaces gráficas.

Uma das vantagens de se utilizar um ambiente para o desenvolvimento de interfaces gráficas é a produtividade e a maneira visual de prever como a tela ficará na execução do programa. Porém isso traz uma desvantagem a qual era presente inclusive no *Delphi* na época: o uso do arrastar e soltar para montar as telas gera um código fonte o qual pode ser difícil de ser entendido pelos programadores. Além disso, ele pode não ser tão otimizado quanto feito "na mão".

A grande vantagem de se utilizar um ambiente de programação como o *Netbeans* é usufruir das várias ferramentas de desenvolvimento embutidas como por exemplo as ferramentas de depuração de código (*debugger*), integração e manipulação de servidores de aplicações como o *JBoss, Apache Tomcat* e outros, além de criação de projetos específicos para a *Web*, Celulares, etc.. O *Netbeans* também possui uma paleta de componentes específicos para a criação de interfaces gráficas.

2.2 Conhecendo o Netbeans

Observe a figura 2.1. Ela mostra a tela inicial do *Netbeans* após a criação de um projeto simples do tipo Java SE. Para criar uma aplicação com componentes gráficos e usar os recursos de arrastar e soltar, é necessário criar um novo arquivo e em seguida escolher entre os vários tipos de containers (veremos isso mais a frente) entre eles: *JFrame, JPanel, JFormDialog, JInternalFrame* e outros.

A figura 2.2 mostra a tela do *Netbeans* com a paleta de componentes do pacote *Swing* habilitada. A figura ainda mostra um frame com um botão adicionado, obtido por meio do arraste do componente, desde a paleta até a posição desejada.

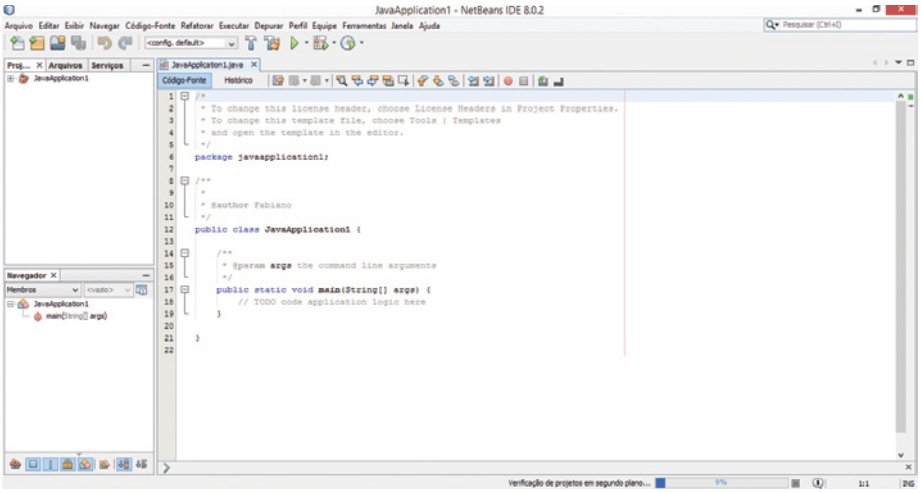


Figura 2.1 –Tela inicial do Netbeans após a criação de um projeto.

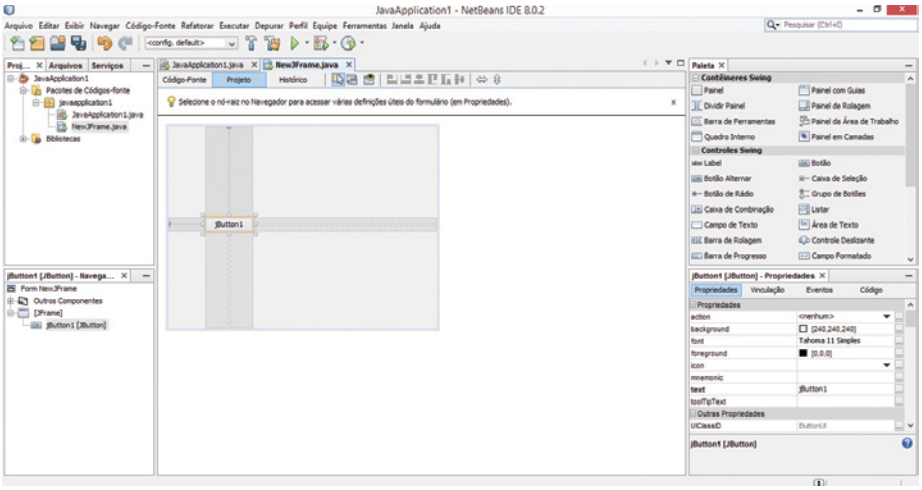


Figura 2.2 – Projeto no *Netbeans* com paleta de componentes *Swing*.

2.3 Criação de interfaces gráficas

O objetivo de uma *interface* gráfica com o usuário (GUI – *Graphical User Interface*) é prover uma forma amigável para poder interagir com um aplicativo. A linguagem Java possui componentes que permitem ao desenvolvedor construir programas com esta característica. Um bom exemplo é o programa que usamos para fazer a declaração de imposto de renda e envio dos dados. É um programa feito em Java que usa os componentes que vamos tratar brevemente aqui.

A linguagem Java possui algumas alternativas para o desenvolvimento de interfaces gráficas para aplicações *desktop*. Todas elas são bibliotecas de componentes baseadas em uma hierarquia de classes a qual vamos tratar mais tarde mas é fundamental que o desenvolvedor a conheça.

Inicialmente o Java contou com uma biblioteca chamada AWT (*Abstract Window Toolkit*). Esta biblioteca possui alguns principais componentes usados nas aplicações como por exemplo botões, caixas de texto, menus e eventos. Componentes mais modernos e encontrados nas aplicações atualmente não existiam no AWT. Depois do AWT apareceu o *Swing Application Framework*. Por muito tempo o *Swing* foi a biblioteca mais utilizada, estudada e publicada no ambiente Java e muitas aplicações ainda são desenvolvidas neste *framework*. Porém ele foi descontinuado da versão oficial do Java mas ainda é mantido por terceiros. Atualmente a Sun apoia o desenvolvimento usando uma biblioteca chamada *JavaFX* porém a documentação e o aceite pela comunidade de desenvolvedores ainda não é tão significativo quanto foi o *Swing* e por isso vamos dar preferência a este.

Para você ter uma ideia do que é o *Swing*, observe o programa abaixo:

```
import java.awt.EventQueue;
import javax.swing.JFrame;

public class Swing1 extends JFrame {
    public Swing1() {
        setTitle("Primeiro exemplo");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
            SimpleExample ex = new SimpleExample();
                ex.setVisible(true);
            }
        });
    }
}
```

O programa é muito simples. Ele não faz nada além do que colocar uma janela na tela com o nome "Primeiro exemplo". Observe que no programa principal nós colocamos a chamada da tela em um agendador do próprio *Swing* a fim de evitar problemas em algumas situações. Vamos ver isso mais tarde.

Vamos agora estudar alguns componentes.

2.4 Entrada e saída de dados com *Swing*

O *Swing* possui uma classe chamada *JOptionPane* muito utilizada para a entrada e saída de dados de usuários. Vamos mostrar o seu uso com um programa:

```
import javax.swing.JOptionPane;

1  public class Adicao {
2      public static void main(String[] args) {
3          String num1, num2;
4          int n1, n2, soma;
5
6          num1 = JOptionPane.showInputDialog("Digite o primeiro número");
7          n1 = Integer.parseInt(num1);
8
9          num2 = JOptionPane.showInputDialog("Digite o segundo número");
10         n2 = Integer.parseInt(num2);
11
12         soma = n1+n2;
13
14         JOptionPane.showMessageDialog(null, "A soma eh:"+soma);
15     }
16 }
```

A figura 2.3 mostra o resultado do programa *Adicao.java*. Perceba no programa que para usar as classes *Swing* é obrigatório a importação do pacote *javax.swing.JOptionPane*. O método *showInputDialog()* serve para pedir uma entrada do usuário. Esta entrada é sempre uma *String* e como vamos somar as duas entradas (e fazer uma conta aritmética) é preciso converter a entrada para int. Vimos isso no capítulo 1.

O método *showMessageDialog()* possui vários construtores e foi usado na linha 14 a forma mais simples. Observe que o segundo parâmetro do método é a *String* que desejamos mostrar na tela.

CONEXÃO

Novamente enfatizamos o estudo da API do Java e neste caso especificamente a API do Swing para poder saber as possíveis formas de uso da classe JOptionPane e seus inúmeros (e úteis) métodos para criar diálogos com o usuário. Não deixe de ver o link: http://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html

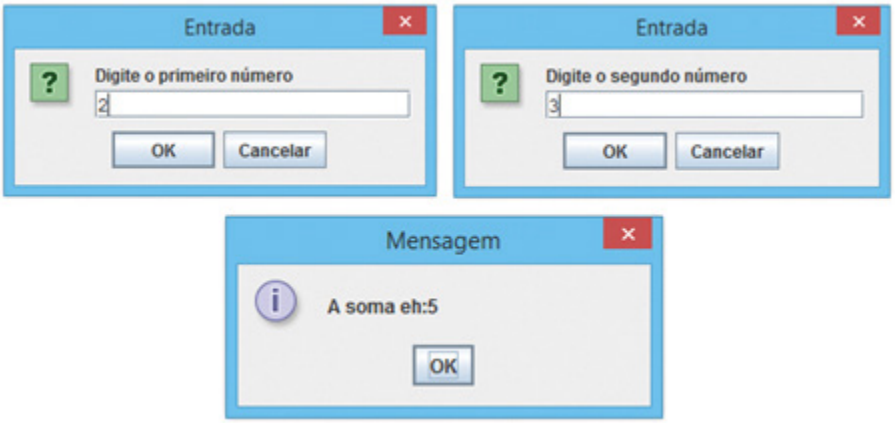


Figura 2.3 – Uso da classe *JOptionPane*

2.5 Visão geral do Swing

É claro que ler e mostrar dados usando *JOptionPane* é apenas o começo e em uma aplicação real existem mais inúmeras formas para se obter e mostrar dados. É possível usar em Java componentes gráficos como caixas de texto (*JTextField*), botões (*JButton*), check boxes (*JCheckBox*), combo boxes (*JComboBox*), listas (*JList*) e outros.

Os componentes mencionados precisam estar "apoiados" e contidos em outros componentes chamados containers. O *Swing* possui 3 classes containers de uso geral que servem para receber os outros componentes: *JFrame, JDialog e JApplet.*

A hierarquia da Error! Reference source not found. mostra a hierarquia de classes que compõem os componentes e os containers. É muito importante conhecer estas classes e métodos para um melhor entendimento do *Swing*.

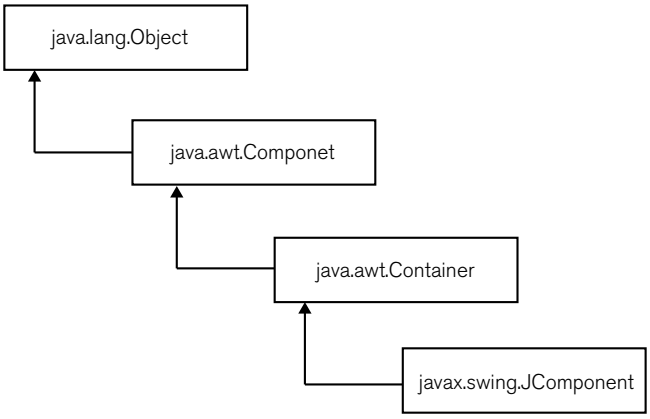


Figura 2.4 – Hierarquia da classe JComponent.

A figura 2.5 mostra o que ocorre com os componentes e os containers. Temos o container Frame o qual contém um painel, uma barra de menu verde (sem o menu ainda) e um label amarelo como componentes. A figura ainda mostra como ficaria a hierarquia simplificada entre o JFrame, o JLabel e uma barra de menu (MenuBar).

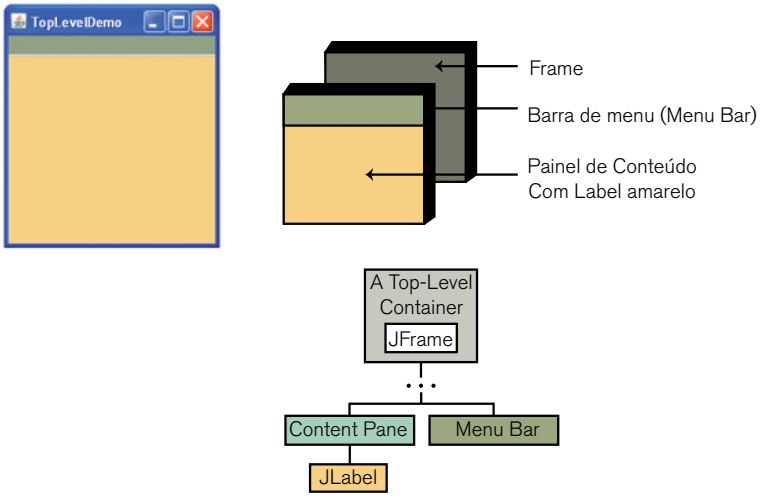


Figura 2.5 – Um frame criado por uma aplicação. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html6.

O código para gerar o frame da figura 2.5 está mostrado logo em seguida.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class TopLevelDemo {
6      private static void mostra() {
7          //Cria e configura a janela
8          JFrame frame = new JFrame("TopLevelDemo");
9          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10
11         //Cria a barra de menus com fundo verde
12         JMenuBar greenMenuBar = new JMenuBar();
13         greenMenuBar.setOpaque(true);
14         greenMenuBar.setBackground(new Color(154, 165, 127));
15         greenMenuBar.setPreferredSize(new Dimension(200, 20));
16
17         //Cria um label amarelo para colocar no painel
18         JLabel yellowLabel = new JLabel();
19         yellowLabel.setOpaque(true);
20         yellowLabel.setBackground(new Color(248, 213, 131));
21         yellowLabel.setPreferredSize(new Dimension(200, 180));
22
23         //Configura a barra de menu e coloca no painel
24         frame.setJMenuBar(greenMenuBar);
25         frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
26
27         //Mostra a janela.
28         frame.pack();
29         frame.setVisible(true);
30     }
31
32     public static void main(String[] args) {
33         javax.swing.SwingUtilities.invokeLater(new Runnable() {
34             public void run() {
35                 mostra();
36         }
37     });
38 }
```

Como percebemos, cada programa em *Swing* terá pelo menos um container principal, chamado *top-level container*. Ele será a raiz de uma hierarquia de containers. Como regra, uma aplicação stand-alone com *Swing* terá uma hierarquia de containers com um *JFrame* na raiz. Por exemplo, se uma aplicação tiver uma janela principal e duas janelas de diálogo, então a aplicação terá três hierarquias de compartimentos e três top-level containers. Uma hierarquia terá um *JFrame* na raiz e cada uma das duas terá um *JDialog* na raiz.

2.6 Criando o primeiro formulário

Vamos começar com um programa bem simples e rápido: mostrar uma janela na tela. Observe os comentários e o código a seguir:

```
    import java.awt.EventQueue;
    import javax.swing.JFrame;

    public class PrimeiroExemplo extends JFrame {
       public PrimeiroExemplo() {
    // ***depois vamos inserir um botão aqui***
    // nas linhas abaixo configuramos a janela
           setTitle("Exemplo Simples");
           setSize(300, 200);
           setLocationRelativeTo(null);
           setDefaultCloseOperation(EXIT_ON_CLOSE);
       }

    public static void main(String[] args) {
/* aqui nós criamos uma instância da classe PrimeiroExemplo e a mostramos na tela.
O método invokeLater() coloca o programa na Swing Event Queue – Fila de eventos
do Swing. Ele é usado (e recomendado) para garantir que todas as atualizações que
são feitas na interface sejam protegidas contra concorrência, ou seja, em outras
palavras, o método é usado para evitar que o programe “trave” em alguma situações.
    */
       EventQueue.invokeLater(new Runnable() {
              @Override
              public void run() {
```

```
                PrimeiroExemplo ex = new PrimeiroExemplo();
                ex.setVisible(true);
            }
        });
    }
}
```

Percebemos que no código anterior temos uma classe chamada PrimeiroExemplo que herda de uma superclasse chamada *JFrame*. Os comandos de configuração da janela são métodos da classe *JFrame* que são usados para definir o tamanho da janela, sua localização inicial e a ação que será executada quando a janela for fechada.

Podemos incrementar essa janela com um botão (classe *JButton*). O botão nada mais fará do que fechar a janela e para isso, é necessário entender sobre eventos e listeners. Este assunto em especial será tratado no Capítulo 5 porém vamos iniciar o assunto aqui.

O código a seguir deve substituir o comentário referente ao botão no código anterior.

```
1  JButton botaoSair = new JButton(" Sair ");
2  botaoSair.setToolTipText("Clique-me");
3  setLayout(new FlowLayout());
4  botaoSair.addActionListener(new ActionListener() {
5      @Override
6      public void actionPerformed(ActionEvent event) {
7          System.exit(0);
8      }
9  });
10
11 add(botaoSair);
```

A linha 1 define e instancia um botão, chamado botaoSair. Na linha 2 usamos um método da classe JButton no qual colocamos uma “dica” (tooltip) no botão. A linha 3 configura o layout do JFrame como FlowLayout. Veremos sobre

isso mais a frente. As linhas de 4 a 9 definem a resposta do clique do botão e na linha 7 o programa é finalizado. A linha 11 insere o botão definido no frame.

A Figura 2.6 mostra o resultado final do código do exemplo do botão.

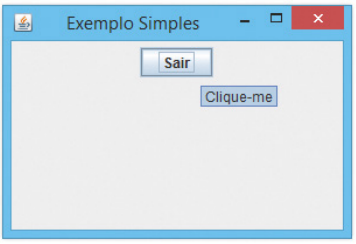


Figura 2.6 – Janela com um botão e uma tooltip

2.7 Princípios de orientação a objetos

Entender o que é um objeto é fundamental. Os objetos são a chave para a tecnologia orientada a objetos. Ao nosso redor você encontrará vários objetos: seu computador, seu carro, este texto, enfim, tudo.

Os objetos do mundo real têm duas características: estado e comportamento. Um cachorro por exemplo tem os seguintes estados: nome, raça, tamanho, peso, etc. e os seguintes comportamentos: latir, brincar, correr, dormir, etc.. Uma bicicleta também contém estados: marcha atual, velocidade, etc. e comportamentos: mudar marcha, acelerar, frear, etc. Identificar os estados e comportamentos de cada objeto do mundo real é uma forma de exercitar a orientação a objetos. Portanto, faça essa experiência: busque os estados e comportamentos dos objetos que você está vendo agora. Você notará que existem objetos que contém outros objetos, além disso, perceberá que um objeto depende de outros. Todas essas observações fazem parte da orientação a objetos.

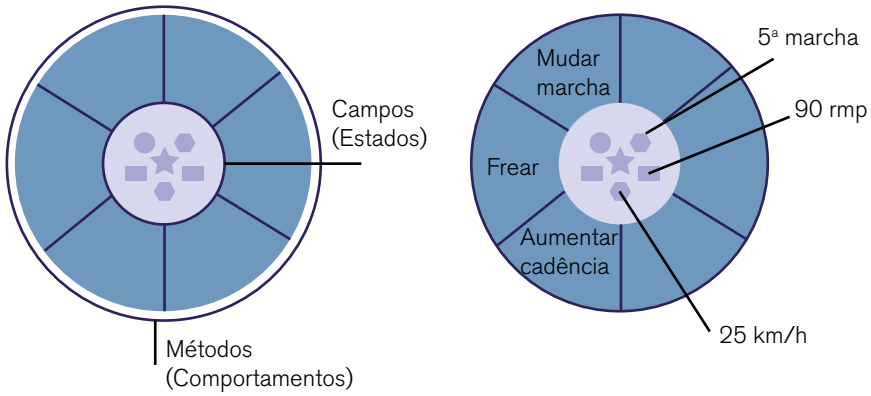


Figura 2.7 – Representação de um objeto como software e um exemplo: uma bicicleta.

Quando falamos em *software*, um objeto terá conceitos iguais aos objetos do mundo real: eles consistem de estado e comportamento. Um objeto guarda seu estado em campos (ou variáveis) e expõe seu comportamento por meio de métodos (funções). Os métodos operam nos estados internos de um objeto e servem como o mecanismo primário para a comunicação entre objetos. Esconder o estado interno de um objeto e fazer com que todas as interações sejam feitas pelos métodos é chamado de encapsulamento de dados.

Usar estados e fornecer métodos para mudar os estados, o objeto fica com o controle de como o mundo externo pode fazer para interagir com o objeto. Por exemplo, se a bicicleta tem 10 marchas, um método para mudar as marchas pode rejeitar qualquer valor que seja menor que 1 ou maior que 10.

O uso de objetos empacotados e encapsulados desta forma possibilita vários benefícios, como:

MODULARIDADE	o código fonte para um objeto pode ser escrito e mantido independentemente do código fonte de outros objetos, componentizando (modularizando) assim o desenvolvimento de software

OCULTAMENTO DA INFORMAÇÃO	usando apenas os métodos dos objetos, os detalhes da implementação interna permanecem escondidos do mundo externo
REUSO DE CÓDIGO	se um objeto já existe, você poderá usá-lo no seu programa.
PLUGABILIDADE E FACILIDADE DE DEPURAÇÃO	se um objeto particular se torna problemático, é possível removê-lo da aplicação e plugar outro objeto no lugar. Isto é parecido com a reposição de peças mecânicas de um carro.

2.8 Classes

Vamos usar o exemplo da bicicleta para poder entender o conceito de classes e objetos. No mundo real temos vários tipos de bicicleta: as speed (usadas para corridas de longa distância), as mountain bike, e o tipo padrão. Existem outras mas essas já são suficientes para o nosso estudo. Mesmo tendo usos e características diferentes, toda bicicleta tem um conjunto de propriedades e comportamentos em comum.

No vocabulário da orientação a objetos, dizemos que uma bicicleta é uma instância da classe de objetos conhecida como bicicleta. Uma classe é um padrão no qual objetos individuais são criados.

Já dissemos que um objeto é implementado por um conjunto de variáveis e funções. Sendo assim, vamos considerar que uma bicicleta pode ser implementada da forma mostrada na Listagem 1 em Java.

```
    class Bicicleta {
        int cadencia = 0;
        int velocidade = 0;
        int marcha = 1;
        void mudarCadencia (int novoValor) {
            cadencia = novoValor;
        }
        void mudarMarcha(int novoValor) {
            marcha = novoValor;
        }
        void acelerar(int incremento) {
            velocidade = velocidade + incremento;
        }
        void brecar(int decremento) {
            velocidade = velocidade - decremento;
        }
        void imprimirEstado() {
            System.out.println("Cadencia:" + cadencia + " Velocidade:" + velocidade
+ " Marcha:" + marcha);
        }
    }
```

Listagem 1

As variáveis cadência, velocidade e marcha representam os estados do objeto e os métodos mudarCadencia, mudarMarcha, acelerar, brecar e imprimiEstado definem a interação do objeto com o mundo real.

Esta classe não possui um método principal, ou seja, para ela poder funcionar é necessário que um método main exista nela, o que não é recomendável, ou em outra classe. Uma bicicleta na vida real não possui um método main, certo? A responsabilidade de criar objetos da classe Bicicleta é de outra classe.

Para declarar uma classe usamos a seguinte sintaxe:

```
Classe MinhaClasse{
   // campos, construtor e declarações de métodos
}
```

O corpo da classe, que é a área entre as chaves { e }, contém todo o código que trata do ciclo de vida dos objetos criados da classe: construtores para inicializar novos objetos, declarações dos campos que fornecem o estado da classe e seus objetos, e métodos para implementar o comportamento da classe e seus objetos.

A forma de declaração de classe mostrada é a mais simples. Outras construções são permitidas mas iremos ver isto mais a frente neste capítulo. Nós podemos também colocar modificadores public ou private no começo da declaração. Estes modificadores determinam as outras classes que poderão acessar a classe em questão.

Em geral, as declarações de classe podem incluir os seguintes componentes:

1. Modificadores como *public, private* e um número de outros que veremos mais tarde
2. O nome da classe, com a primeira letra em maiúscula
3. O nome da classe pai (superclasse), quando houver. Neste caso teremos que usar a palavra chave *extends* para indicar qual é a classe pai.
4. Uma lista de *interfaces* implementadas pela classe, separadas por vírgula. Neste caso temos que usar a palavra chave implements para indicar as *interfaces*.

O corpo da classe, rodeada pelas chaves { e }.

A Listagem 2 mostra um programa que testa a classe Bicicleta e cria objetos desta classe. Observe que nesta classe só temos basicamente a presença do método main e seu código de execução.

```
class DemoBicicleta {
   public static void main(String[] args) {
      // criar dois tipos de objetos Bicicleta
      Bicicleta bike1 = new Bicicleta();
      Bicicleta bike2 = new Bicicleta();

      // Chamando os métodos das bicicletas
      bike1.mudarCadencia(50);
      bike1.acelerar(10);
      bike1.mudarMarcha(2);
      bike1.imprimirEstado();

      bike2.mudarCadencia(50);
      bike2.acelerar(10);
      bike2.mudarMarcha(2);
      bike2.mudarCadencia(40);
      bike2.acelerar(10);
      bike2.mudarMarcha(3);
      bike2.imprimirEstado();
   }
}

   A saída deste teste imprime a cadência final, velocidade e marcha das duas
bicicletas:

Cadencia:50 Velocidade:10 Marcha:2
Cadencia:40 Velocidade:20 Marcha:3
```

Listagem 2

2.8.1 Declarando variáveis membro

Existem vários tipos de variáveis:

- Variáveis membro de uma classe – chamadas de campos
- Variáveis em um bloco ou métodos – chamadas de variáveis locais
- Variáveis em declarações de métodos – chamadas de parâmetros

A classe Bicicleta (*Error! Reference source not found.*) usa as seguintes linhas de código para definir seus campos:

```
int cadencia = 0;
int velocidade = 0;
int marcha = 1;
```

As declarações de campos são compostas por três partes:

1. Nenhum ou alguns modificadores como public ou private
2. O tipo do campo
3. O nome do campo

Os campos da classe Bicicleta são: cadencia, marcha e velocidade e todos são do tipo *int*. Eles são public e assim são acessíveis por qualquer método e objeto que possa acessar a classe.

2.8.2 Modificadores de acesso

Existem alguns modificadores de acesso em Java porém por enquanto vamos estudar o *public* e o *private*.

- O *public* faz com que o campo seja acessível para todas as outras classes
- O *private* faz com que o campo seja acessível somente dentro da própria classe

De acordo com o que já vimos sobre encapsulamento, é comum e recomendável que os campos sejam *private*. Isto significa que eles somente podem ser acessados diretamente pela classe Bicicleta. Porém, é necessário acessar esses campos e seus valores. Isto pode ser feito indiretamente por meio de métodos públicos que obtém o valor para nós. Veja os métodos *get e set* na

```
    public class Bicicleta {
       private int cadencia;
       private int marcha;
       private int velocidade;
       public Bicicleta(int cadenciaInicial, int VelocidadeInicial, int marchaI-
nicial){
           marcha = marchaInicial;
           cadencia = cadenciaInicial;
           velocidade = velocidadeInicial;
       }
       public int getCadencia() { return cadencia; }
       public void setCadence(int novo) { cadencia = novo; }
       public int getMarcha() { return marcha; }
       public void setMarcha(int novo) { marcha = novo; }
       public int getVelocidade() { return velocidade; }
       public void brecar(int decremento) { velocidade -= decremento; }
       public void acelerar(int incremento) { velocidade += incremento; }
    }
```

Listagem 3

2.8.3 Tipos e nomes de variáveis

Todas as variáveis devem ter um tipo. Podemos usar tipos primitivos como *int, float, boolean*, etc ou usar tipos referenciados como *strings, arrays* e objetos.

Todas as variáveis, sendo elas campos, variáveis locais ou parâmetros seguem as mesmas regras de nomeação e convenções. Em Java convenciona-se que toda variável deve ser iniciada com letras minúsculas e adotar a convenção *CamelCase.* Fique atento porque a convenção *CamelCase* também é aplicada para classes e métodos, exceto que nas classes, a primeira letra sempre é maiúscula e a primeira letra (ou palavra) no método deve ser um verbo.

Segundo a Wikipedia, "CamelCase é a denominação em inglês para a prática de escrever palavras compostas ou frases, onde cada palavra é iniciada com Maiúsculas e unidas sem espaços. É um padrão largamente utilizado em diversas linguagens de programação, como Java, Ruby, PHP e Python, principalmente nas definições de Classes e Objetos".

2.8.4 Definindo métodos

Vamos usar um exemplo para uma definição de um método:

```
public double calcularResposta(double areaAsa, int numeroDeMotores,
          ouble comprimento, double peso) {
   //estapas do cálculo
}
```

Os únicos elementos requeridos na declaração de métodos são: o tipo de retorno do método, o nome, o par de parênteses, e o corpo entre chaves { e }.

Em geral, a definição de um método tem seis componentes:

1. Modificadores: como *public, private* e outros
2. O tipo de retorno: o tipo de dado do valor retornado pelo método ou void se não há retorno
3. O nome do método
4. A lista de parâmetros entre parênteses: os parâmetros são separados por vírgula, seguido pelo tipo de dados e pelo nome do parâmetro. Se não houver parâmetros, usa-se os parênteses sem parâmetros
5. A lista de exceções, que veremos no Capítulo 4
6. O corpo do método, entre chaves

> Os métodos possuem uma assinatura. A assinatura é composta pelo nome do método, tipos dos parâmetros em ordem. A assinatura do método anterior é: *calcularResposta(double, int, double, double)*

Assim como os nomes de classes e variáveis, os nomes dos métodos também devem obedecer à convenção usada no Java (*CamelCase*). Além disso, é recomendável que todo método comece por um verbo (com a primeira letra em minúscula). Veja alguns exemplos:

correr, correrRapido, getBackground, getDados, compareTo, setX, isEmpty

Alguns nomes estão em inglês porque são métodos bastante usados em vários programas, e para manter uma maior legibilidade, ficam em inglês.

Normalmente um nome de método é único dentro da classe. Entretanto, um método pode ter o mesmo nome que outros métodos. Isto é chamado de sobrecarga.

2.8.5 Sobrecarga de métodos

O Java suporta a sobrecarga de métodos e a distinção de métodos com o mesmo nome é feita por meio de suas assinaturas. Isto significa que métodos dentro de uma classe podem ter o mesmo nome se eles possuírem uma lista de parâmetros diferentes. Veja o próximo trecho de código. Perceba que temos 3 métodos soma porém com assinaturas diferentes. Eles servem para o mesmo propósito, porém seus tipos de retornos e parâmetros são diferentes.

O método que será chamado depende somente do tipo do parâmetro passado e esta decisão será feita pelo compilador.

Você não pode declarar mais do que um método com o mesmo nome e o mesmo número de tipos de argumentos a fim de evitar erros de compilação. O compilador também não considera os diferentes tipos de retorno em métodos com o mesmo nome portanto você também não poderá criar métodos com a mesma assinatura e tipos de retorno diferentes.

```
public class Soma {
   public int soma(int x, int y) { return x+y;   }

   public String soma(String x, String y) { return x+y; }

   public double soma(double x, double y) { return x+y;
   }
}
```

2.8.6 Construtores

Uma classe contém construtores que são invocados quando um objeto da classe é criado. As declarações de construtores se parecem com declarações de métodos, exceto que eles tem exatamente o mesmo nome da classe e não possuem tipos de retorno. A classe Bicicleta a seguir possui o construtor:

```
public Bicicleta(int cadenciaInicial, int velocidadeInicial, int marchaInicial) {
   marcha = marchaInicial;
   cadencia = cadenciaInicial;
   velocidade = velocidadeInicial;
}
```

Para criar um novo objeto da classe Bicicleta chamado *minhaBike*, um construtor é chamado com o operador *new*:

```
Bicicleta minhaBike = new Bicicleta(30,0,8);
```

new Bicicleta(30,0,8) cria espaço na memória para o objeto e inicializa os seus campos.

É possível ter mais do que um construtor para a classe Bicicleta. Podemos criar um construtor sem argumentos, como o seguinte:

```
Public Bicicleta(){
   marcha = 1;
   cadencia = 10;
   velocidade = 0;
}
```

Bicicleta suaBike = new Bicicleta(); chama o construtor sem argumentos para criar um objeto chamado *suaBike*, da classe Bicicleta.

Ambos os construtores estão corretos e podem ser criados e usados porque eles tem uma lista de argumentos diferentes. Assim como os métodos em Java, o compilador diferencia os diferentes construtores baseado em suas assinaturas. O que não é permitido é escrever dois construtores com o mesmo número e tipos para a mesma classe.

Você não precisa escrever um construtor para sua classe porém isso não é recomendado. Quando não é criado um construtor, o compilador automaticamente cria internamente um construtor sem argumentos para a classe. Neste caso, o construtor padrão irá chamar o construtor sem argumentos da

superclasse e nesta situação, o compilador vai acusar que a superclasse não tem um construtor assim, portanto, é bom você verificar isso antes de usar este recurso. O melhor mesmo é sempre criar um construtor. Caso a classe não tenha uma superclasse explícita, será usado o construtor da classe *Object* o qual tem um construtor sem argumentos.

Você pode usar modificadores de acesso em uma declaração de construtor para controlar quais outras classes podem chamar o construtor.

2.8.7 Enviando informações para um método ou construtor

A declaração para um método ou para um construtor declara o número e o tipo dos argumentos daquele método ou construtor. Por exemplo, veja o método *calculaPagamento*:

```
    public double calculaPagamento(
               double quantia,
               double taxa,
               double valorFuturo,
               int tempo) {
        double interesse = taxa / 100.0;
        double parcial1 = Math.pow((1 + interesse),-tempo);
        double denominador = (1 - parcial1) / interesse;
        double   resposta   =   (-quantia/denominador)-   ((valorFuturo*parcial1)/
denominador);
        return resposta;
    }
```

Este método tem 4 parâmetros: *quantia, taxa de interesse, valor futuro e o número de períodos*. Os primeiros 3 são double e o quarto é int. Os parâmetros são usados no corpo do método e em tempo de execução receberão os valores que foram passados.

Parâmetros referem-se à lista de variáveis na declaração do método.
Argumentos são os valores atuais que são passados quando os métodos são chamados. Quando um método é executado, os argumentos usados "casam" com as declarações de parâmetros pelo tipo e pela ordem.

2.8.8 Sobre os parâmetros

Você pode usar qualquer tipo de dado para um parâmetro de um método ou construtor. Isto inclui tipos primitivos e tipos referenciados, como arrays e objetos.

Em relação aos nomes dos parâmetros, é importante saber que quando você declara um parâmetro, ele terá o seu nome conhecido por todo o escopo do método. Não é possível criar uma variável dentro do escopo com o mesmo nome nem do construtor da classe. Além disso, um parâmetro pode ter o mesmo nome que o campo da classe correspondente, isso é chamado de sombra do campo. Veja o exemplo a seguir:

```
public class Circulo {
   private int x, y, raio;
   public void setOrigem(int x, int y) {
       ...
   }
}
```

Perceba que a classe tem 3 campos: x, y e raio. O método *setOrigem* possui 2 parâmetros, cada um com o mesmo nome dos campos da classe e portanto são sombras dos campos. Dentro do método *setOrigem,* quando x e y forem utilizados, estarão sendo utilizados os parâmetros e não as variáveis dos campos da classe. Para acessar o campo é feito de outra forma, usando a palavra-chave this, que veremos mais tarde.

Os tipos primitivos como *int* ou *double* são passados por valor para os métodos. Isso significa que quaisquer mudanças nos valores dos parâmetros existirão somente no escopo do método. Quando o método finalizar e retornar, os parâmetros serão descartados e os seus valores perdidos. Veja o exemplo:

```
public class PassagemPorValor {
   public static void main(String[] args) {
       int x = 3;

       // chamando o método passando x como argumento
       metodo(x);

       // imprime x para ver se houve mudança no valor
       System.out.println("Após a execução do método, x = " + x);

   }

   // mudando o parâmetro no método
   public static void metodo(int p) {
       p = 10;
   }
}

A saída do programa será:

Após a execução do método, x = 3
```

Os tipos de dados referenciados (como objetos e *arrays*) também são passados por valor. Isto significa que quando o método retorna, a referência ainda será o mesmo objeto que antes. Entretanto, os valores dos campos do objeto podem ser mudados dentro do método se eles possuírem acesso a eles.

2.9 Objetos

Para estudarmos os objetos, vamos usar a Error! Reference source not found. Listagem 4 como exemplo. Ela possui duas classes: Ponto e Retangulo e uma classe para testar as duas.

O programa cria, manipula e mostra informação de vários objetos. A saída do programa é a seguinte:

```
Largura do ret1: 100
Altura do ret1: 200
Área do ret1: 20000
```

```
    Posição X do ret2: 23
    Posição Y do ret2: 94
    Posição X do ret2: 40
    Posição Y do ret2: 72
```

2.9.1 Criação de objetos

Vimos que uma classe é um padrão, um modelo para criar objetos. O código abaixo, obtido da Listagem 4, cria um objeto e o associa a uma variável:

```
    Ponto origem1 = new Ponto(23, 94);    // cria um objeto chamado origem1 da
classe Ponto
    Retangulo ret1 = new Retangulo(origem1, 100, 200);
    Retangulo ret2 = new Retangulo(50, 100); //cria um objeto (ret2) da classe
Retangulo
```

Cada uma dessas linhas tem 3 partes:

DECLARAÇÃO	o código em negrito são declarações de variáveis que associam um nome de variável a um tipo de objeto
INSTANCIAÇÃO	A palavra chave new é um operador Java que cria o objeto
INICIALIZAÇÃO	O operador new é seguido por uma chamada a um construtor que inicializa o novo objeto

```
public class Ponto {
    public int x = 0;
    public int y = 0;
    public Ponto(int a, int b) {
```

```
    x = a;
    y = b;
    }
}

public class Retangulo {
    public int largura = 0;
    public int altura = 0;
    public Point origem;

    public Retangulo ()        { origem = new Ponto(0, 0); }
    public Retangulo (Ponto p) { origem = p;    }
    public Retangulo (int l, int a) {
    origem = new Point(0, 0);
    largura = l;
    altura = a;
    }
    public Retangulo (Ponto p, int l, int a) {
    origem = p;
    largura = l;
    altura = a;
    }

    public void move(int x, int y) {
    origem.x = x;
    origem.y = y;
    }

    public int getArea() { return largura * altura; }
}

public class CriaObjeto {
    public static void main(String[] args) {

        // Declara e cria um objeto ponto e dois objetos retângulos.
        Point origem1 = new Point(23, 94);
        Retangulo ret1 = new Retangulo(origem1, 100, 200);
        Retangulo ret2 = new Retangulo(50, 100);
        // mostrando a largura, altura e área de ret1
        System.out.println("Largura do ret1: " + ret1.largura);
```

```
        System.out.println("Altura do ret1: " + ret1.altura);
        System.out.println("Área do ret1: " + ret1.getArea());

        // setando a posição do ret2
        ret2.origem2 = origem1;

      //mostrando a posição X e Y do ret2
        System.out.println("Posição X do ret2: " + ret2.origem.x);
        System.out.println("Posição Y do ret2: " + ret2.origem.y);

        // move ret2 e mostra sua nova posição
        ret2.move(40, 72);
        System.out.println("Posição X do ret2: " + ret2.origem.x);
        System.out.println("Posição Y do ret2: " + ret2.origem.y);
    }
}
```

Listagem 4

Aprendemos que para declarar uma variável a sintaxe é

```
Tipo nome_da_variável;
```

Isto faz com o que o compilador identifique que o nome da variável terá aquele tipo especificado. Para um tipo primitivo, esta declaração também reserva um espaço de memória. Para um objeto a sintaxe é semelhante:

```
Ponto origem1;
```

Se você declara origem1 como o exemplo, seu valor será indeterminado até que o objeto seja criado e associado. Somente declarar a variável não cria o objeto. Para a criação você deve usar a palavra-chave new. Veja a figura 2.8:

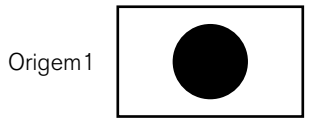


Figura 2.8 – Declaração da variável origem1

2.9.2 Instanciando uma classe

O operador *new* instancia uma classe alocando memória para um novo objeto e retornando uma referência para a memória. Além disso, o new chama o construtor da classe.

> Quando dizemos que "instanciamos uma classe" é o mesmo que dizer que "criamos um objeto". Quando você cria um objeto você cria uma instância, um elemento de uma classe.

O new requer um único argumento: a chamada de um construtor. O nome do construtor fornece o nome da classe para ser instanciada. O *new* retorna uma referência para o objeto criado e esta referência é usualmente associada a uma variável do tipo apropriado como:

```
Ponto origem1 = new Ponto(23,94);
int altura = new Retangulo().altura;
```

Veja a segunda forma, também é possível e aceitável ser usado diretamente em uma expressão.

2.9.3 Inicializando um objeto

Veja na Listagem 4 o código da classe Ponto. Esta classe só possui 1 construtor. O construtor é facilmente reconhecido porque ele tem o mesmo nome da classe e não possui tipo de retorno. O construtor da classe Ponto tem 2 argumentos, como declarados pelo código (*int a, int b*). A declaração seguinte atribui os valores 23 e 94 para esses argumentos:

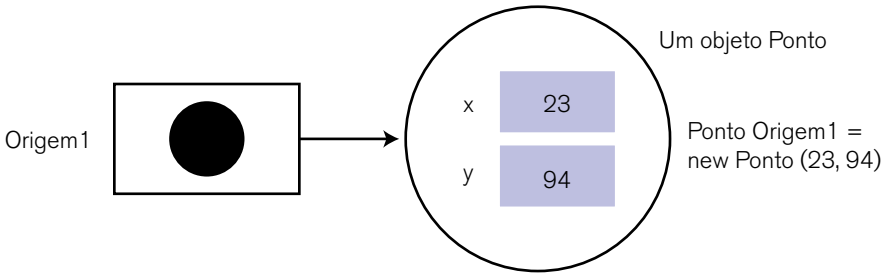


Figura 2.9 –Resultado da execução da criação do objeto origem1.

Olhando ainda para a Listagem 4 percebe-se que a classe Retangulo possui 4 construtores:

```
public Retangulo ()
public Retangulo (Ponto p)
public Retangulo (int l, int a)
public Retangulo (Ponto p, int l, int a)
```

Cada construtor permite que seja fornecido valores iniciais para a origem, altura e largura do retângulo usando tanto tipos primitivos e tipos referenciados. Se uma classe tem múltiplos construtores, eles devem ter diferentes assinaturas. O compilador diferencia os construtores baseados no número e o tipo dos argumentos.

Quando o compilador encontra o código seguinte, ele sabe chamar o construtor na classe Retangulo que precisa de um argumento Ponto seguido por 2 argumentos:

```
Retangulo ret1 = new Retangulo (origem1, 100, 200);
```

Isto chama um dos construtores de Retangulo que inicializa o campo da classe origem para *origem1*. Além disso, o construtor atribui 100 ao campo largura e 200 para altura. Portanto temos 2 referências para o mesmo objeto Ponto – um objeto pode ter múltiplas referências a ele, como mostra a Error! Reference source not found..

O código seguinte chama o construtor de Retangulo que pede dois inteiros como argumentos referentes à largura e altura. Se você ler o código dentro do construtor perceberá que ele cria um novo objeto Ponto o qual x e y são inicializados com 0.

```
Retangulo ret2 = new Retangulo(50,100);
```

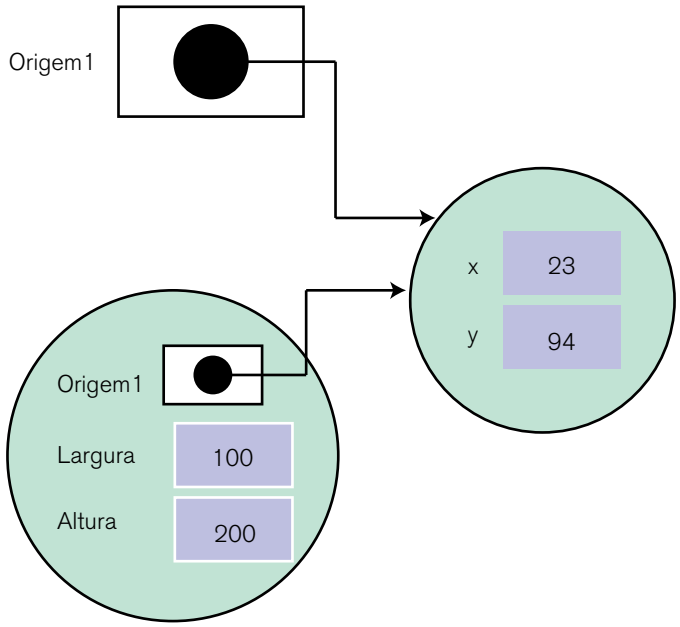


Figura 2.10 – Múltiplas referências de um objeto.

E o construtor do código abaixo não possui argumentos e é chamado de Construtor sem argumentos (é claro!)

```
Retangulo = new Retangulo();
```

Lembre-se: é muito recomendável que todas as classes devem ter pelo menos um construtor.

2.9.4 Usando objetos

Uma vez que você criou um objeto, você vai querer usá-lo. Você pode precisar usar o valor de um de seus campos, mudar um dos seus campos ou chamar um de seus métodos para fazer alguma coisa.

Os campos de um objeto são acessados pelo seu nome. Você deve usar um nome único e sem duplas interpretações.

```
System.out.println("A largura e a altura são:" + largura + "," + altura);
```

O código que fica fora da classe do objeto deve usar uma referência de objeto ou expressão, seguida de um ponto (.), seguida por um nome de campo simples, por exemplo:

```
System.out.println("Largura do ret1:" + ret1.largura);
System.out.println("Altura do ret2:" + ret1.altura);
```

Observando o programa, depois as informações sobre ret2 são mostradas com um código semelhante. Objetos do mesmo tipo tem sua própria cópia dos mesmos campos de instância. Cada objeto do tipo Retangulo tem campos largura, altura e origem. Quando você acessa um campo de instância por meio de uma referência de objeto, você referencia aquele campo em particular. Os dois objetos ret1 e ret2 no programa da Listagem 3-6 tem campos largura, altura e origem diferentes.

Para acessar um campo, você pode uma referência nomeada para um objeto, como nos exemplos anteriores, ou você pode usar qualquer expressão que retorne uma referência em objeto. O operador new retorna um objeto, logo, você pode usar o valor retornado por ele para acessar os campos de um novo objeto:

```
int altura = new Retangulo().altura;
```

A declaração acima cria um novo objeto Retangulo e imediatamente guarda sua altura. Na essência, a declaração calcula a altura padrão de um Retangulo.

Note que após esta declaração for executada, o programa não terá mais a referência criada para Retangulo, porque o programa nunca guardou a referência. O objeto fica sem referência e seus recursos são liberados para serem consumidos pela JVM.

2.9.5 Chamando métodos de objetos

Você também pode usar uma referência de objeto para invocar os métodos de um objeto. Para fazer isso basta você colocar o nome do objeto, o ponto, e o nome do método. Além disso, é necessário passar a lista de argumentos entre parênteses ou deixar a lista vazia quando não houver argumentos.

```
nomeDoObjeto.nomeDoMétodo(listaDeArgumentos);
ou
nomeDoObjeto.nomeDoMétodo();
```

A classe Retangulo tem dois métodos: getArea() para calcular a área do retângulo e move() para movimentar o retângulo. Veja como é feita a chamada de um método:

```
System.out.println("Área do ret1: " + ret1.getArea());
...
ret2.move(40, 72);
```

Assim como o acesso aos campos, é possível usar um nome de variável ou uma expressão que retorna um objeto para chamar um métodos:

```
new Retangulo(50,100).getArea();
```

Alguns métodos, como o getArea() retornam um valor. Neste caso, a chamada do método precisa ser feita com uma variável para receber o valor retornado:

```
int areaDoRetangulo = new Retangulo(50,100).getArea();
```

Lembre-se que chamar um método é o mesmo que mandar uma mensagem para ele, esta é a nomenclatura usada nas linguagens orientadas a objeto.

2.9.6 Retornando um valor de um método

Um método retorna para o código que o chamou quando (o que ocorrer primeiro):

1. Completa todos so comandos dentro do método
2. Encontra um comando return
3. Ocorre uma exceção

O tipo de retorno do método é colocado na declaração do método. Dentro do corpo do método é usado o comando return para retornar o valor.

Quando o método é declarado com void não há valor de retorno e não deve conter um comando return no seu corpo. Qualquer método que não é declarado como void obrigatoriamente deve ter um comando return. O retorno é feito usando o comando return e o valor de retorno, que pode ser explícito ou por meio de uma variável.

Outro detalhe a ser observado é que o tipo de retorno no comando return deve ser o mesmo que o tipo de retorno do método. Você não pode retornar um valor inteiro, por exemplo, de um método declarado como booleano.

Veja o exemplo a seguir:

```
// método que calcula a área de um retângulo
   public int getArea() {
       return largura * altura;    //lembre-se que altura e largura são int
   }
```

O método getArea() retorna um tipo primitivo mas também é possível retornar um objeto. Por exemplo, um programa que manipula objetos do tipo Bicicleta poderíamos ter um método como esse:

```
public Bicicleta quemEOMaisRapido(Bicicleta minha, Bicicleta sua, Ambiente amb) {
    Bicicleta maisRapida;
    // aqui vai o código para calcular qual bicicleta é mais rápida, dada sua ca-
dência     e ambiente(terreno e vento)
    return maisRapida;
}
```

Por enquanto o que vimos nesse capítulo é suficiente para fazermos algumas brincadeiras e desenvolver alguns programas simples. Vamos lá?

ATIVIDADES

01. Considere a seguinte classe

```
public class Exercicio1 {
    public int x = 7;
    public int y = 3;
}
```

a) Quais são as variáveis de instância?

b) Qual é a saída do seguinte código?

```
Exercicio1 a = new Exercicio1();
Exercicio1 b = new Exercicio1();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("Exercicio1.x = " + Exercicio1.x);
```

02. Baseado nos atributos de uma Pessoa como nome e idade e no comportamento fazAniversario, construa uma classe e crie alguns objetos. Faça com que alguns objetos façam aniversário e imprima os dados dos objetos criados.

03. Uma conta corrente possui um número, um saldo, um status que informa se ela é especial ou não, um limite e um conjunto de movimentações. Crie uma classe em Java com essas especificações e crie alguns objetos para testá-la.

04. Escreva uma classe que represente um país. Um país tem como atributos o seu nome, o nome da capital, sua dimensão em km2 e uma lista de países com os quais ele faz fronteira. Crie a classe em Java e forneça os seguintes construtores e métodos:

a) Construtor que inicialize o nome, capital e dimensão do país

b) Métodos de acesso (get e set) para obter as propriedades indicadas no item (a)

c) Um método que permita verificar se dois países são iguais. Dois países são iguais se tiverem o mesmo nome e a mesma capital. A assinatura desse método deve ser: public boolean equals(final Pais outro)

d) Um método que define quais outros países fazem fronteira (note que um país não pode fazer fronteira com ele mesmo)

e) Um método que retorne a lista de países que fazem fronteira

f) Um método que receba um outro país como parâmetro e retorne uma lista de vizinhos comuns aos dois países.

OBS: Este exercício é muito interessante e abrangente. Você vai precisar usar arrays de objetos para poder resolvê-lo. Encare como um bom desafio. Use a API do Java e suas outras fontes de informação para resolver o exercício corretamente.

REFLEXÃO

Vimos neste capítulo o que são classes e objetos e um pouquinho sobre interfaces gráficas. É importante que você observe o mundo real e perceba que tudo que nos rodeia pode ser abstraído em campos e estados, assim como as classes e objetos. Conseguir encontrar esses campos e comportamentos, criar classes genéricas e relacioná-las umas com as outras certamente fará de você um grande analista e programador em qualquer linguagem de programação orientada a objetos.

LEITURA

Procure na internet uma IDE chamada BlueJ. Trata-se de um programa que vai ajudá-lo a programar em Java com mais facilidade. No site deste programa você encontrará várias documentações a respeito de como usar o programa e outras explicações a respeito de classes e objetos. Apesar de ser em inglês, é uma leitura fácil e muito recomendada.

Existem alguns livros que ensinam Java por meio de jogos. É uma abordagem interessante e muito válida. A lista seguinte apresenta alguns títulos interessantes sobre o assunto:

Developing games in Java. David Brackeen, Bret Barker, Lawrence Vanhelsuwe. New Riders, 2003

Fundamental 2D game programming in Java. Timothy M. Wright. Cengage Learning PTR, 2014

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Bookman, 2002.

FLANAGAN, D. **Java o guia essencia**l. Bookman, 2006.

GARY, C.; HORSTMANN, C. Core **Java 2:** Fundamentos. Makron Books, 2001.

GARY, C.; HORSTMANN, C. Core **Java 2:** Recursos Avançados. Makron Books, 2001.

HUBBARD, J. R. **Programação com Java**. Bookman, 2006.

3

Herança e Associações de Classe

A orientação a objetos tem grande aceitação no mercado e nas empresas que desenvolvem ferramentas de software devido a mecanismos eficientes para abstrair o mundo real em pedaços de código.

A herança certamente é um desses mecanismos e por meio dela é possível desenvolver sistemas cada vez mais complexos e completos. Neste capítulo vamos estudar seus conceitos gerais e algumas de suas derivações.

Além disso vamos estudar a forma pela qual as classes se relacionam entre si por meio dos vários tipos de associações.

Bons estudos!

OBJETIVOS

Neste capítulo nosso objetivo é aprender os conceitos básicos de:

- Herança;
- Interfaces e classes abstratas;
- Encapsulamento e polimorfismo;
- Associações entre classes e objetos.

3.1 Introdução

Antes de entrarmos nos assuntos do capítulo, vamos adotar algumas convenções e apresentar um exemplo para posteriormente entendermos os conceitos propostos para este capítulo.

Podemos representar uma classe por meio de uma notação, um diagrama. Existe uma linguagem de modelagem chamada UML (Unified Modeling Language) cujo objetivo é usar diagramas para representar um sistema orientado a objetos. A UML serve para diagramar classes, objetos e várias outras situações encontradas na análise de sistemas. Ela serve para qualquer linguagem orientada a objetos e não é exclusiva do Java. A figura 3.1 mostra a classe Conta, que iremos usar nos nossos exemplos, e sua implementação em Java. Observe a figura atentamente e como é o código correspondente.

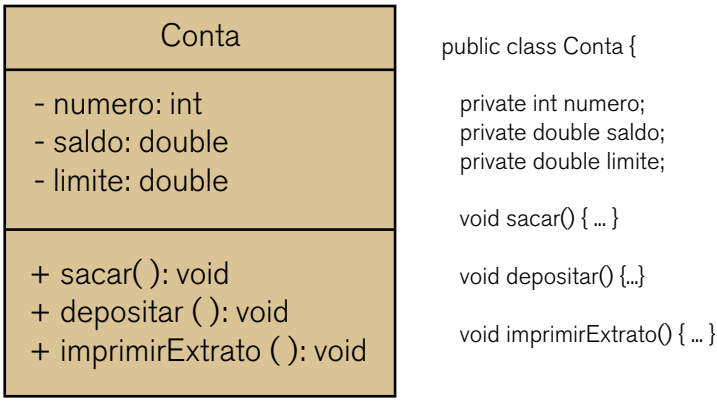


Figura 3.1 – Representação da classe Conta em UML e o código em Java correspondente.

Vamos explicar a figura:

O nome da classe é fácil descobrir: ela fica na parte superior do retângulo do diagrama.

Na próxima parte do diagrama ficam os campos, os atributos da classe: no caso da figura 3.2 temos 3 atributos:

- numero: int → corresponde a um atributo private do tipo int chamado numero
- saldo: double → um atributo private do tipo double chamado saldo
- limite: double → um atributo private do tipo double chamado limite

O sinal de subtração (–) significa que o elemento é private. O sinal de adição (+) significa que o elemento seguinte é public.

Na parte inferior do diagrama encontram-se os métodos. Temos no exemplo 3 métodos:

• + sacar() : void → corresponde a um método public chamado sacar sem retorno

• + depositar() : void → corresponde a um método public chamado depositar, sem retorno

• + imprimirExtrato(): void → um método public chamado imprimirExtrato, sem retorno

Uma vez que mostramos como podemos representar uma classe por meio de um diagrama, considere o diagrama de classes mostrado na figura 3.2:

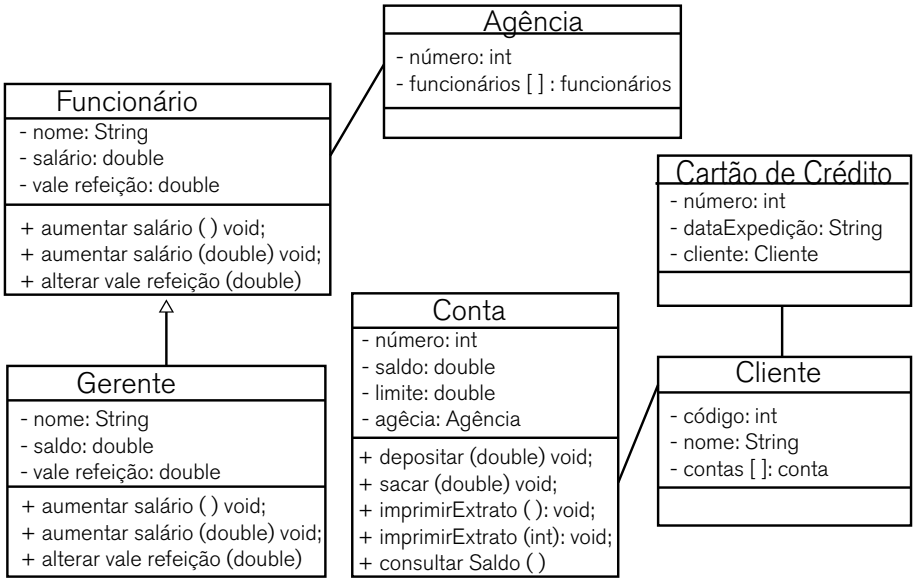


Figura 3.2 – Diagrama de classes completo.

O diagrama mostrado nos conta muitas coisas que refletem o mundo real na visão do analista que fez esta modelagem. Um cliente possui vários cartões de crédito, um cliente está associado a uma ou mais contas, uma conta a uma agência, a qual possui várias contas. Em uma agência trabalham vários

funcionários e um deles ou mais, é do tipo gerente. A especificação do diagrama de classes você conhecerá na disciplina de modelagem de sistemas usando UML.

Portanto, o diagrama mostra que existem classes que estão relacionadas de várias formas e cada uma dessas associações tem uma representação gráfica diferente que será discutida brevemente neste capítulo.

Vamos usar este diagrama com suas associações durante todo esse capítulo. O diagrama reflete um sistema bancário muito simples.

3.2 Atributos e métodos de classe

No sistema bancário da Figura 20 observe que temos uma classe específica para os funcionários do banco. E um de seus campos é o valeRefeicaoDiario. Este campo representa um valor fixo que é dado para os funcionários diariamente. Pense na codificação desta classe em Java. Cada funcionário seria um objeto da classe. Sendo assim, não faz muito sentido que cada objeto possua um valor de valeRefeicao diferente.

Mesmo que em uma atribuição como funcionário.setValeRefeicao(100.00), por exemplo, se tivéssemos 1000 objetos Funcionario não seria uma boa ideia ter que controlar esta taxa de cada funcionário. E se o valor subisse 10%? Teríamos que alterar objeto por objeto.

Sendo assim, é melhor criar um campo da classe que possa te ro seu valor fixo e igual em todas as instâncias da classe. Logo, aparece o conceito de atributo da classe e neste caso o valeRefeicaoDiario seria este atributo. Os atributos que não são atributos de classe são chamados então de atributos de instância.

Para que o atributo valeRefeicao seja considerado atributo da classe, usamos o modificador static:

```
static double valeRefeicao;
```

O acesso a este atributo é feito de uma maneira direta, sem precisar criar uma instância dessa classe para poder acessar e modificar o atributo.

```
    Funcionario.valeRefeicao = 25.50;
```

Pela maneira acima, todas as instâncias (objetos) que forem criados da classe Funcionario terão o mesmo valor neste atributo.

Já vimos que a recomendação para a visibilidade dos atributos é que eles sejam private. Portanto, o acesso ao atributo mostrado acima na prática não irá ocorrer. Então precisamos criar métodos para poder manipular esses atributos, da mesma forma que fazemos com os atributos de instâncias.

Por exemplo, se o banco quiser alterar o valor do vale refeição de seus funcionários é preciso criar um método para tal pois isto não depende dos dados de um funcionário só. O método para alterar o atributo está apresentado a seguir:

```
    static void alterarValeRefeicao (double taxa) {
       Funcionario.valeRefeicao = Funcionario.valeRefeicao + Funcionario .
valeRefeicao*taxa;
    }
```

Percebeu o static antecedendo o método? Logo, para tratar de atributos *static*, são necessários métodos static. E para chamar este método, usamos o código abaixo:

```
    Funcionario.alterarValeRefeicao(0.2);
```

3.3 Herança

A herança é um mecanismo fundamental para a orientação a objetos. Usamos este mecanismo diariamente na área de desenvolvimento e como usuários de internet, programas e outros softwares, temos contato constante com isso.

Uma das grandes vantagens do uso da herança é sem dúvida a reutilização de código. Como exemplo, vamos pensar na modelagem do banco da Figura 20. Um banco normalmente oferece serviços para os seus clientes como empréstimos, financiamentos, seguros e outros. Todo serviço, independente de

qual seja, tem características semelhantes: cliente, valor, data de vencimento, data de abertura, entre outras. O programador deste sistema naturalmente vai desenvolver uma classe para um serviço com esses atributos. Porém quando ele for escrever o código para os tipos de serviços especificamente como o empréstimo, o financiamento, ele teria que escrever novamente as classes com esses atributos, repetindo todo o trabalho que teve anteriormente. A herança evita esse retrabalho.

Observe os exemplos no código em Java. A classe serviço completa está mostrada abaixo:

```
public class Servico{
    private Cliente contratante;
    private Funcionario responsavel;
    private Date dataDeContratacao;

    //dados do empréstimo
    private double valor;
    private double taxa;

    //dados do seguro
    private Carro carro
    private double valorSeguroCarro;
    private double franquia;
}
```

Na aplicação, teremos que criar objetos Servico. Nesta criação os objetos serviços terão dados de empréstimo e seguro numa mesma instância. Isso não soa um tanto estranho? Pois um empréstimo deveria ter somente os dados de empréstimo, idem para um seguro! Baseado nisto, fica claro que devemos separar um serviço em cada classe.

O melhor é criar uma classe genérica, que contenha os dados comuns a todos os tipos de serviços e criar classes específicas. A classe genérica é chamada de superclasse (ou classe base, ou classe mãe) e suas dependentes chamadas de subclasses, classes derivadas ou classes-filha. Isto é feito de acordo com o diagrama em UML a seguir:

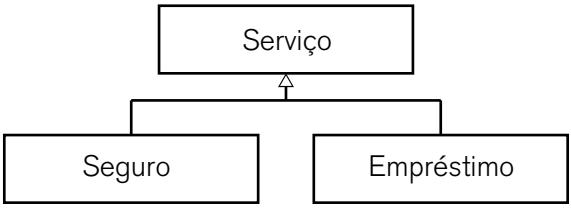


Figura 3.3 – Diagrama de classe representando herança.

O diagrama em UML é equivalente ao seguinte código em Java:

```
class Servico {
    private Cliente contratante;
    private Funcionario responsavel;
    private Date dataDeContratacao;
}

class Emprestimo extends Servico {
    private double valor ;
    private double taxa ;
}

class Seguro extends Servico {
    private Carro carro ;
    private double valorDoSeguroCarro ;
    private double franquia ;
}
```

O comando extends faz o vínculo de herança entre as subclasses (Emprestimo e Seguro) e a superclasse Servico. Na figura 3.3 ocorre o mesmo com Gerente e Funcionario. É importante entender que todo Emprestimo é um Servico, todo Gerente é um Funcionario, ou seja, toda subclasse antes de mais nada é uma parte da superclasse. Veja a figura 3.4.

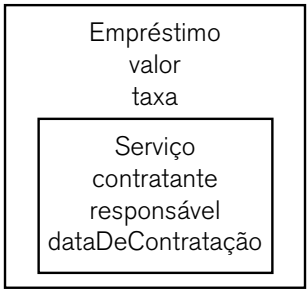


Figura 3.4 – Todo Emprestimo é um Serviço.

Ao criar um novo objeto Emprestimo, ele será também um Serviço:

```
Emprestimo e = new Emprestimo();
e.dataDeContratacao = “01/01/2015”;
       //dataDeContratação não está definida na classe
       //Empréstimo e sim na classe Serviço. O objeto
       //Empréstimo acessa o atributo da superclasse
       //(porque ele é público e faz parte da família)
```

3.3.1 Herança de métodos

É frequente o uso de métodos que são usados por todas as classes de uma determinada hierarquia. Podemos exemplificar com um atributo da classe Servico que represente uma taxa administrativa presente em todos os tipos de serviços. Este atributo será definido na classe Servico. Teremos também um método calcularTaxa() o qual é responsável por fazer os cálculos desta taxa. Todas as classes que herdam de Servico irão reaproveitar este método.

```
class Servico {
    // atributos
    public double calcularTaxa(){
        return 100;
    }
}
```

```
Emprestimo e = new Emprestimo();
Seguro s = new Seguro();
e.calcularTaxa(); //veja que tanto o objeto e o objeto s utilizam o método
s.calcularTaxa();
```

O exemplo será incrementado: vamos supor que a forma de cálculo da taxa administrativa do empréstimo é diferente da forma das demais classes. Poderíamos criar um novo método dentro da classe Emprestimo chamado calcularTaxaEmprestimo(). Isto já resolveria o problema mas ficaria um pouco estranho ter um método chamado calcularTaxa() e um calcularTaxaEmprestimo() disponível para o mesmo objeto. Além disso, com dois métodos os quais fazem praticamente a mesma coisa, algum programador poderia chamar o método errado. Podemos então reescrever o método calcularTaxa() e utilizar um método só, sobrecarregando-o (lembra deste conceito?). Desta forma:

```
class Emprestimo extends Servico {
   // ATRIBUTOS
   public double calculaTaxa() {
      return this.valor*0.1;
   }
}
```

O método rescrito calculaTaxa() da classe Emprestimo possui a mesma assinatura da sua superclasse. Os métodos das subclasses tem maior prioridade sobre os métodos das superclasses. Ou seja, se o método chamado existe na subclasse ele será chamado, caso contrário o método será procurado na superclasse.

Vamos continuar incrementando o exemplo. A taxa agora será dada por um valor fixo somada a um valor que depende do tipo serviço (seguro ou empréstimo). A tada do empréstimo terá como valor fixo 5 reais mais 10% do valor emprestado. O preço do seguro será 5 reais mais 5% do valor segurado.

```
class Emprestimo extends Servico {
 // ATRIBUTOS
   public double calculaTaxa() {
   return 5 + this.valor*0.1;   //10% do valor + R$5,00 fixos
 }
 }

 class Seguro extends Servico {
 // ATRIBUTOS
   public double calculaTaxa () {
       return 5 + this.carro.getTaxa()*0.05;  //5% do valor
}
 }
```

O problema agora é: se o valor fixo tiver que ser alterado, todas as subclasses de Servico deverão ser alteradas. No nosso exemplo é fácil pois são 2 classes somente. Mas e se fossem bem mais do que isso? Seria um grande retrabalho. Uma solução é criar um método na superclasse para essa tarefa e depois replicar este método nas subclasses. Assim:

```
class Servico {
   public double calculaTaxa(){
   return 5 ;
   }
}

class Emprestimo extends Servico {
// ATRIBUTOS
   public double calculaTaxa(){
       return super.calculaTaxa() + this.valor*0.1;
   }
 }
```

Usando o comando super é possível acessar o método equivalente (mesmo nome e assinatura) da superclasse.

Quando tratamos os métodos e herança não podemos deixar de lado os construtores das classes. Como sabemos, os construtores são métodos especiais que inicializam os atributos e sendo assim merecem uma atenção especial quando existe uma hierarquia de classes.

Observe a figura 4.3 e lembre que um Emprestimo é um tipo de Servico. Quando criamos um objeto empréstimo, o construtor da classe Emprestimo ou da classe Servico deve ser chamado. Se houver um construtor em cada uma dessas classes, o construtor da classe mais genérica é chamado antes (neste caso, o objeto empréstimo ao ser criado passa pelo construtor de Servico e depois Emprestimo). Quando não há construtores definidos, o compilador chamará o construtor sem argumentos da superclasse.

3.4 Polimorfismo

A palavra polimorfismo possui 2 partes: poli (muitas) morfismo (formas) – muitas formas. O polimorfismo é outro recurso que contribui para a reutilização de código. No exemplo do banco que estamos usando, falta um controle de ponto dos funcionários.

Todo funcionário deve "picar o cartão" no início e no fim do expediente. Portanto, se pensarmos em uma classe para essa finalidade, precisaremos de 2 métodos: um para a entrada e outro para a saída. Porém em uma empresa, os tipos de funcionários possuem formas de controle de ponto diferentes. Logo, se para 1 tipo de funcionário precisaríamos de 2 métodos, para N tipos de funcionários precisaríamos de N * 2 métodos! E que fazem a mesma coisa! Imagine o retrabalho!

A solução para este problema é analisar o mundo real e abstrai-lo na programação. No mundo real temos vários tipos de funcionários: Gerentes, Atendentes, Caixas, etc.. Logo, percebe-se que existe uma relação de herança: Um funcionário que deriva outros tipos, ou seja, o gerente é um funcionário, o atendente é um funcionário, o caixa é um funcionário, etc..

Na programação, a herança faz com que os objetos das subclasses criados sejam tratados como objetos da superclasse, ou seja, um objeto Gerente e um objeto Atendente são tratados como objetos da classe Funcionario.

```
class Gerente extends Funcionario    // todo gerente é um funcionário
...
Gerente g = new Gerente();      //criamos um objeto Gerente
Funcionario f = g;          //tratando um gerente como da classe Funcionario
```

Vamos dar uma olhada na classe que poderia implementar o nosso ponto eletrônico:

```
class PontoEletronico {
public void registraEntrada(Funcionario fun){
   //implementação do código
}

public void registraSaida(Funcionario fun){
   //implementação
}
 }
```

Como já tratamos anteriormente, não faz sentido criar uma classe de ponto eletrônico para cada tipo de funcionário. A classe acima é uma boa forma de implementar a solução pois os métodos registraEntrada() e registraSaida() recebem como parâmetro um objeto da classe Funcionario e sendo assim podem receber referências de objetos de qualquer subclasse de Funcionario. Temos aqui então um exemplo de polimorfismo.

Polimorfismo provém da biologia onde um organismo ou espécie pode ter várias formas ou estágios (sapo, borboleta, etc). No nosso caso, polimorfismo é quando subclasses de uma classe pode definir seu próprio e único comportamento e ainda compartilhar as mesmas funcionalidades que sua superclasse.

A vantagem de usar o polimorfismo no ponto eletrônico é que qualquer alteração que for necessária, só será feita em uma classe. Se novos tipos de funcionários forem criados, eles poderão acessar a classe PontoEletronico e seus métodos também.

3.5 Classes abstratas

Em um banco sabemos que existem vários tipos de contas: conta corrente, conta salário e conta poupança são alguns exemplos. Não é necessário explicar que então temos uma hierarquia de classes novamente na qual a superclasse é a Conta e suas subclasses são: ContaCorrente, ContaPoupanca e ContaSalario.

Na hora da criação de objetos, iremos sempre criar objetos de uma dessas três classes somente, não vamos precisar criar um objeto do tipo Conta porque ele será incompleto, ele não será uma conta com todas as suas propriedades e métodos, ela servirá apenas como uma base para poder criar outros tipos de contas. Logo, como criaremos somente objetos de um dos três tipos das subclasses, dizemos que essas classes são concretas e Conta é uma classe abstrata. Uma classe concreta serve então para criar instâncias de objetos e a classe abstrata, não. Veja abaixo como definir uma classe abstrata:

```
abstract class Conta {
   //atributos, construtores e métodos
}
```

Desta forma não será mais possível criar um objeto Conta e a linha abaixo representará um erro ao compilar:

```
Conta con = new Conta();    // ERRO!! Conta é abstract, não pode criar objetos
```

3.6 Métodos abstratos

É interessante que toda conta do nosso banco possa ter um método para imprimir o extrato dos seus lançamentos bancários. Você pode pensar que é um método a ser implementado na classe Conta e assim desta forma, todas as outras subclasses poderiam usar este método para esta finalidade. Mas não esqueça que cada tipo de conta pode ter um extrato específico e assim a implementação deste método passaria a ser em cada subclasse, o que é mais natural.

O problema é que cada subclasse pode implementar de uma maneira diferente o método da impressão dos extratos e é sempre importante manter um padrão no desenvolvimento de software. Como a classe base será a classe Conta, ela poderia conter uma forma de padronizar o método de impressão de extrato e assim orientar que suas subclasses implementem o método de uma maneira uniforme.

Para garantir que cada subclasse (concreta) que provém direta ou indiretamente de uma superclasse tenha uma implementação de método para imprimir os extratos, inclusive com uma mesma assinatura (e manter o padrão comentado), usamos o conceito de métodos abstratos.

```
abstract class Conta {
   //atributos, construtores e métodos
   public abstract void imprimirExtrato(Date dataInicial, Date dataFinal){
   } // não há implementação aqui
}

class ContaCorrente extends Conta {
   //atributos, construtores, métodos
   public abstract void imprimirExtrato(Date dataInicial, Date dataFinal){
      // aqui vai a implementação do método
   }
}
```

Perceba que na classe Conta o método imprimirExtrato() não possui código de implementação. Isto só será feito nas subclasses que usarem este método. Nestas subclasses o método deverá ter o mesmo nome e a mesma assinatura que o método da superclasse e além disso, ser obrigatoriamente implementado senão ocorrerá um erro na compilação.

3.7 Interfaces

Em muitas situações da engenharia de software é importante que os desenvolvedores entrem em acordo com um "contrato" o qual mostra como o software irá interagir. Cada grupo de programadores deverá ser capaz de escrever o seu código sem conhecimento de como o código do outro grupo foi desenvolvido. Estes contratos mencionados, estes acordos, são chamados de interfaces.

No exemplo do banco, podemos definir uma interface Conta para padronizar as assinaturas dos métodos oferecidos pelos objetos que representam as contas do banco.

```
interface Conta{
   void depositar(double quantia);
   void sacar(double quantia);
}
```

Os métodos das interfaces não possuem corpo nem implementação. Se uma interface funciona como um contrato, os métodos serão implementados obrigatoriamente nas classes concretas que "assinarem" este contrato.

```
class ContaSalario implements Conta{
   //atributos, construtores

   //implementação dos métodos da interface
   public void depositar(double quantia){
      ...
   }

   public void sacar(double quantia){
      ...
   }
}
```

A maior vantagem de usar interfaces é padronizar as assinaturas dos métodos que compõe a(s) interface(s) de um sistema. Outra vantagem é garantir que as classes concretas implementem o que foi estabelecido na interface.

CONEXÃO

Um padrão de projeto é como uma especificação de uma determinada solução para um problema específico no software. Este conceito será melhor explicado e desenvolvido em outras

disciplinas. Mas para antecipar, veja os padrões J2EE da Oracle: eles estão "recheados" de interfaces. Um padrão muito útil e usado é o Data Access Object (DAO). Veja no link: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

As interfaces são usadas em Java para prover um mecanismo chamado herança múltipla. A herança múltipla permite que uma subclasse pode ser derivada de duas superclasses. A rigor, isto não existe em Java mas existe em outras linguagens porém existe uma discussão muito grande, inacabada e inconclusiva em torno de herança, interfaces e herança múltipla.

Vamos usar as duas hierarquias abaixo para exemplificar:

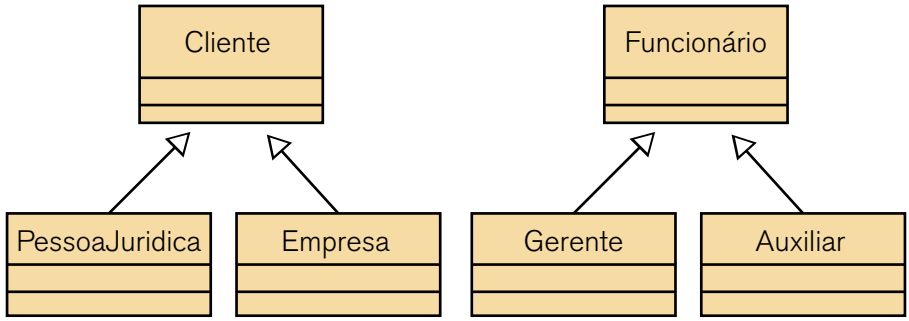


Figura 3.5 – Duas hierarquias independentes. I23

O banco pode ter uma classe para autenticar usuários no sistema. Os usuários podem ser funcionários do banco ou empresas externas mas neste caso, como implementar um método de autenticação que aceite tanto gerentes (que é de uma hierarquia) e empresas (de outra hierarquia)? Perceba que não há como usar polimorfismo neste caso e para existir, devemos de alguma forma juntar as duas hierarquias o que seria incorreto do ponto de vista natural (clientes e funcionários são entidades, coisas diferentes). Usando uma interface temos uma possibilidade:

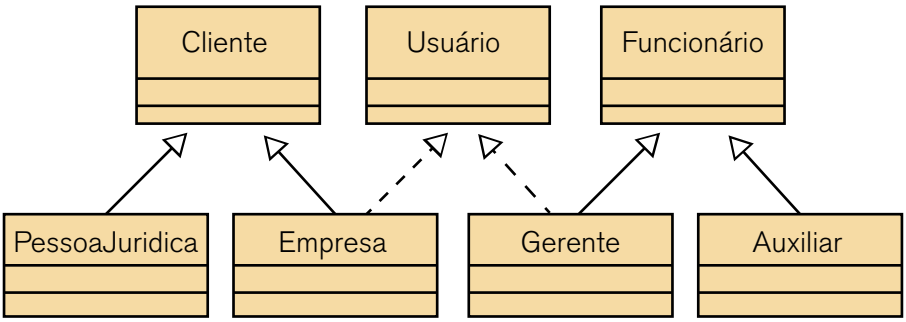


Figura 3.6 – Interface permitindo herança múltipla.

Desta forma podemos criar um método que autentica o Usuario:

```
class Login{
   public boolean autenticar(Usuario u) {
      // implementação
   }
}
```

3.8 Encapsulamento e associações de classe

No mundo real os objetos se relacionam. Um professor se relaciona com um aluno por meio de uma aula, um produto está presente em uma nota fiscal, um cliente pode ser pessoa física e jurídica, vários departamentos compõem uma empresa e assim por diante.

Um relacionamento na orientação a objetos é chamado de associação. A associação ocorre entre objetos da mesma classe ou de classes diferentes. No diagrama da figura 3.2 temos alguns exemplos de associações. Uma delas está representada na figura 3.7:

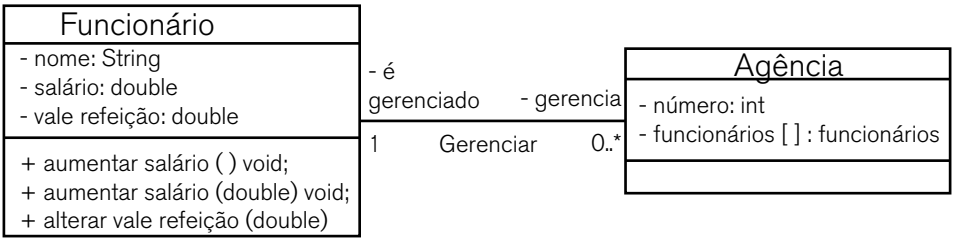


Figura 3.7 – Associação entre classes diferentes.

A Figura 25 possui várias informações:

• A associação possui um nome: "Gerenciar", que expressa a ação entre as classes

• O nome possui uma seta que mostra a direção na qual o nome deve ser lido: "Funcionário gerencia Agencia"

• A associação possui multiplicidade, ou seja, possui uma forma de mostrar o número de objetos envolvidos no relacionamento. No caso da Figura 25, "1 Funcionário gerencia 0 (ou não gerencia) ou muitas (*) Agências"

As associações são mecanismos que permitem aos objetos se comunicarem e descrevem as diferentes conexões entre as classes. As associações podem ter um tipo de regra a qual especifica o objetivo da associação e pode ser uni ou bidirecional, indicando se as duas classes da associação podem mandar mensagens uma para a outra ou se apenas uma delas sabe sobre a outra. Cada ponta da associação também possui um valor de multiplicidade que dita como vários objetos de um lado da associação podem se relacionar com o outro lado.

3.8.1 Agregação

A agregação é um tipo de associação. Nesta associação, uma das classes participantes do relacionamento é uma parte ou está contida em outra classe. Normalmente para identificar uma agregação usamos as palavras "consiste em", "contém", "é parte de". As classes que participam do relacionamento não estão no mesmo nível mas possuem um relacionamento "todo-parte".

Na agregação, a classe que possui a regra do "todo" é composta (tem) de outras classes, que possuem a regra das "partes". Só pode ocorrer agregação entre duas classes.

Podemos dar algumas dicas para identificar possíveis associações de agregação:

- Quando ocorrer uma relação física todo-parte entre duas classes
- Alguns atributos da classe "todo" se propagam para a classe "parte"
- Há uma dependência no tempo de vida: quando uma classe "todo" ou "parte" é criada ou destruída, a outra também é.

3.8.2 Composição

A composição é outra forma de associação. Também existe um relacionamento "todo-parte" e é mais restritiva que a agregação, logo, mais forte também. Isso significa que as partes envolvidas devem estar relacionadas a um único objeto "todo" ou seja, um objeto da classe "parte" pertence somente a um objeto da classe "todo", além disso, o tempo de vida dos objetos da classe "parte" coincidem com o tempo de vida da classe "todo".

A figura 3.8 mostra um bom exemplo para entender a composição. A classe "todo" Empresa contém uma "parte" Divisão e esta por sua vez (agora ela é que é a "todo") contém uma "parte" Departamento. A existência do objeto "parte" não tem sentido sem o seu "todo" e vice-versa.

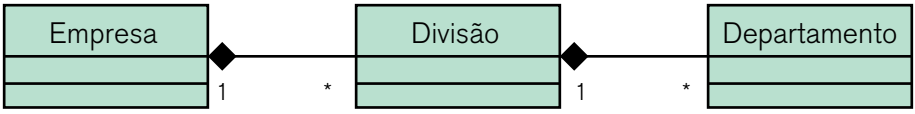


Figura 3.8 – Composição.

Resumindo, a agregação é uma forma de associação na qual o objeto composto apenas usa ou tem conhecimento da existência do(s) objeto(s) componente(s). Os objetos componentes podem existir sem o agregado e vice-versa.

Na composição, a associação define que o objeto composto é responsável pela existência dos componentes. E neste caso, o componente não tem sentido fora da composição.

Mais alguns exemplos finais:

• Composição: uma conta corrente é formada por várias transações de crédito e de débito
• Agregação: um cadastro de clientes é formado por vários clientes
• Agregação: um cliente tem uma conta corrente
• Composição: um documento possui um conjunto de parágrafos
• Composição: uma turma é um conjunto de alunos.

Visto estes dois tipos de associações, vamos tratar um pouco da programação. Em Java, não há uma forma declarativa para implementar agregações nem associações. Apenas são criadas associações unidirecionais em Java por meio de referências, por exemplo: uma conta tem um atributo cliente, ou seja, uma referência de conta para cliente, mas não de cliente para conta:

```
public class Conta{
   private int código;
   private Cliente cliente;    //associação unidirecional!
}
```

Veja a figura 3.9, vamos mostrar como fica o código deste diagrama. Observe que em UML a agregação é simbolizada com um losango preenchido e a composição é simbolizada com um losango vazado.

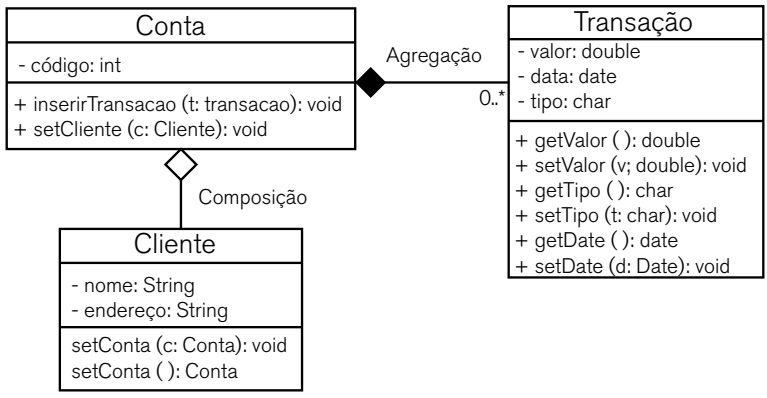


Figura 3.9 – Notação UML para agregação e composição.

```
class Conta{
   private int código;
   private Cliente cliente;           //apenas 1 referência para Cliente
   private Transacao[] transação;     //várias referências para Transação
   private int qtdTransacoes;
   ...
   public setCliente(Cliente c){
      this.cliente = c;
   }

   public void inserirTransacao(Transacao t){   //forma simples de inserir
Transação
      transação[qtdTransacoes++]=t;
   }
}

class Cliente{
   private String nome;
   private Conta conta;
   private String endereço;
   ...
   public void setConta(Conta c){     //como o relacionamento entre Cliente
e Conta é
      conta = c;                      // bidirecional, cada uma das classes
possui um
   }         // atributo que é do tipo da outra
}

class Transacao {
   private double valor;
   private char tipo;
   priavte Date data;
   ...
}
```

Em Java é estabelecida uma associação unidirecional entre objetos definindo, na classe do objeto de origem, uma propriedade cujo tipo é a classe do objeto destino.

Lembre-se que ao instanciar um objeto da classe de onde parte a associação você não estará criando uma instância do objeto destinatário da associação. Ou seja, ao criar um objeto da classe Conta, você não criará um objeto da classe Cliente ou vários objetos da classe Transação. Para isso, métodos em ambas as classes deverão fazer a agregação e composição em tempo de execução da aplicação. Veja o código abaixo, o qual mostra esse processo em alguma parte da aplicação:

```
    public static void main(String[] args) {
        Conta con = new Conta();
        Cliente cli = new Cliente();
        Transacao credito = new Transacao();

        con.setCodigo(100);
        con.setCliente(clie);           // nessas duas linhas
        cli.setConta(con);              //    temos a agregação entre Cliente e
Conta
        credito = new Transacao();
        credito.setValor(10000000.0);
        credito.setTipo('C');
        credito.setData(new Date());
        con.addTransacao(credito);      // aqui temos a composição
    }
```

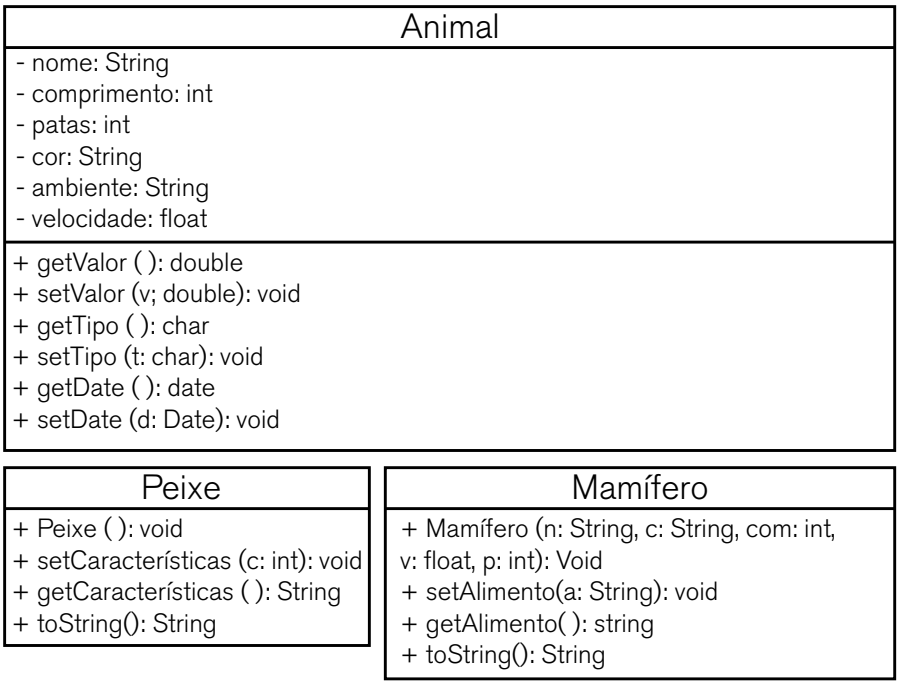
ATIVIDADES

01. Um animal possui um nome, comprimento, cor, ambiente, velocidade e número de patas (4 é o padrão).

Um peixe é um animal mas não tem patas. O seu ambiente é o mar, a cor é cinza (padrão) e além disso um peixe tem barbatanas e cauda.

Um mamífero é um animal que vive na terra por padrão (ambiente). Um urso é um mamífero que vive na terra, é marrom, e o seu alimento preferido é o mel.

Faça em Java as classes Animal, Peixe e Mamífero de acordo com o diagrama UML a seguir:



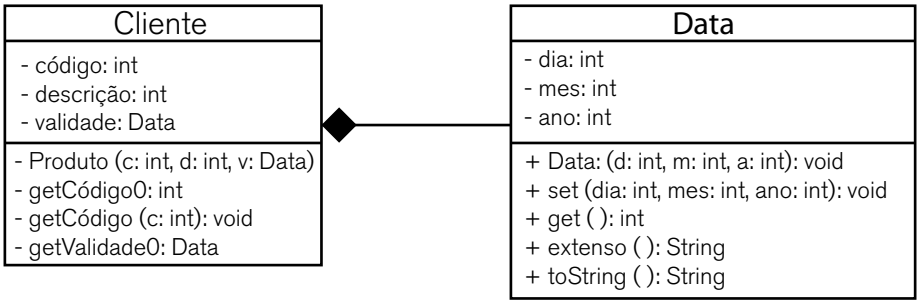
Crie um zoológico (programa teste) contendo os seguintes animais: um camelo, um tubarão e um urso pardo.

O método toString de cada classe deve retornar os dados da seguinte forma:

```
camelo: 150cm, 4 patas, amarelo, terra, 2m/s
tubarão: 300cm, 0 patas, cinza, mar, 1.5m/s, barbatana e cauda
ursopardo: 180cm, 4 patas, marrom, terra, 0.5m/s, mel
```

Depois faça um outro teste para que possam ser incluídos quantos animais o usuário quiser, porém no máximo 10 animais de cada tipo (use vetores de objetos).

02. Implemente o diagrama a seguir em Java:



REFLEXÃO

Esperamos que esses conceitos que estudamos até agora sirvam para você refletir nas grandes vantagens que a orientação a objetos pode trazer para as atividades de desenvolvimento. Esperamos que seja percebido que a orientação a objetos torne mais rápida as atividades de programação e manutenção de sistemas. Além disso, a orientação a objetos tem um caráter unificador ou seja, trata todas as etapas do desenvolvimento sob uma única abordagem destacando a reusabilidade de código, a escalabilidade das aplicações, a manutenibilidade entre outras características.

LEITURA

Não é só a linguagem Java que é orientada a objetos. Existem várias como já citamos. Uma delas é a Smalltalk, uma das primeiras linguagens orientadas a objetos que apareceu. Ler sobre Smalltalkvai ajuda-lo a entender muitos conceitos que estamos tratando aqui. A seguir apresentamos algumas boas referências desta linguagem:

Dynamic Web Development with Seaside. Stephane Ducasse, Lukas Renggli, David C. Shaffer and Rick Zaccone. Square Bracket Associates, 2009.

Smalltalk design pattern companion book drafts. Sherman Alpert, Kyle Brown, and Bobby Woolf. Addison-Wesley, 978-02011846241998.

Smalltalk by Example: the Developer's GuideAlex Sharp, McGraw Hill Text; ISBN: 0079130364, 1997.

Smalltalk With Style by Edward Klimas, Suzanne Skublics and David A. Thomas. ISBN: 0-13-165549-3, Publisher: Prentice Hall, Copyright: 1996

Todos estes livros estão disponíveis online.

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Bookman, 2002.

FLANAGAN, D. **Java o guia essencial**. Bookman, 2006.

GARY, C.; HORSTMANN, C. Core **Java 2:** Fundamentos. Makron Books, 2001.

GARY, C.; HORSTMANN, C. Core **Java 2:** Recursos Avançados. Makron Books, 2001.

HUBBARD, J. R. **Programação com Java**. Bookman, 2006.

4

Interfaces Gráficas

Todo mundo gosta de interagir com um programa de uma maneira agradável e objetiva. Com a explosão da internet, cada dia que passa exigimos que os programas tenham uma aparência e um comportamento parecido uns com os outros para aumentar a nossa produtividade e experiência no uso da informática. Com a popularização dos celulares isso aumenta ainda mais.

Porém ainda existe uma grande demanda de programas que rodam exclusivamente em computadores *desktop*, em redes e que não precisam estar na internet ou ter os mesmos recursos que a *internet* oferece. Muitos programas que usamos no dia a dia são assim como os editores de texto, planilhas eletrônicas, *softwares* para apresentações, jogos, entre outros.

Este capítulo vai mostrar os principais componentes para gerar programas para esta categoria de *software*. Bons estudos!

OBJETIVOS

O objetivo principal deste capítulo é introduzi-lo nos principais componentes básicos de interface gráfica usando a linguagem Java e o *Swing*.

4.1 Introdução

No Capítulo 2 começamos o estudo sobre interfaces gráficas. Vimos que existem dois pacotes muito utilizados na programação de programas para *desktop* chamados *java.awt* e *java.swing*. Atualmente a tecnologia suportada pela Oracle é a *JavaFX*.

Naquele capítulo fizemos um exemplo simples de um formulário para podermos conhecer o mecanismo de desenvolvimento. Agora é hora de aprofundarmos o estudo em outros componentes que poderão dar nova cara às suas aplicações.

4.2 Menus e barras de ferramentas

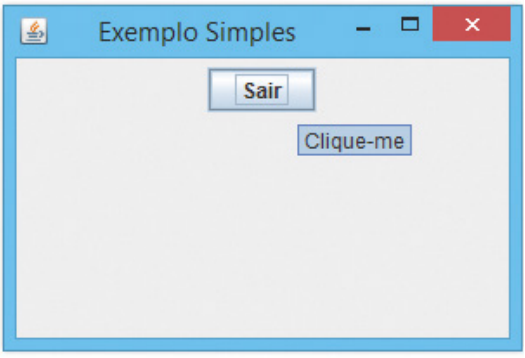
Sabemos que um menu é um grupo de comandos localizados em uma barra de menus (*menubar*). Uma barra de ferramentas (*toolbar*) tem botões com comandos comuns na aplicação.

Para implementar uma *menubar* em Java, nós usaremos 3 objetos das classes *JMenuBar*, *Jmenu* e *JMenuItem*. Veja a implementação e as explicações no código a seguir. A Figura 4.1 mostra à esquerda como fica a execução do programa.

```
public class Exemplo extends JFrame {
    public Exemplo() {
        //cria a barra de menu, onde o menu ficará hospedado
        JMenuBar menubar = new JMenuBar();
        //cria o menu Arquivo
        JMenu arquivo = new JMenu("Arquivo");
        //cria uma tecla de atalho para o menu arquivo: ALT+a
        arquivo.setMnemonic(KeyEvent.VK_A);
        //lê uma imagem
        ImageIcon icon = new ImageIcon("exit.png");
        //cria um item de menu e associa a imagem lida ao item de menu Sair
        JMenuItem itemSair = new JMenuItem("Sair", icon);
```

```
    //cria uma tecla de atalho para o item Sair: ALT+r
           itemSair.setMnemonic(KeyEvent.VK_R);
           itemSair.setToolTipText("Sai do programa");
    //cria o evento
           itemSair.addActionListener(new ActionListener() {
              @Override
              public void actionPerformed(ActionEvent event) {
                 System.exit(0);
              }
           });
    //coloca o item Sair no menu Arquivo, o menu Arquivo na barra de menu e o menu
no Frame
           arquivo.add(itemSair);
           menubar.add(arquivo);
           setJMenuBar(menubar);
    //configura o frame
           setTitle("Menu Simples");
           setSize(300, 200);
           setLocationRelativeTo(null);
           setDefaultCloseOperation(EXIT_ON_CLOSE);
       }
    // aqui ficaria o método main, semelhante ao código anterior
    }
```

Os submenus também são muito úteis em qualquer aplicação, veja a figura 4.1b. É uma outra forma de agrupar comandos semelhantes em um item de menu. Nos menus existem também os separadores os quais são apenas uma linha para separar os comandos.



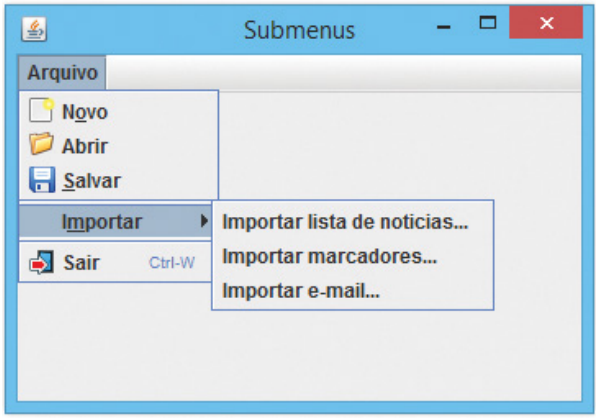


Figura 4.1 – Implementação do código sem e com submenu.

Para criar um submenu é o mesmo processo de criação de um menu normal. Para criar o submenu *Importar* da figura 4.1 é preciso o seguinte código:

```
    //criamos um menu chamado imp, o qual conterá 3 itens em um submenu
    JMenu imp = new JMenu("Importar");

    //cria um item de menu como anteriormente
    JMenuItem noticias = new JMenuItem("Importar lista de noticias...");

    //e depois inserimos o item de menu no menu desejado
    imp.add(noticias);

    //para inserir um separador, basta usar o método addSeparator() na posição
desejada
    arquivo.add(arquivoNovo);
    arquivo.add(arquivoAbrir);
    arquivo.add(arquivoSalvar);
    arquivo.addSeparator();
    arquivo.add(imp);
    arquivo.addSeparator();
    arquivo.add(arquivoSair);
```

4.2.1 Menu com checkbox

Um *JCheckBoxMenuItem* é um item de menu que pode ser selecionado ou não. Quando está selecionado existe uma marcação para tal. Este tipo de item de menu funciona como os outros vistos até agora, podendo ou não ter uma figura associada a ele. A figura 4.2 mostra um exemplo de um *frame* com um item do tipo *checkbox*

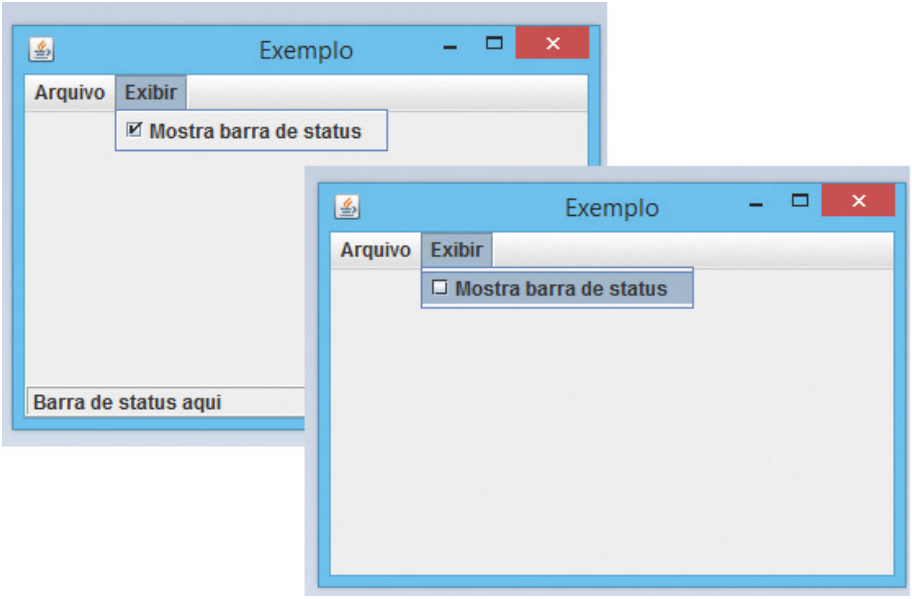


Figura 4.2 – Menu com CheckBox.

Para criar este item de menu e a barra de status usamos o código a seguir:

```
JMenu view = new JMenu("Exibir");
JCheckBoxMenuItem menuCheck = new JCheckBoxMenuItem("Mostra barra de status");
menuCheck.setState(true);   //true faz com que o checkbox esteja marcado
view.add(menuCheck);
//Para criar e posicionar a barra de status, usamos os comandos abaixo
statusbar = new JLabel(" Barra de status aqui");
statusbar.setBorder(BorderFactory.createEtchedBorder(EtchedBorder.RAISED));
add(statusbar, BorderLayout.SOUTH);
```

O evento que mostra e esconde a barra de status será visto no Capítulo 5 onde iremos tratar exclusivamente dos eventos nas interfaces gráficas.

4.2.2 Menus popup

Outro tipo de menu que usamos muito é o menu popup. Em Java existe a classe *JPopupMenu* que implementa este tipo de menu o qual algumas vezes é chamado de menu de contexto e aparece quando clicamos com o botão direito do mouse.

Basicamente, a forma de criação e vinculação de itens de menu ao menu *popup* é parecida com os menus que já estudamos. Observe:

```
//declaração e criação do popup
JPopupMenu pmenu;
pmenu =new JPopupMenu();

//criação do item de menu, da mesma forma como já estudamos antes
JMenuItem maxitem = new JMenuItem("Maximizar");

//vinculação do item de menu ao menu popup
pmenu.add(maxitem);
```

4.2.3 Barras de ferramentas (toolbar)

Usando o *Swing* podemos usar a classe *JToolBar* para criar barras de ferramentas conforme o código abaixo:

```
//criamos a barra
JToolBar toolbar1 = new JToolBar();

//criamos um painel dentro de um frame para receber a barra
JPanel panel = new JPanel();

//definimos um layout vertical no painel
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

```
//alinhamos a barra à esquerda
toolbar1.setAlignmentX(0);

//colocamos a barra no painel e o alinhamos na posição superior do frame
panel.add(toolbar1);
add(panel, BorderLayout.NORTH);
```

4.3 Gerenciadores de layout

O *Swing* tem como vimos dois tipos principais de componentes: os containers e os componentes-filho. Os *containers* agrupam os componentes filho em layouts apropriados, os quais podem ser criados usando os gerenciadores de *layout*.

Basicamente, temos 3 tipos de organização dos componentes em um *container*:

• Posicionamento absoluto: neste caso o layout de um *container* é configurado como *null* e assim o programador pode especificar a posição absoluta do componente no próprio código em relação ao canto superior esquerdo do *container*

• Com gerenciador de layout: neste caso, é mais simples e rápido criar e usar uma posição porém perde-se o controle e o tamanho do componente. A grande vantagem de se usar um gerenciador de layout é que a aplicação que o uso pode ser mais portável que outras que não usam. Lembre-se que uma das grandes vantagens do uso do Java é exatamente sua portabilidade.

• Programação visual: neste caso o posicionamento dos componentes é feito usando o recurso arrastar e soltar dentro do *container*

É importante diferenciar o que é uma linguagem de programação e um Ambiente de Desenvolvimento Integrado (IDE – *Integrated Development Environment*). Você já deve ter conhecido alguém que "programa em Delphi" ou "programa em Visual Studio". Tanto o Delphi quanto o Visual Studio são excelentes IDEs para o desenvolvimento em *Object Pascal* no Delphi, e várias outras linguagens que o Visual Studio suporta. Essas 2 IDEs se destacam pela grande facilidade de arrastar e soltar componentes. As IDEs que trabalham com Java em geral não possuem essa facilidade. Porém, por meio de plug-ins é possível arrastar e soltar os componentes como é feito no Delphi ou Visual Studio, auxiliando assim o gerenciamento do layout.

Um gerenciador de layout é um objeto que implementa a interface *LayoutManager* e determina o tamanho e posição dos componentes dentro de um container. Mesmo que os componentes possam determinar seu tamanho, alinhamento e outras propriedades, um gerenciador de layout de um *container* é quem "diz" o tamanho e a posição dentro dele.

4.3.1 Containers sem gerenciador de layout

É possível ter um *container* sem um gerenciador de layout. Existem algumas situações onde não é necessário se ter um. Mas em outras muitas situações, principalmente devido a portabilidade das aplicações, é recomendável que se use. Sem o gerenciador de layout, nós posicionamos os componentes usando valores absolutos. Por exemplo:

```
public final void teste() {
    setLayout(null);

    JButton ok = new JButton("OK");
    ok.setBounds(50, 50, 80, 25);  //posicionamento absoluto

    JButton fechar = new JButton("Close");
    fechar.setBounds(150, 50, 80, 25);  //posicionamento absoluto

    add(ok);
    add(fechar);

    setTitle("Posicionamento absoluto ");
    setSize(300, 250);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}
```

4.3.2 Tipos de gerenciadores de layout

Existem várias classes *Swing* e AWT que fornecem gerenciadores de layout para vários propósitos:

4.3.2.1 FlowLayout

O *FlowLayout* é o gerenciador de layout padrão para todos os *JPanel*. Ele simplesmente posiciona os componentes em uma linha simples, começando uma nova linha se o *container* não for largo o suficiente. Veja o exemplo a seguir e a figura 4.3 resultante:

```
public class ExemploFlowLayout extends JFrame {
   private FlowLayout layout;
   private Container container;

   public ExemploFlowLayout() {
      layout = new FlowLayout();
      container = getContentPane();
      setLayout(layout);
   JPanel panel = new JPanel();
      JTextArea area = new JTextArea("Text Area");
      area.setPreferredSize(new Dimension(100, 100));
      JButton button = new JButton("Botao");
      panel.add(button);
      JTree tree = new JTree();
      panel.add(tree);
      panel.add(area);
      add(panel);
      pack();
      setTitle("Exemplo de Flow Layout");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLocationRelativeTo(null);
   }

   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            ExemploFlowLayout ex = new ExemploFlowLayout();
            ex.setVisible(true);
         }
      });
   }
}
```

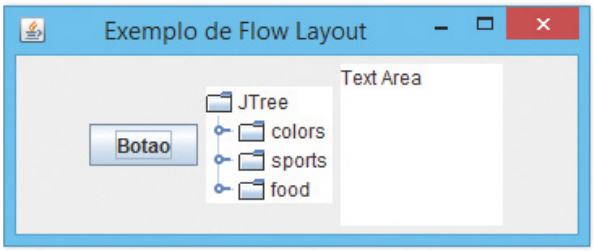


Figura 4.3 – FlowLayout.

Perceba que as três linhas definem o gerenciador de layout a ser usado no container onde o layout será aplicado.

```
layout = new FlowLayout();
container = getContentPane();
setLayout(layout);
```

Os outros gerenciadores de layout seguem o mesmo comportamento de criação e uso porém, por se tratarem de classes diferentes, a forma de instanciar um gerenciador de layout pode variar e depende do construtor da classe do gerenciador de layout.

4.3.2.2 *BorderLayout*

O gerenciador *BorderLayout* coloca os componentes em até 5 áreas: *top* (superior), *bottom* (inferior), *left* (esquerda), *right* (direita) e *center* (centro). Todo espaço extra é colocado na área central. As *toolbars* devem ser criadas dentro de um container com o *BorderLayout*.

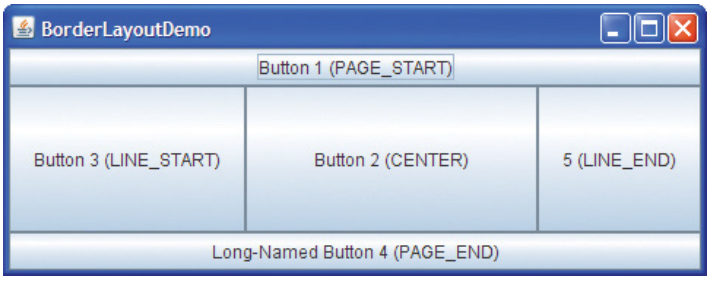


Figura 4.4 – BorderLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.3 BoxLayout

O *BoxLayout* coloca os componentes em uma coluna ou linha simples. Ele respeita o componente de maior tamanho e permite que os outros sejam alinhados.

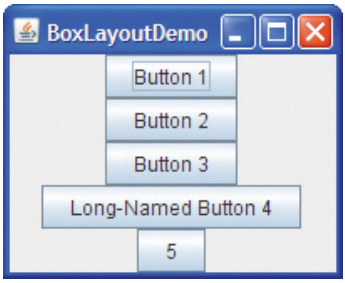


Figura 4.5 – BoxLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.4 *CardLayout*

Este gerenciador permite que seja implementada uma área que contém componentes diferentes em vezes diferentes. Normalmente se usa este gerenciador com um *combobox* no qual o estado do combo determina a aparência da janela, veja a Figura 33: conforme o usuário seleciona um valor no c*ombo*, o painel inferior muda.

Figura 4.6 – CardLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.5 GridBagLayout

Este gerenciador é mais flexível e sofisticado. Ele alinha os componentes colocando-os dentro de uma tabela permitindo que os componentes tenham um comprimento maior do que uma célula. As linhas podem ter alturas diferentes e as colunas, larguras diferentes.

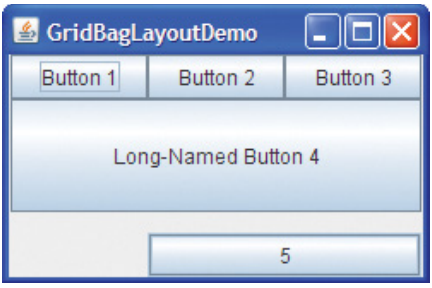


Figura 4.7 – GridBagLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.6 GridLayout

Este componente usa uma tabela com células de tamanho igual.

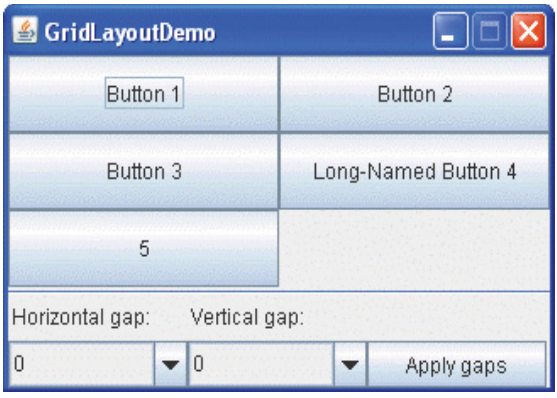


Figura 4.8 – GridLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.7 GroupLayout

Este gerenciador trabalha com layouts vertical e horizontal separadamente. O layout é definido por cada dimensão independentemente. Consequentemente, cada componente precisa ser definido duas vezes no layout.

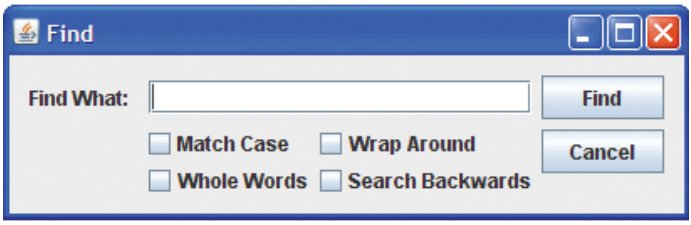


Figura 4.9 – GroupLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.2.8 pringLayout

Assim como o *GroupLayout*, este gerenciador é muito usado em *IDEs* que possuem *plug-ins* para interface gráfica. Este gerenciador permite que seja especificada relações entre os limites dos componentes sobre seus controles.

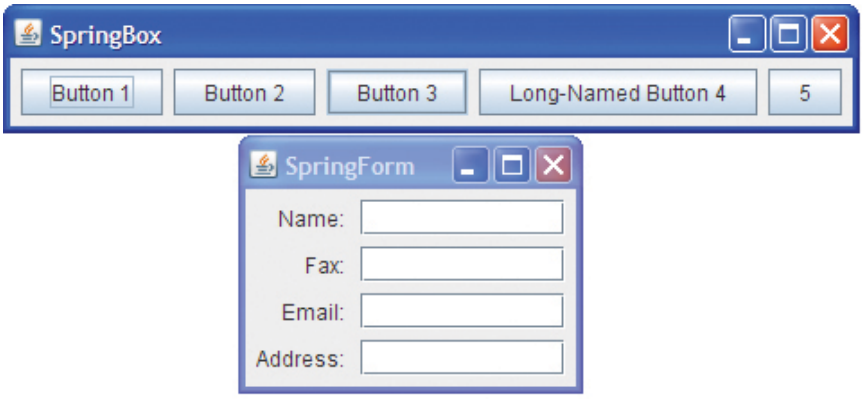


Figura 4.10 – SpringLayout. Fonte: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

4.3.3 Outros componentes

O *Swing* possui outros componentes muito úteis para os seus programas. Não é nosso escopo cobrir todos os componentes mas mostrar os principais e as formas de criar e usa-los. Você vai perceber que eles possuem uma forma semelhante de definição, criação e uso. Na verdade é uma aplicação dos conceitos de orientação a objetos que veremos no próximo capítulo.

Observe a Figura 4.11. Ela mostra uma tela simples contendo vários componentes *Swing*. Os componentes mostrados nesta figura são bastante utilizados na maioria das aplicações que conhecemos.

Logo após a figura mostramos o código em Java para gerar a tela. Vamos explicar vários componentes usando o código portanto, preste atenção!

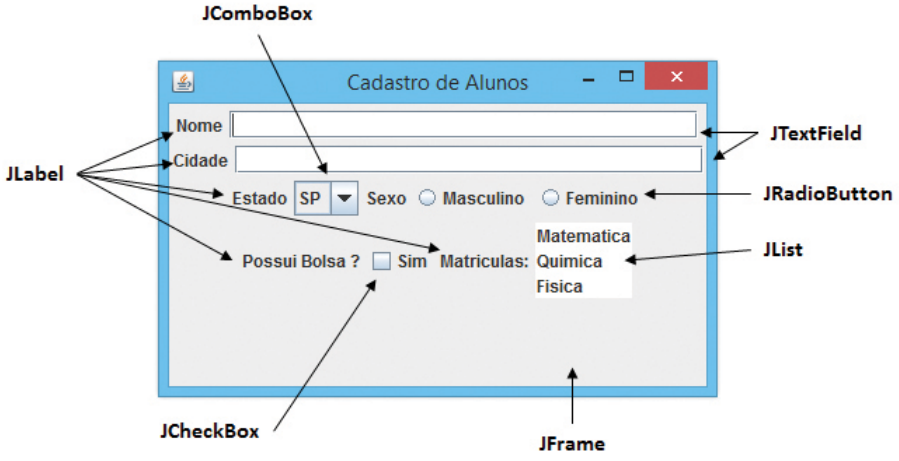


Figura 4.11 – Tela simples de cadastro de alunos com vários componentes Swing

```
1  public class Cadastro extends JFrame{
2      private JLabel labNome,
3              labSexo,
4              labCidade,
5              labEstado,
6              labBolsa,
7              labDisciplinas;
8
9      private JTextField texNome,
10             texCidade;
11
12     private JCheckBox cheBolsa;
13     private JRadioButton radMasculino,
14                radFeminino;
15     private ButtonGroup grupo;
16     private JComboBox comEstado;
```

```
17    private Container c;
18    private JList lista;
19    private DefaultListModel modeloLista;
20
21    public Cadastro(){
22       super("Cadastro de Alunos");
23       labNome        = new JLabel("Nome");
24       labSexo        = new JLabel("Sexo");
25       labCidade      = new JLabel("Cidade");
26       labEstado      = new JLabel("Estado");
27       labBolsa       = new JLabel("Possui Bolsa ?");
28       labDisciplinas = new JLabel("Matriculas:");
29
30       String[] estados = { "SP", "RJ", "MG", "ES" };
31       comEstado        = new JComboBox(estados);
32
33       cheBolsa = new JCheckBox("Sim");
34
35       texNome   = new JTextField(30);
36       texCidade = new JTextField(30);
37
38       grupo        = new ButtonGroup();
39       radMasculino = new JRadioButton("Masculino");
40       radFeminino  = new JRadioButton("Feminino");
41       grupo.add(radMasculino);
42       grupo.add(radFeminino);
43
44       c = getContentPane();
45       c.setLayout(new FlowLayout());
46
47       modeloLista          = new DefaultListModel();
48       String[] disciplinas = {"Matematica","Quimica","Fisica"};
49       lista                = new JList(disciplinas);
50
51       c.add(labNome);        c.add(texNome);
52       c.add(labCidade);      c.add(texCidade);
53       c.add(labEstado);      c.add(comEstado);
54       c.add(labSexo);        c.add(radMasculino); c.add(radFeminino);
55       c.add(labBolsa);       c.add(cheBolsa);
```

```
56        c.add(labDisciplinas); c.add(lista);
57
58        setSize(450, 200);
59        setVisible(true);
60     }
61
62     public static void main(String args[]){
63        Cadastro app = new Cadastro();
64        app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
65     }
66  }
```

As linhas 2 a 19 declaram os componentes que vamos usar na tela. Veja que cada componente precisa ser declarado e depois instanciado para poder ser usado. O componente é um objeto e lembre-se que Java é fortemente tipada. Os *JLabel* são objetos que exibem textos pequenos ou imagens, ou ambos. No exemplo usamos apenas texto e encorajamos você a pesquisar a *API* do *Swing* e ver outras formas de usar este componente.

A partir da linha 21 temos a montagem da tela de fato. Declaramos o construtor da classe *Cadastro* e a partir da linha 22 até a linha 49 temos a instanciação dos componentes da tela. Verifique a forma de instanciar cada componente. Lembre-se que estamos usando construtores básicos e que cada um deles possui outras formas de serem instanciados.

As linhas 51 a 56 inserem os componentes no *Frame*. Preste atenção que por questões de espaço colocamos 2 ou 3 comandos por linha. Usamos o *layout FlowLayout* no *Frame* do exemplo: lembre-se também que quando não definimos explicitamente o gerenciador de *layout* de um container, o *FlowLayout* será usado por padrão.

Um componente que merece uma atenção maior na sua criação é o *JList*. Veja que no exemplo não foram usadas barras de rolagem na *JList* que colocamos na tela. As barras de rolagem são outros objetos, de outras classes e tem que ser inseridos e acoplados nas *JList* separadamente.

As linhas 58 e 59 definem o tamanho do frame e tornam-no visível. As linhas de 62 a 65 compõem o método principal da classe e instanciam o *frame Cadastro*.

CONEXÃO

Lembre-se sempre de estudar e investigar a API da biblioteca que você está usando para desenvolver seus programas. Como estamos tratando de interfaces gráficas neste capítulo, é importante estudar os componentes da API do *Swing* em http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

Uma interface gráfica possui outros detalhes além dos componentes como por exemplo o tratamento de eventos como os cliques do *mouse*, comandos de arrastar e soltar (*drag and drop*), conexão com bancos de dados entre outras características que não são possíveis cobrir aqui neste material.

Alguns assuntos que sugerimos que você estude para completar o seu estudo sobre interfaces gráficas são:

- Como usar ícones nos componentes, para deixá-los mais intuitivos e com melhor aparência. Botões, menus, listas são exemplos de componentes que podemos associar ícones
- Bordas: as bordas são importantes também para melhorar a aparência das janelas. Lembre-se que existem janelas que não podem ser redimensionadas e isto implica no estudo das bordas das janelas
- Caixas de diálogo: no uso dos programas *desktop* é comum abrir janelas para salvar e abrir arquivo, imprimir um documento, escolher fonte, cores e outras tarefas. Estas janelas são caixas de diálogo. O Java possui classes que implementam estes tipos de janelas e facilitam o trabalho do programador
- *Timers*: componentes que lidam com temporizações diversas.
- *System Tray*: componentes que são executados em segundo plano e ficam alojados na barra do sistema operacional
- *Splash Screens*: criação de telas de abertura de programas com dicas e outras funcionalidades
- Impressão: classes que suportam a impressão em papel
- Modificar o *look and feel*: as aplicações em *Swing* possuem formas de se modificar sua aparência e comportamento. É possível criar temas ou usar outros existentes para deixar a aplicação mais interativa e rica com o usuário.

Lembre-se que ainda vamos abordar no Capítulo 5 o tratamento de eventos e outros conceitos importantes para as interfaces gráficas como as classes aninhadas, as classes internas e classes anônimas.

Percebemos que fornecer componentes consistentes e intuitivos de interface com o usuário aos diferentes programas permite, de certa maneira, que os usuários se familiarizem com um programa para que possa aprendê-lo mais rapidamente.

Vimos que as GUI (*Graphical User Interface*) são construídas a partir de componentes GUI que em Java são implementados por classes do pacote *Swing*. Esses componentes também são chamados de controles ou *widgets*.

A maioria dos componentes *Swing* estão localizados no pacote *javax.swing*. É importante lembrar que as listagens de programas que vimos não possuem as linhas das importações dos pacotes. Ao utilizar uma IDE como o *Netbeans* ou *Eclipse*, a própria ferramenta irá apontar as dependências dos pacotes.

Um último detalhe a ser considerado é que os componentes gráficos podem ser leves ou pesados. Os componentes *Swing* leves não são amarrados aos componentes GUI reais suportados pela plataforma subjacente em que um aplicativo é executado. Logo, os componentes leves são independentes de plataforma ou seja, ele terá a mesma aparência e comportamento executando no *Windows*, *Linux* ou MacOS. Logo, os componentes pesados, principalmente os componentes AWT, são chamados de pesados.

Em seguida apresentamos outros componentes *Swing* disponíveis em Java com suas classes.

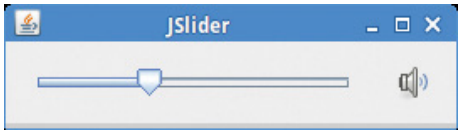


Figura 4.12 – JSlider. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

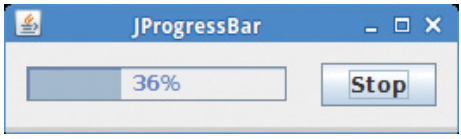


Figura 4.13 – JProgressBar. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

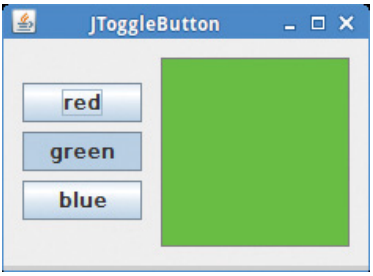


Figura 4.14 – JToggleButton: este botão apresenta 2 estados: pressionado ou não pressionado.
Fonte: http://docs.oracle.com/javase/tutorial/uiswing

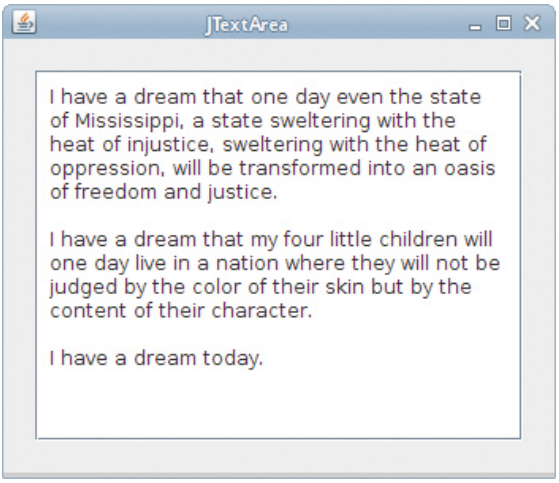


Figura 4.15 – JTextArea. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

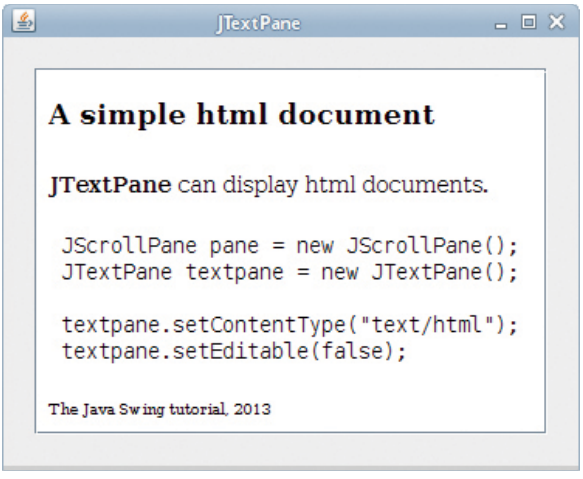


Figura 4.16 – TextPane. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

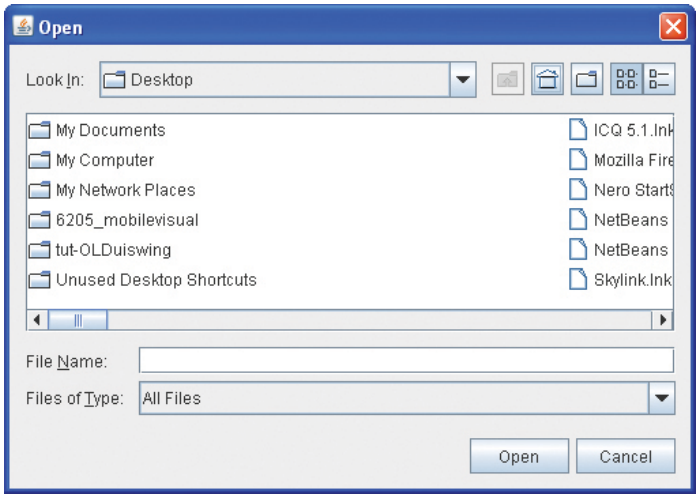


Figura 4.17 – JFileChooser. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

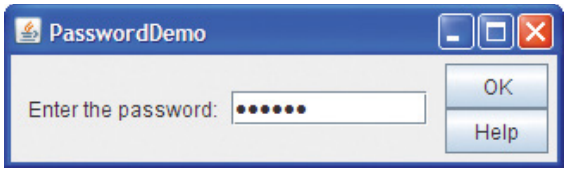


Figura 4.18 – JPasswordField. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

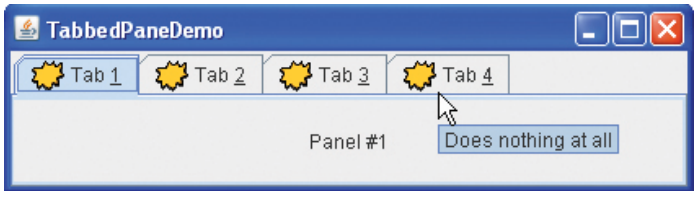


Figura 4.19 – JTabbedPane. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

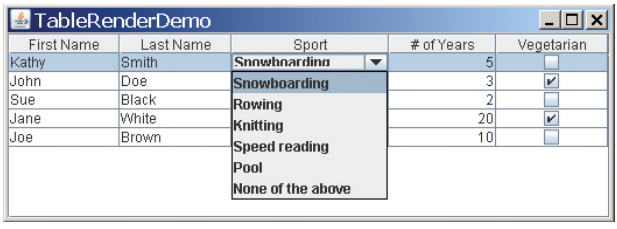


Figura 4.20 – JTable. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

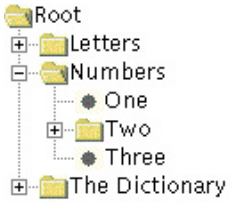


Figura 4.21 – JTree. Fonte: http://docs.oracle.com/javase/tutorial/uiswing

4.4 Telas polimórficas

Nas equipes de desenvolvimento de software existe uma meta que é comum a todas: produtividade com eficiência. A linguagem Java possui recursos para prover a produtividade e a eficiência desejadas.

Considere um desenvolvimento de um sistema no qual são construídos vários cadastros. Um cadastro é composto basicamente por quatro operações: inserção, consulta, atualização e remoção de dados. De uma maneira geral, todo cadastro é igual, logo, poderiam ter uma tela abstrata na qual outras são derivadas e aplicadas a situações específicas.

O código a seguir é proveniente do Netbeans com a paleta de componentes *Swing* e foi obtido arrastando e soltando os componentes no *frame*. O código se refere à classe abstrata *TelaCad* que será a padrão para todas as outras telas de cadastro do sistema. Observe que o código gerado pelo Netbeans é um pouco diferente dos códigos que já demos como exemplo neste capítulo.

```
public abstract class TelaCad extends javax.swing.JFrame {
    public TelaCad() {
        initComponents();
    }
    private void initComponents() {
        botaoIncluir = new javax.swing.JButton();
        botaoConsultar = new javax.swing.JButton();
        botaoAlterar = new javax.swing.JButton();
        botaoExcluir = new javax.swing.JButton();
        botaoSair = new javax.swing.JButton();
```

```
            setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
            botaoIncluir.setText("Incluir");
            botaoConsultar.setText("Consultar");
            botaoAlterar.setText("Alterar");
            botaoExcluir.setText("Excluir");
            botaoSair.setText("Sair");
    botaoSair.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                    handlerBotaoSair(evt);
                }
            });

            javax.swing.GroupLayout      layout      =      new      javax.swing.
GroupLayout(getContentPane());
            getContentPane().setLayout(layout);
            layout.setHorizontalGroup(
                layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addContainerGap()
                    .addComponent(botaoIncluir)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                    .addComponent(botaoConsultar)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                    .addComponent(botaoAlterar)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                    .addComponent(botaoExcluir)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                    .addComponent(botaoSair)
                    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,   Short.
MAX_VALUE))
            );
```

```
          layout.setVerticalGroup(
             layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
             .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,      layout.
createSequentialGroup()
                .addContainerGap(266, Short.MAX_VALUE)
    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
BASELINE)
                    .addComponent(botaoIncluir)
                    .addComponent(botaoConsultar)
                    .addComponent(botaoAlterar)
                    .addComponent(botaoExcluir)
                    .addComponent(botaoSair))
                .addContainerGap())
          );
          pack();
       }
       private void handlerBotaoSair(java.awt.event.ActionEvent evt) {
          System.exit(0);
       }

       public static void main(String args[]) {
          try {
             for  (javax.swing.UIManager.LookAndFeelInfo  info  :  javax.swing.
UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                   javax.swing.UIManager.setLookAndFeel(info.getClassName());
                   break;
                }
             }
          } catch (ClassNotFoundException ex) {
    java.util.logging.Logger.getLogger(TelaCad.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
          } catch (InstantiationException ex) {
    java.util.logging.Logger.getLogger(TelaCad.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
          } catch (IllegalAccessException ex) {
    java.util.logging.Logger.getLogger(TelaCad.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
```

```
            } catch (javax.swing.UnsupportedLookAndFeelException ex) {
        java.util.logging.Logger.getLogger(TelaCad.class.getName()).log(java.util.
    logging.Level.SEVERE, null, ex);
            }
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
                    new TelaCad().setVisible(true);
                }
            });
        }
        private javax.swing.JButton botaoAlterar;
        private javax.swing.JButton botaoConsultar;
        private javax.swing.JButton botaoExcluir;
        private javax.swing.JButton botaoIncluir;
        private javax.swing.JButton botaoSair;
    }
```

As outras telas de cadastro são herdadas de TelaCad. Mas, cada uma contém um painel que determina as peculiaridades de cada cadastro. Todas as funcionalidades comuns são descritas em TelaCad, enquanto que as necessidades particulares são escritas novamente.

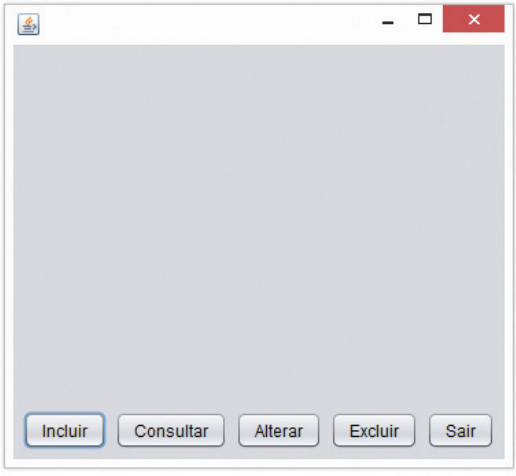


Figura 4.22 – TelaCad sendo executada

4.5 Introdução ao tratamento de eventos com classes aninhadas

Antes de começar a mostrar códigos, listagens e conceitos, vamos pensar em alguma aplicação para poder exemplificar o que vamos estudar neste capítulo. Talvez um cliente de e-mail seja uma aplicação bem popular e conhecida.

Em uma tela deste tipo de aplicação, temos vários componentes: lista (*JList*) com mensagens, remetentes, etc., botões de ação (*JButton*) para envio, composição de uma nova mensagem, respostas, encaminhamentos, telas de abertura de anexos (*JDialog*) e muitos outros componentes.

O programa ao ser executado fica esperando que alguma ação "perturbe" o comportamento de repouso. Esta ação pode ser um clique no mouse, arrastar e soltar uma mensagem e muitas outras. Ou seja, o programa não sabe qual será a próxima ação do usuário.

Portanto, o programa fica "ouvindo" (*Listen*) e esperando alguma ação para poder responder a ela. Basicamente essa é a base para o conceito chamado de programação orientada a eventos.

> A programação orientada a eventos é um paradigma de programação na qual o fluxo do programa é determinado por eventos tais como as ações de usuários (cliques de mouse, pressionamento de teclas, etc.), saídas de sensores ou mensagens de outros programas ou até mesmo de threads. A programação orientada a eventos é determinante em programas com interface gráfica e outras aplicações na web que são centradas em executar certas respostas para entradas de usuários.

O código que executa uma tarefa em resposta a um evento é chamado de **tratador de evento (event handler)** e o processo de responder a eventos é chamado de **tratamento de evento**.

Vamos usar dois componentes do Swing já vistos e que geram eventos para exemplificar: o *JTextField* e o *JPasswordField*. Vamos criar a tela simples da figura 4.23. A listagem para a criação da tela será dividida para melhor entendimento e é mostrada na Listagem 5. Esta listagem já foi estudada anteriormente. Nela está a criação do *frame*, a definição de seus atributos, construtor, definição do *LayoutManager* e o registro dos tratadores de evento. Como não poderia deixar de ser, o tratador é um objeto e neste caso é um objeto da classe *TextFieldHandler*. O nome dado para o objeto foi *handler*.

```
public class TextFieldFrame extends JFrame {
    private JTextField textField1;
    private JTextField textField2;
    private JTextField textField3;
    private JPasswordField passwordField;

    public TextFieldFrame() {
        super( "Testando ..." );
        setLayout(new FlowLayout());

        textField1 = new JTextField( 10 );
        add(textField1);

        textField2 = new JTextField( "Digite um texto aqui");
        add( textField2 );

        textField3 = new JTextField( "TextField Uneditable", 21 );
        textField3.setEditable( false );
        add( textField3 ); //

        passwordField = new JPasswordField( "Texto Oculto" );
        add( passwordField );

        // registro dos tratadores de evento
        TrataTextField handler = new TrataTextField();
        textField1.addActionListener( handler );
        textField2.addActionListener( handler );
        textField3.addActionListener( handler );
        passwordField.addActionListener( handler );
    } // fim do construtor
```

Listagem 5 – Primeira parte da classe de criação da tela da figura 4.23

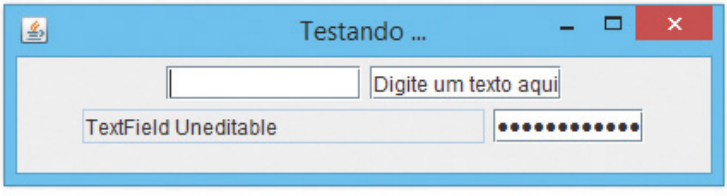


Figura 4.23 – Tela de teste para os tratadores de evento

A Listagem 6 contém a segunda parte da listagem 5. Nesta listagem mostramos a parte na qual o tratamento de evento é configurado. O objetivo deste programa é mostrar um diálogo de mensagem que contém o texto de um campo de texto quando o usuário apertar a tecla *Enter* neste campo de texto.

```
    //classe interna para o tratamento de eventos
        private class TrataTextField implements ActionListener {
            // processa os eventos dos textfields
            public void actionPerformed( ActionEvent evento ) {
                String string = "";

                // usuário apertou Enter no textField1
                if ( evento.getSource() == textField1 )
                    string = String.format( "textField1: %s",
                        evento.getActionCommand() );

                    // usuário apertou Enter no textField2
                else if ( evento.getSource() == textField2 )
                    string = String.format( "textField2: %s",
                        evento.getActionCommand() );

                    // usuário apertou Enter no textField3
                else if ( evento.getSource() == textField3 )
                    string = String.format( "textField3: %s",
                        evento.getActionCommand() );

                    // usuário apertou Enter no passwordField
                else if ( evento.getSource() == passwordField )
                    string = String.format( "passwordField: %s",
                        new String( passwordField.getPassword() ) );
                //mostra o conteudo
                JOptionPane.showMessageDialog( null, string );
            }
        } // fim da classe interna TextFieldHandler
}
```

Listagem 6 – Continuação da listagem 5 contendo os tratadores de eventos

Toda vez que você for implementar um aplicativo que responda a eventos de um componente gráfico, lembrar de alguns passos na codificação:

1. criar uma classe que represente o tratador do evento
2. implementar uma interface apropriada conhecida como **interface *listener* de evento**, na classe do passo 1
3. Indicar que um objeto da classe dos passos 1 e 2 deve ser notificado quando o evento ocorrer. Isso é conhecido como **registrar um tratador de evento**.

4.6 Classes aninhadas

Até agora, todas as classes que usamos como exemplo e foram estudadas são chamadas de **classes de primeiro nível** – são classes que não foram declaradas dentro de outras.

Observe a listagem 6. A classe *TrataTextField* está declarada dentro da classe TextFieldFrame e é chamada de **classe aninhada**. Estas classes podem ser static ou não e também podem ser chamadas de **classes internas** e são usadas frequentemente para tratamento de eventos.

Para que um objeto de uma classe interna seja criado, é necessário criar anteriormente o objeto da classe externa afinal o objeto externo conterá uma referência ao objeto interno.

Existe também uma particularidade entre esses objetos: o objeto da classe interna tem permissão de acessar diretamente todas as variáveis de instância e métodos da classe externa. Uma classe interna que é *static* não precisa de um objeto de sua classe externa e não tem implicitamente uma referência a um objeto da classe externa.

Na listagem 6, o tratamento de eventos é feito pela classe interna *TrataTextField*. Ela foi declarada como private porque vai ser usada apenas para criar handlers de eventos para os *textfield* na classe externa *TextFieldFrame*. Uma classe interna pode ser usada como outros membros de uma classe, ou seja, *public*, *private* ou *protected*.

Estamos usando um exemplo simples mas que já poderia apresentar vários tipos de eventos para serem tratados. Todo evento, como já foi dito, é representado por uma classe e pode ser processado apenas pelo tipo de tratador de evento apropriado. Para conhecer os eventos possíveis e seus tratadores, é

interessante estudar a API do Java para aquela classe do componente e de suas superclasses.

No nosso caso, quando o usuário aperta a tecla Enter é disparado pelo textfield um evento de teclado chamado *ActionEvent* e é processado por um objeto que implementa a interface *ActionListener*. Uma vez que *JPasswordField* é uma subclasse de *JTextField*, *JPasswordField* possui os mesmos eventos.

A classe interna *TrataTextField* implementa a interface *ActionListener* e declara um método, o *actionPerformed()*. Dentro deste método está o código que vai especificar o que será feito quando o evento *ActionEvent* for tratado e assim a classe *TrataTextField* contempla os passos 1 e 2 mostrados anteriormente.

No construtor da classe, na listagem 5, é criado um objeto chamado *handler* da classe *TrataTextField*. O método *actionPerformed()* deste objeto é chamado automaticamente quando o usuário apertar Enter em qualquer um dos *textfields* declarados na classe. Porém antes que isso aconteça, o programa deve registrar esse objeto como o tratador de evento para os três componentes da tela. Isso foi feito no final da listagem 5. O programa chama o método *JTextField.addActionListener* para registrar o tratador de evento para cada componente. Esse método recebe um objeto *ActionListener* como argumento que pode ser um objeto de qualquer classe que implemente *ActionListener*.

O objeto *handler* é um *ActionListener* porque a classe *TrataTextField* implementa a interface *ActionListener* (lembre-se desses conceitos, vistos no Capítulo 4). Depois que as linhas finais da listagem 5 são executadas e os tratadores de eventos registrados, o objeto *handler* passa a ouvir eventos (torna-se então o *listener* – ouvinte). Logo, toda vez que o usuário apertar *Enter*, em qualquer um dos componentes o método *actionPerformed()* da classe *TrataTextField* será executado para tratar o evento. Caso acontecer que um evento não for registrado, após o pressionamento da tecla *Enter*, ele será ignorado pelo programa.

4.6.1 Sobre o método TrataTextField.actionPerfomed()

No nosso exemplo foi usado o método *actionPerformed* de um objeto de tratamento de eventos para tratar eventos gerados por *textfields*. A figura 4.24 mostra o resultado dos eventos no *textField1*, *textField2*, *textField3* e *passwordField*.

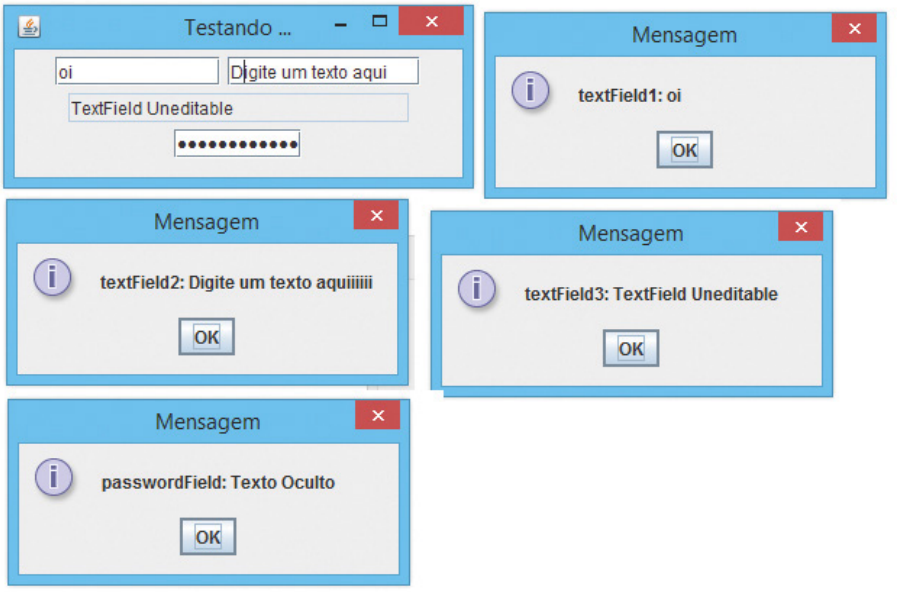


Figura 4.24 – Eventos disparados

No nosso exemplo, o objetivo era determinar qual *textfield* gerou o evento para toda vez que o *actionPerformed* for chamado. O componente no qual o usuário interage é a **origem do evento**. No nosso caso, a origem do evento é um dos *textfields* incluindo o *passwordField*. Quando o usuário apertar *Enter* enquanto um desses componentes tiver o foco, o sistema cria um objeto *ActionEvent* único que contém informações sobre o evento que acabou de ocorrer, como a origem do evento e o texto no *textfield*.

O sistema passa então esse objeto *ActionEvent* em uma chamada de método para o método *actionPerformed* do *listener* de eventos. No nosso caso, mostramos algumas dessas informações em uma caixa de diálogo.

O método *ActioEvent.getSource()* encontrado na Listagem 6 retorna uma referência à origem do evento. A condição (o *if*) verifica se *textfield1*é a origem do evento. Se for, então o programa saberá que o usuário apertou *Enter* no *textfield1* e será montada uma *string* a qual será exibida na tela em uma caixa de diálogo. Isso ocorrerá para os outros componentes.

4.7 Tipos comuns de eventos e interfaces ouvintes

No nosso exemplo tratamos apenas um tipo de evento. Porém sabemos que existem vários outros que podem ocorrer quando um usuário interage com uma aplicação gráfica. As informações sobre qualquer evento de interface gráfica são armazenadas em um objeto de uma classe que estende de *AWTEvent*.

Ao consultar a hierarquia da classe *AWTEvent* na API do Java temos as seguintes subclasses, divididas em categorias (pacote *java.awt.event*):

- ActionEvent (fonte: componentes de ação)
- MouseEvent (fonte: componentes afetados pelo mouse)
- ItemEvent (fonte: checkboxes e similares)
- AdjustmentEvent (fonte: scrollbars)
- TextEvent (fonte: componentes de texto)
- WindowEvent (fonte: janelas)
- FocusEvent (fonte: componentes em geral)
- KeyEvent (fonte: componentes afetados pelo teclado).

CONEXÃO

Mostramos aqui 8 classes do pacote java.awt.event. Porém na atual versão do Java existem 18 e se você vai desenvolver interfaces gráficas é muito recomendável estuda-las. O link para essa parte na API do Java é: http://docs.oracle.com/javase/7/docs/api/java/awt/event/package-summary.html

Vimos que para cada tipo de objeto de evento, há em geral uma interface ouvinte de eventos (um objeto que implementa uma ou mais das interfaces ouvintes de evento) correspondente. Alguns deles são mostrados a seguir:

- ActionEvent: ActionListener
- MouseEvent: MouseListener e MouseMotionListener
- ItemEvent: ItemListener
- AdjustmentEvent: AdjustmentListener

- TextEvent: TextListener
- WindowEvent: WindowListener
- FocusEvent: FocusListener
- KeyEvent: KeyListener

4.8 Como funciona o tratamento de eventos?

Quando um evento de um componente gráfico ocorre, este componente notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte.

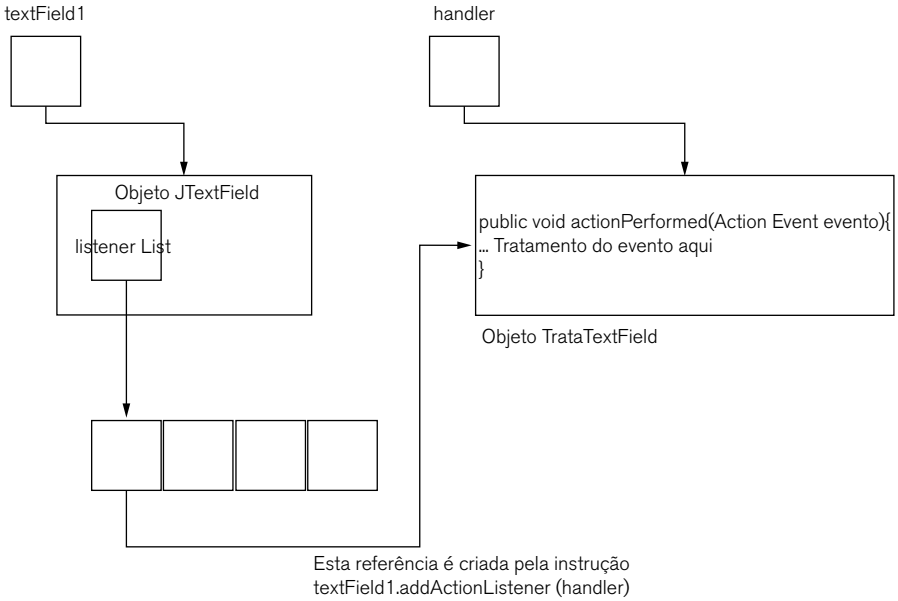


Figura 4.25 – Registro do evento do textField1.

A figura 4.25 ilustra como o mecanismo de eventos funciona usando textField1 como exemplo.

O handler de eventos é registrado no final da listagem 1 pelas linhas:

```
// registro dos tratadores de evento
   TrataTextField handler = new TrataTextField();
   textField1.addActionListener( handler );
   textField2.addActionListener( handler );
   textField3.addActionListener( handler );
   passwordField.addActionListener( handler );
```

Cada *JComponent* tem uma variável de instância chamada *listenerList* a qual faz referência a um objeto da classe *EventListenerList*. Cada objeto de uma subclasse de *JComponent* mantém uma referência a todos os seus *listeners* registrados na *listenerList*. A figura 4.25 ilustra essa situação.

Como mostra a figura, quando o método *addActionListener*() for chamado, uma nova entrada que contém uma referência ao objeto *TrataTextField* é colocada na *listenerList* de *textField1*. Essa nova entrada contém o tipo de *listener* (*ActionListener* neste caso). E assim, cada componente que usar este mecanismo mantém sua própria lista de listeners que foram registrados para tratar os eventos do componente.

Uma dúvida que pode ocorrer é como o componente sabe chamar *actionPerformed* ao invés de outro método. Sabemos que cada componente suporta vários tipos de eventos e quando um deles ocorre, há um envio apenas para os *listeners* de evento do tipo apropriado. Esse envio é chamado de despacho (*dispatch*) e se refere simplesmente ao processo pelo qual o componente chama um método de tratamento de evento em cada um de seus *listeners* que são registrados para o tipo particular de cada evento que ocorreu.

Quando ocorre um evento, o componente recebe um **id de evento** da JVM para especificar o tipo de evento. O componente então usa este id para decidir o tipo de listener para o evento ser enviado (*dispatched*) e decidir qual método chamar em cada objeto ouvinte. No evento *ActionEvent* é despachado para um actionPerformed de cada *ActionListener* que foi registrado. Quem faz este trabalho é o componente e tudo que o programador precisa fazer é registrar um tratador de evento correspondente para o tipo particular de evento que deseja ser tratado e o componente assegurará que o método apropriado do tratador de evento será chamado quando o evento ocorrer.

A tabela 4.1 mostra alguns eventos, *listeners* e operações para você se familiarizar:

ACTIONEVENT	ACTIONLISTENER	ACTIOPERFORMED (ACTIONEVENT)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseEvent	MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
TextEvent	TextListener	textValueChanged(TextEvent)
WindowEvent	WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconifed(WindowEvent) windowIconifed(WindowEvent) windowOpened(WindowEvent)

Tabela 4.1 – Alguns eventos, listener e ações

Visto esses conceitos, vamos estudar outro exemplo para tratarmos do próximo assunto: os adaptadores e as classes internas anônimas.

4.8.1 Adaptadores de eventos

Algumas interfaces de listeners contém mais do que um método. Por exemplo, a interface *MouseListener* contém 5 métodos: *mousePressed*, *mouseReleased*, *mouseEntered*, *mouseExited* e *mouseClicked*. Como estamos falando de uma interface, então mesmo se a sua aplicação só necessite tratar o clique do *mouse*, é obrigatório que você implemente todos os outros quatro métodos. Neste caso, é possível deixar o corpo dos outros métodos em branco como no exemplo a seguir:

```
//Um exemplo que implementa uma interface de listener diretamente.
public class MinhaClasse implements MouseListener {
   ...
      objeto.addMouseListener(this);
   ...
   /* Definição de método vazia. */
   public void mousePressed(MouseEvent e) {
   }

   /* Definição de método vazia. */
   public void mouseReleased(MouseEvent e) {
   }

   /* Definição de método vazia. */
   public void mouseEntered(MouseEvent e) {
   }

    /* Definição de método vazia. */
    public void mouseExited(MouseEvent e) {
    }

   public void mouseClicked(MouseEvent e) {
      ...//A implementação do evento do listener vai aqui...
   }
}
```

Como podemos perceber, este conjunto de métodos vazios prejudica a leitura do código e sua manutenção. Para evitar a implementação de métodos vazios, a API geralmente inclui uma classe **adaptadora** para cada interface que possui mais de um método.

CONEXÃO

A tabela de *Listeners* da API lista todos os listeners e seus adaptadores.
http://docs.oracle.com/javase/tutorial/uiswing/events/api.html

Para usar um adaptador, você cria uma subclasse e sobrescreve somente os métodos de interesse ao invés de implementar todos os métodos da interface. O código a seguir mostra um exemplo da modificação do código anterior para estender *MouseAdapter*. Estendendo *MouseAdapter*, ele herda as definições vazias de todos os cinco métodos que MouseListener contém.

```
    /*
     * Um exemplo de classe adaptadora ao invés de implementar uma interface de
listener
     */
    public class MinhaClasse extends MouseAdapter {
       ...
          objeto.addMouseListener(this);
       ...
       public void mouseClicked(MouseEvent e) {
          ...//Implementação vai aqui...
       }
    }
```

Vamos usar outro exemplo para ilustrar o uso de classes adaptadoras. O resultado do exemplo é apresentado na figura 4.26. Trata-se de um programa que conta quantas vezes o botão direito, central e esquerdo do mouse foram pressionados e mostra em uma barra de status na parte inferior do *frame*.

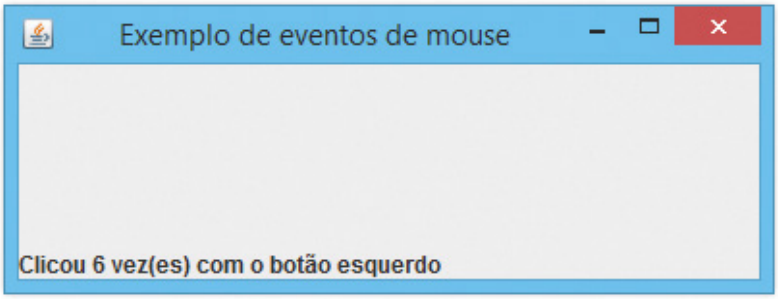


Figura 4.26 – Execução do programa da Listagem 6.

```
public class MouseFrame extends JFrame {
   private String detalhes;
   private JLabel statusBar;
   // constructor sets title bar String and register mouse listener
   public MouseFrame() {
       super( "Exemplo de eventos de mouse" );

       statusBar = new JLabel( "Clique o mouse" );
       getContentPane().add( statusBar, BorderLayout.SOUTH );
       addMouseListener( new MouseClickHandler() ); // add handler
   }
   // classe interna para lidar com os eventos do mouse
   private class MouseClickHandler extends MouseAdapter {
       //trata o clique do mouse e verifica qual botão foi pressionado
       public void mouseClicked( MouseEvent evento ){
           int posX = evento.getX();
           int posY = evento.getY();

           detalhes = "Clicou " + evento.getClickCount() + " vez(es)";

           if ( evento.isMetaDown() ) // botão direito
               detalhes += " com o botão direito";
           else if (evento.isAltDown() ) // botão do meio
               detalhes += " com o botão do meio";
           else // botão esquerdo
               detalhes += " com o botão esquerdo";
```

```
            statusBar.setText(detalhes);
        }
    }
}
-------------------------------------------------------------
public class TestaMouseFrame {
    public static void main(String[] args) {
        MouseFrame mf = new MouseFrame();
        mf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mf.setSize(400, 150);
        mf.setVisible(true);
    }
}
```

Listagem 7 – Exemplo de classe adaptadora

No programa da listagem 7 foi usada a classe *MouseClickHandler* que herda a classe *MouseAdapter*. Neste caso percebemos que não é necessário implementar todos os métodos de *MouseAdapter* pois esta funciona como uma classe adaptadora.

O evento do mouse é captado e avaliado junto a 2 métodos: *isMetaDown*(), *isAltDown*. Dependendo da ação do mouse, o evento é captado, associado ao evento e a string detalhes é atualizada.

A *string* atualizada é então copiada para a propriedade *text* do componente *statusBar*.

4.8.2 Classes internas e classes anônimas internas

Embora já visto neste capítulo, vamos enfatizar as classes internas neste tópico e relaciona-las com as classes **anônimas.**

E se você quiser usar uma classe adaptadora mas não quer que sua classe pública herde de um adaptador? Por exemplo, vamos supor que você escreva um *Applet* e queira que ele tenha um tratamento de eventos de *mouse*.

Um *Applet* é uma pequena aplicação Java que normalmente roda em um navegador e executa uma atividade específica. Um exemplo de *applet* é encontrado nos *internet banking* que possuem teclados virtuais. Normalmente possui uma interface gráfica com o usuário. Uma aplicação *Applet* é um objeto que herda da classe *JApplet*.

Uma vez que a linguagem Java não permite herança múltipla, a classe do *applet* que vamos implementar não poderá herdar das classes *Applet* e *MouseAdapter* ao mesmo tempo. Uma solução é definir uma classe interna como já vimos.

Também vimos que classes internas podem ser úteis para *listeners* de eventos que implementam uma ou mais interfaces diretamente como no exemplo a seguir:

```
//Exemplo de classe interna
public class MinhaClasse extends Applet {
   ...
       objeto.addMouseListener(new MeuAdaptador());
   ...
   class MeuAdaptador extends MouseAdapter {     //classe interna
       public void mouseClicked(MouseEvent e) {
           ...//Aqui vai a implementação
       }
   }
}
```

É possível criar uma classe interna sem especificar um nome. Ela é chamada de **classe anônima**. Embora possa parecer estranho, classes anônimas podem fazer o seu código ficar mais fácil de ser lido porque a classe está definida onde ela é referenciada. Entretanto, você vai precisar verificar a conveniência contra possíveis implicações de desempenho no aumento do número de classes. Veja o exemplo a seguir:

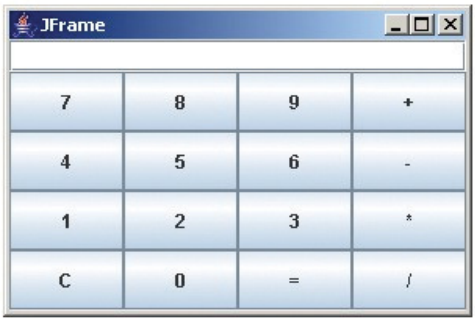
```
//Exemplo de classe anônima
public class MinhaClasse extends Applet {
   ...
      objeto.addMouseListener(new MouseAdapter() {  //classe anônima
         public void mouseClicked(MouseEvent e) {
            ...//Implementação do listener de evento vai aqui...
         }
      });
   ...
   }
}
```

As classes internas trabalham mesmo que o seu ouvinte de evento precisa ter acesso a variáveis de instância privadas da classe externa . Contanto que você não declare uma classe interna para ser estática, uma classe interna pode se referir a variáveis de instância e métodos, assim como se o seu código está na classe externa. Para fazer uma variável local disponível para uma classe interna, basta salvar uma cópia da variável como uma variável local *final*.

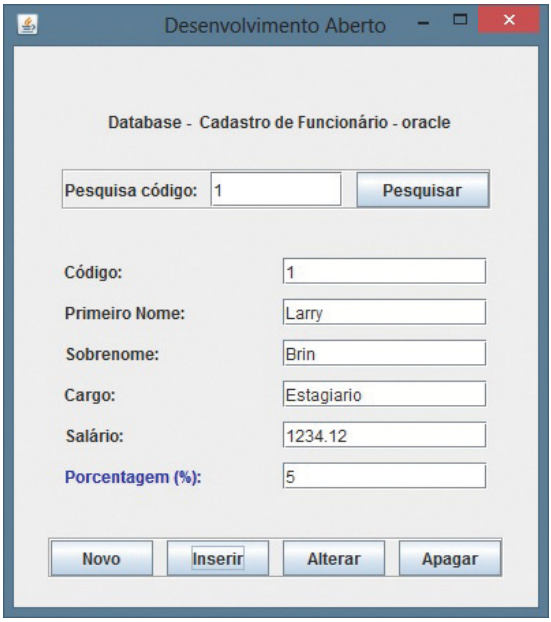
Em Java, uma classe *final* não pode ter subclasses. Um método ***final*** não pode ser sobrecarregado. Uma variável ***final*** só pode ser instanciada uma vez, ou por um inicializador ou por uma declaração de atribuição.

ATIVIDADES

01. Monte a seguinte tela com as suas funcionalidades.

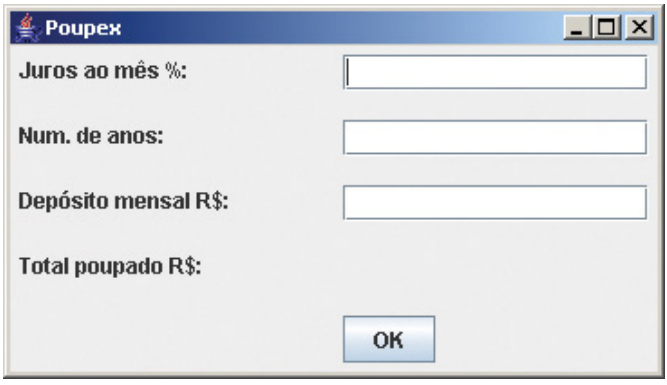


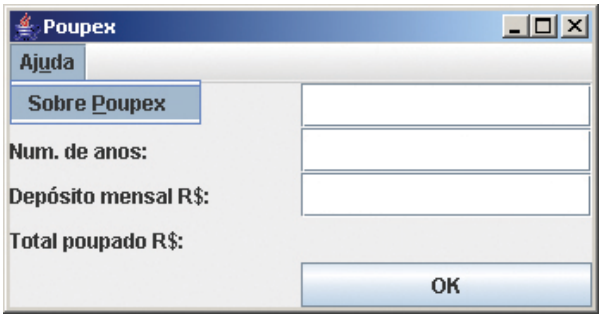
02. Monte a seguinte tela, sem acrescentar funcionalidades



03. Usando a classe *JOptionPane* faça um programa que converta graus Fahrenheit para Celsius. A fórmula de conversão é: Celsius = 5/9*(Fahrenheit-32).

04. Faça a seguinte tela para calcular o total poupado por uma aplicação na poupança.





REFLEXÃO

Os componentes apresentados neste capítulo são os mais básicos. Ao longo do tempo, percebemos que as empresas que desenvolvem aplicações como editores de texto, planilhas eletrônicas e outras tem inovado em componentes mais intuitivos e dinâmicos. Um bom programador Java também precisa acompanhar essa evolução e usar a própria plataforma da linguagem, a qual é aberta, a criar e inovar as interfaces dos seus programas a fim de poder tornar o programa cada vez mais amigável para o seu usuário.

LEITURA

Os tutoriais da Oracle são excelentes para o aprendizado de Java. Para o *Swing* existe uma colação de tutoriais disponíveis em: http://docs.oracle.com/javase/tutorial/uiswing/

Um livro que traz bastante informação e integra com outros conceitos é o: Java 6 – Fundamentos, Swing, BlueJ e JDBC (Ivan Mecenas, Editora Alta Books, 2008).

Um livro excelente é o Java Swing (Robert Eckstein, Marc Loy, Dave Wood, O'Reilly Media, 1998). É um pouco antigo mas os fundamentos presentes no livro não mudaram até hoje.

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Bookman, 2002.

FLANAGAN, D. **Java o guia essencial**. Bookman, 2006.

GARY, C.; HORSTMANN, C. **Core Java 2**: Fundamentos. Makron Books, 2001.

GARY, C.; HORSTMANN, C. **Core Java 2**: Recursos Avançados. Makron Books, 2001.

HUBBARD, J. R. **Programação com Java**. Bookman, 2006.

5

Tratamento de Exceções e Breve Estudo de Caso

Já vimos que a interação entre o usuário e o computador é muito importante. Uma boa interface, com componentes modernos é recomendável para que se tenha um bom uso do programa. Porém as vezes ocorrem alguns erros e isso é mostrado para o usuário.

O usuário por sua vez certamente preferirá que a mensagem de erro seja esclarecedora e o oriente para a correta resolução do problema. Mas as vezes a mensagem vem de forma extremamente técnica e não possui uma descrição clara do problema. Isso é chamado de exceção que na verdade pode ser entendida como uma condição anormal ou especial que requer um processamento especial e que muda o estado normal do programa. Vamos aprender como tratar essas ocorrências.

Além disso, vamos fechar a disciplina mostrando um pedaço de uma aplicação prática envolvendo o padrão MVC.

Bons estudos!

OBJETIVOS

Neste capítulo nosso objetivo é:

- Aprender os conceitos básicos do tratamento de exceções
- Apresentar um estudo de caso prático

5.1 Introdução ao tratamento de exceções (Exceptions)

Em uma aplicação podem ocorrer erros durante a execução de um programa: uma divisão por zero imprevista, uma falha na entrada de dados, selecionar dados de um banco de dados que não está conectado no momento, enfim, existem várias possibilidades de erro.

Alguns programadores usam códigos alfanuméricos para representar os erros que ocorrem nos programas, por exemplo no código abaixo, se não houver valor suficiente para sacar, o método retorna o número 99 para o chamador. Depois disto, o chamador iria tratar este código (erro) devidamente e informar o usuário.

```
int sacar(double valor) {
   if(valorDisponivel < 0){
      return 99; // código de erro para valor negativo
   }
   else {
      this.valorDisponivel -= valor ;
      return 0; // sucesso
   }
}
```

Usar códigos de erros foi bastante utilizado e isso representava um problema pois documentar todos estes códigos era complicado e não produtivo. As linguagens de programação mais modernas possuem um mecanismo de tratamento de erro mais eficiente. Em Java não são usados códigos de erros ou outros tipos de retorno dos métodos.

Em Java é importante saber o tipo de erro que ocorreu. Existe uma classe chamada *Throwable* que implementa subclasses para essas tratativas:

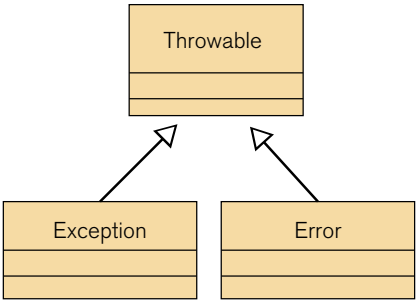


Figura 5.1 – Hierarquia básica da classe Throwable

A subclasse *Exception* possui métodos que definem erros nos quais a aplicação consegue trata-los e continuar sendo executada. A subclasse *Error* possui métodos que definem os erros que não devem ser capturados pelas aplicações porque são erros graves e não possibilitam que a execução do programa continue satisfatoriamente.

Uma exceção é um evento que ocorre durante a execução de um programa que interrompe o fluxo normal das instruções.

Quando um erro ocorre dentro de um método, o método cria um objeto e informa ao sistema de execução da JVM. O objeto, que é a exceção, contém informação sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu. Ao criar um objeto de exceção e fazer o seu tratamento no sistema de execução da JVM é chamado de **tratamento de exceção**.

Depois que um método lança uma exceção, o sistema tenta achar alguma coisa para trata-lo. O conjunto possível de alternativas de tratamento é uma lista ordenada de métodos que são chamados para pegar o método onde o erro ocorreu. Esta lista de métodos é chamada de *call stack* (pilha de chamada) mostrada na 5.1.

O sistema de execução procura a pilha por um método que contenha um código que pegue a exceção. Este bloco é chamado de **tratador de exceção**. A procura começa com o método no qual o erro ocorreu e procede na pilha de chamadas na ordem reversa na qual o método foi chamado (veja a figura 5.2). Quando um tratador apropriado é encontrado, o sistema de execução passa a exceção para o tratador. Um tratador é considerado apropriado se o tipo do objeto lançado "casa" com o tipo que foi pego pelo tratador.

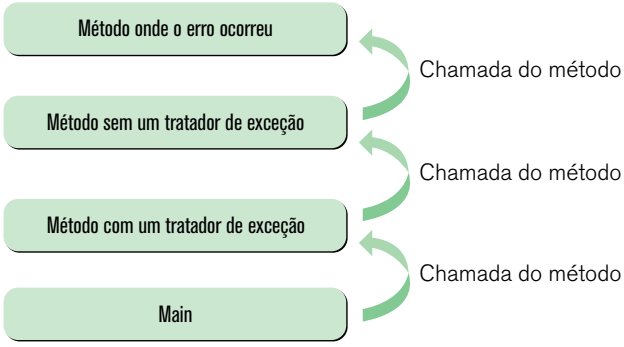


Figura 5.2 – A pilha de chamadas

O tratador escolhido é quem pega (*catch*) a *exceção* (vamos entender o *catch* um pouco mais a frente). Se o sistema de execução não encontrar um tratador apropriado como mostrado na figura 5.3, o programa termina e a mensagem de erro com a exceção é mostrada no console do sistema operacional executando o programa.

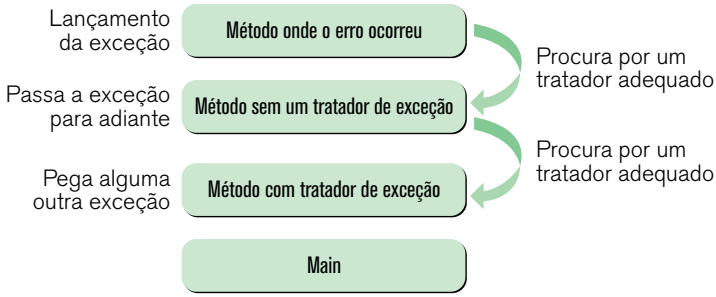


Figura 5.3 – Procura pelo tratador de exceção

As exceções são classificadas em dois tipos: *unchecked* e *checked*. Veja a figura 5.4:

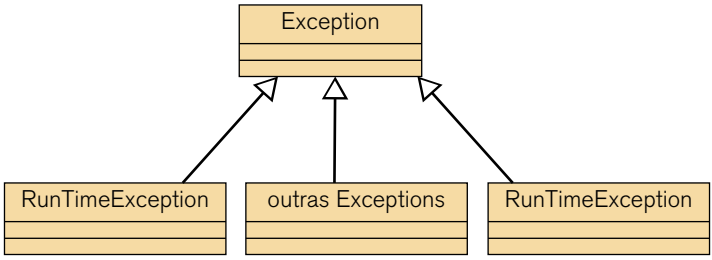


Figura 5.4 – Tipos de erros de execução.

Todas as classes que estão vinculadas à classe *RunTimeException* são *unchecked*. As outras classes que não fazem parte da *RunTimeException* são *checked*.

Quando ocorrer um erro durante a execução do programa e o identificamos este erro, é possível criar algum objeto de uma classe *unchecked* e lançar a referência dele com o comando *throw*:

```
public void depositar(double valor) {
   if (valor<0) {
       IllegalArgumentException erro = new IllegalArgumentException();
       throw erro ;
   }
   else {
       ...
   }
}
```

Outra forma é lançar uma *checked* exception por meio da referência de um objeto com o comando *throw*. Mas para lançar uma *checked exception* é necessário determina-la de maneira explícita com o comando *throws* o qual pode lançar este tipo de erro:

```
public void depositar(double valor) throws Exception {
   if (valor<0) {
       Exception erro = new Exception();
       throw erro ;
   }
   else {
       ...
   }
}
```

Agora vamos estudar outra forma de tratar exceções e a maneira usada na maioria das situações. O primeiro passo para construir um tratador de exceções é criar um bloco *try*. Este bloco normalmente tem a seguinte estrutura (os blocos *catch* e *finally* não são obrigatórios):

```
try {
   ... // código onde pode ser lançada uma exceção
}
catch (nome da exceção){
   ...,
}
catch (nome da exceção) {
   ...
}
finally {
   ...
}
```

O bloco *catch* contém o código que será executado se e quando o tratador de exceção for chamado. O sistema de execução da JVM chama o tratador de exceção quando o tratador for o primeiro da pilha de chamadas onde há o "casamento" do tipo da exceção com o tipo do tratador. Veja o exemplo:

```
1  try {
2     // código onde pode conter uma exceção
3  } catch (IndexOutOfBoundsException e) {
4     System.err.println("IndexOutOfBoundsException: " + e.getMessage());
5  } catch (IOException e) {
6     System.err.println("IOException: " + e.getMessage());
7  }
```

O bloco *try* executa a parte do código na qual a exceção pode aparecer. Se a exceção for um índice não encontrado em um *array* (*IndexOutOfBoundException*) o tratador será o objeto e da linha 4. Caso ocorra uma exceção, como por exemplo uma falha ao escrever um arquivo, vai ocorrer a exceção *IOException* e o tratamento dela será feito a partir da linha 5. Neste caso o tratamento é simples e só irá mostrar uma mensagem no console mas outros comandos mais elaborados podem ser colocados dentro do tratador de exceção como por exemplo mostrar uma janela de erros mais bonita, informar o usuário o procedimento para evitar o erro novamente, etc. Porém neste caso, o programa terminará.

Para o programa não terminar, usamos o bloco *finally*. Ele **sempre** executa quando o bloco *try* termina. Isto garante que o bloco *finally* é executado mesmo que uma exceção imprevista ocorra. Mas o *finally* é útil não somente para o tratamento de exceções – ele pode ser usado sempre.

CONEXÃO

Alguns links que podem te ajudar a entender um pouco mais sobre exceções em Java:
http://docs.oracle.com/javase/tutorial/essential/exceptions/
http://blog.caelum.com.br/lidando-com-exceptions/
http://www.tutorialspoint.com/java/java_exceptions.htm

5.2 Breve estudo de caso

Vamos reunir os conceitos em uma pequena aplicação com o objetivo de mostrar como seria o desenvolvimento de uma aplicação maior.

Antes de começar o desenvolvimento de qualquer sistema é importante que o analista/programador planeje o que vai ser desenvolvido. A arquitetura do sistema é muito importante pois assim trabalhos adicionais podem ser evitados.

Uma arquitetura de sistemas que tem se destacado ao longo dos últimos anos é a MVC (*Model – View – Controller: modelo – visualização – controle*).

Este modelo é composto por três partes:

• Modelo: o modelo representa os dados e as regras que governam o acesso e atualizações destes dados. Em *softwares* empresariais por exemplo, um modelo frequentemente serve como uma aproximação no *software* com o mundo real

• Visualização: a visualização mostra os conteúdos do modelo. Ela especifica exatamente como os dados do modelo devem ser apresentados. Se os dados do modelo forem alterados, a visão precisa atualizar a sua apresentação da forma necessária. Isso pode ser alcançado pelo uso de um *push model*, no qual a própria visão registra-se com o modelo para ser notificado de mudanças, ou de um *pull model*, no qual a visão é responsável por chamar o modelo quando precisa recuperar os dados mais recentes.

• Controlador: o controlador traduz as interações do usuário com a visão

em ações que o modelo executará. Em uma aplicação gráfica *stand-alone*, as interações do usuário podem ser cliques em botões ou seleções de menu, enquanto que em uma aplicação *web*, essas interações tomam forma de requisições HTTP GET e POST. Dependendo do contexto, um controlador pode também selecionar uma nova visão, por exemplo uma página *web* de resultados, para ser apresentada ao usuário.

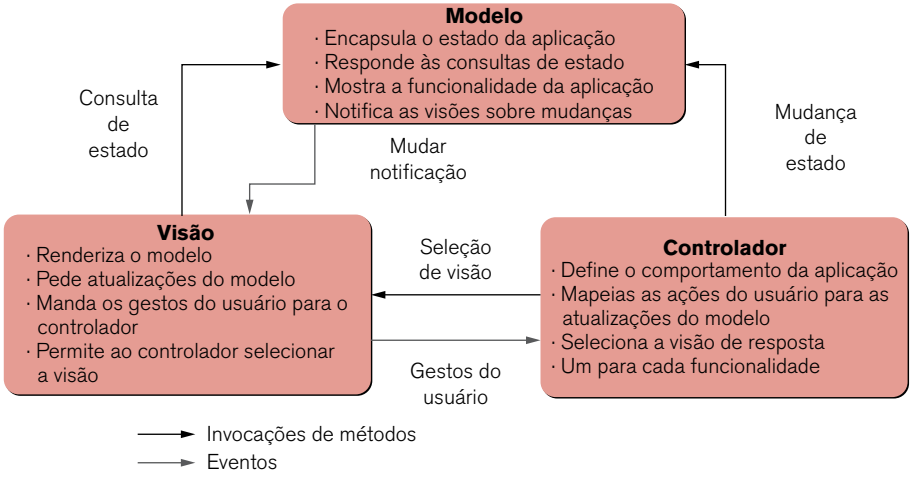


Figura 5.5 – Uma implementação comum do MVC

5.2.1 Interação entre os componentes do MVC

A figura 5.5 mostra uma forma de se implementar o padrão MVC. Uma vez que os objetos do modelo, visão e controlador sejam instanciados, ocorrem os seguintes passos:

7. A visão se registra com um ouvinte do modelo. Qualquer mudança que ocorrer nos dados do modelo imediatamente é transmitida uma notificação a qual é recebida pela visão. Isso é um exemplo do *push-model*. O modelo neste momento não está ciente da visão ou do controlador, ele simplesmente dispara notificações a todos os ouvintes interessados.
8. O controlador é ligado à visão. Portanto, qualquer ação do usuário que é executada na visão invocará um método na classe do controlador.
9. O controlador é fornecido como referência para o modelo.

Uma vez que o usuário interaja com a visão, as seguintes ações ocorrem:

1. A visão reconhece uma ação que vem da interface, como por exemplo, pressionar um botão ou arrastar uma barra de rolagem, usando um método apropriado que é registrado de modo que seja chamado cada vez que a ação ocorra.
2. A visão chama o método apropriado no controlador.
3. O controlador acessa o modelo, possivelmente atualizando-o de forma adequada a ação do usuário.
4. Se o modelo for alterado, ele notifica os ouvintes interessados, com as visões, da alteração. Em algumas arquiteturas, o controlador pode ser responsável também pela atualização da visão.

Em resumo, o uso do padrão MVC permite desacoplar a interface da aplicação em si. A interface não conhece a estrutura da aplicação, e a aplicação não depende de nenhuma interface em particular. Um sistema bem desenhado com MVC permite trocar de interface mais facilmente, assim como permite mudar a estrutura interna dos dados sem alterar a interface com o usuário.

Uma vez que o conceito do MVC foi apresentado, é interessante estudar um exemplo prático.

5.3 Estudo de caso – Desenvolvimento de uma interface gráfica para controle de atividades de um clube

Vamos implementar o sistema representado pela figura 5.6. Veja que na figura temos uma divisão a qual representa a Visão do sistema e o Modelo. Por questões de escopo não vamos tratar a parte do controlador. Isto exigiria outros conhecimentos que não foram tratados neste material porém estão disponíveis na internet com um vasto material.

Observamos que na parte da visão temos as seguintes classes: TelaPrinc, TelaCad, TelaCadClube e TelaCadSocio. Já tratamos anteriormente sobre a TelaCad. Este *frame* servirá como classe abstrata para derivar as outras duas telas da aplicação: TelaCadClube e TelaCadSocio.

As classes de negócio são: Atividade, Sócio e Alocação. O sistema é uma parte de uma aplicação maior que trata da manutenção de um clube recreativo. A parte que vamos considerar no estudo de caso é responsável pelo cadastramento de atividades, como por exemplo carnaval de salão, campeonato de futebol, ginacana, etc e os sócios que vão aderir à atividade. Desta forma é possível guardar um histórico referente às atividades que o sócio já participou.

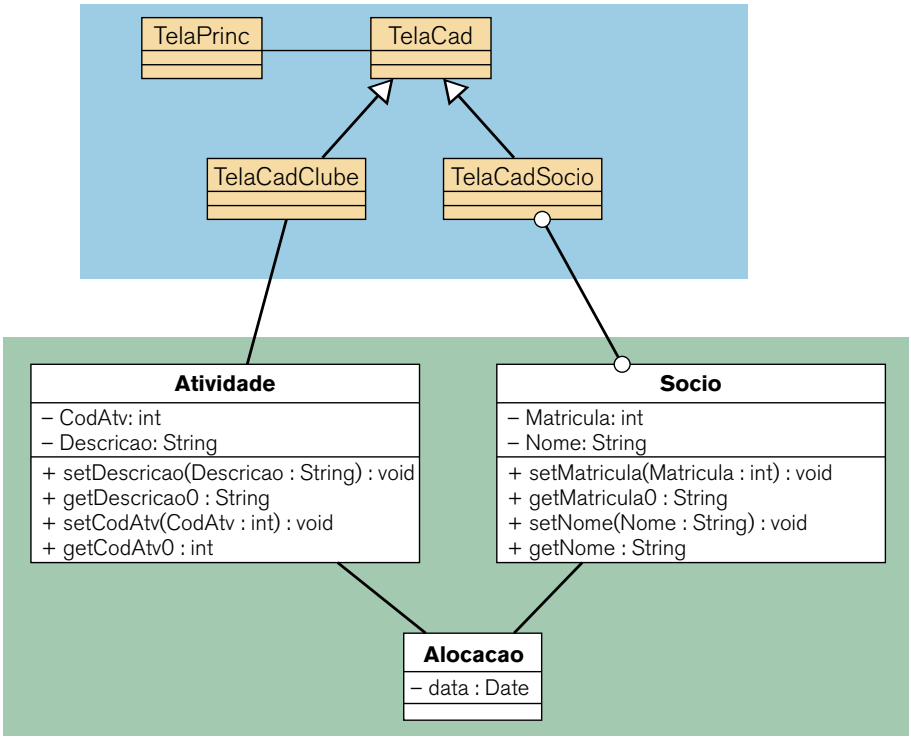


Figura 5.6 – Diagrama de classes do exemplo.

5.3.1 Criando as classes de interface

A tela principal terá a aparência conforme a figura 5.7. O menu Cadastro irá acessar as duas telas de cadastro: a de Sócio e a do Clube e o menu Sair, mostrado na figura 5.8 irá encerrar a aplicação após mostrar uma confirmação usando o método *JOptionPane.showConfirmDialog()*.

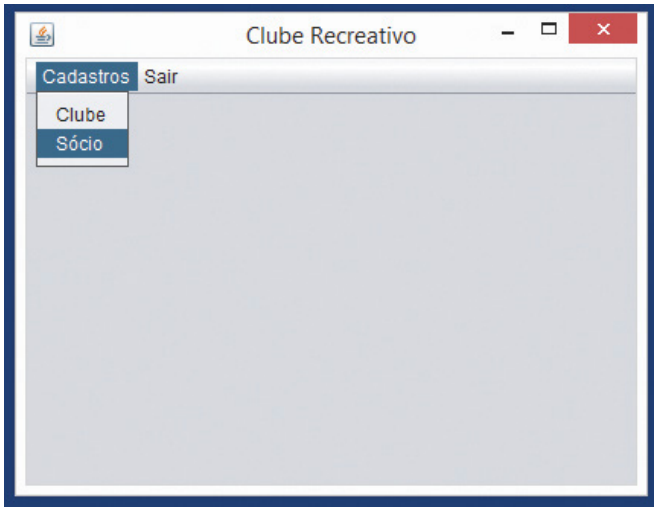


Figura 5.7 – Tela principal

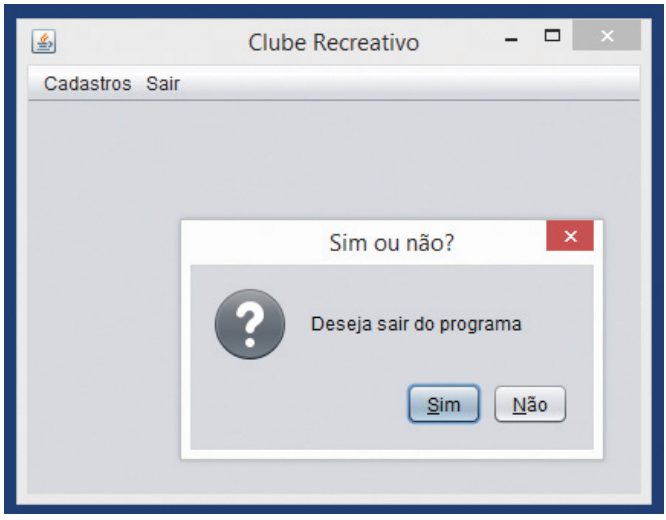


Figura 5.8 – Saindo do programa

A listagem 8 mostra o código da classe TelaPrinc. O código foi gerado pelo Netbeans usando os recursos de arrastar e soltar para criar o *frame* e posteriormente modificado para poder ficar adequado com o nosso exemplo. Veja que o código possui algumas diferenças em relação à forma pela qual montamos os *frames* anteriormente. O Netbeans usa um método chamado *initComponents()*

dentro do construtor o qual é implementado com todas as configurações iniciais do *frame*, inclusive com as chamadas para os tratamentos de evento.

Devido à montagem da tela ter usado o arrastar e soltar, o Netbeans usa um gerenciador de layout próprio para poder prover as funcionalidades e aparência que o usuário deseja. A seta mostrada na linguagem destaca o trecho do código no qual o layout é configurado.

```
import javax.swing.JOptionPane;
public class TelaPrinc extends javax.swing.JFrame {
    public TelaPrinc() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    private void initComponents() {
        jMenuBar1 = new javax.swing.JMenuBar();
        menuCadastros = new javax.swing.JMenu();
        menuClube = new javax.swing.JMenuItem();
        menuSocio = new javax.swing.JMenuItem();
        menuSair = new javax.swing.JMenu();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("Clube Recreativo");

        menuCadastros.setText("Cadastros");
        menuClube.setText("Clube");
        menuClube.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                menuClubeActionPerformed(evt);
            }
        });
```

```
        menuSocio.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                menuSocioActionPerformed(evt);
            }
        });
        menuCadastros.add(menuClube);
        menuSocio.setText("Sócio");
        menuCadastros.add(menuSocio);
        jMenuBar1.add(menuCadastros);
        menuSair.setText("Sair");
        menuSair.addMenuListener(new javax.swing.event.MenuListener() {
            public void menuCanceled(javax.swing.event.MenuEvent evt) {
            }
            public void menuDeselected(javax.swing.event.MenuEvent evt) {
            }
            public void menuSelected(javax.swing.event.MenuEvent evt) {
                menuSairMenuSelected(evt);
            }
        });
        jMenuBar1.add(menuSair);
        setJMenuBar(jMenuBar1);
        javax.swing.GroupLayout layout = new javax.swing.
GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(          layout.createParallelGroup(javax.
swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 400, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(            layout.createParallelGroup(javax.
swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 279, Short.MAX_VALUE)
        );
        pack();
    }
```

```
    private void menuClubeActionPerformed(java.awt.event.ActionEvent evt) {
        TelaCadClube tela = new TelaCadClube();
        tela.setVisible(true);
    }

    private void menuSocioActionPerformed(java.awt.event.ActionEvent evt) {
        TelaCadSocio tela = new TelaCadSocio();
        tela.setVisible(true);
    }

    private void menuSairMenuSelected(javax.swing.event.MenuEvent evt) {
        int opcao = JOptionPane.showConfirmDialog(null,"Deseja sair do progra-
ma", "Sim ou não?", JOptionPane.YES_NO_OPTION);
        if (opcao == JOptionPane.YES_OPTION){
            System.exit(0);
        }
    }

    public static void main(String args[]) {
        try {
            for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.
UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {

    javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
        } catch (Exception ex) {
    java.util.logging.Logger.getLogger(TelaPrinc.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
        }
```

```
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new TelaPrinc().setVisible(true);
            }
        });
    }

    private javax.swing.JMenuBar jMenuBar1;
    private javax.swing.JMenu menuCadastros;
    private javax.swing.JMenuItem menuClube;
    private javax.swing.JMenu menuSair;
    private javax.swing.JMenuItem menuSocio;
}
```

Listagem 8

A listagem 9 refere-se ao código da classe abstrata TelaCad. Esta classe serve como um padrão para gerar as telas de cadastro do sistema e por ser abstrata não será instanciada. Sendo assim, os botões de inclusão, alteração, consulta e exclusão são definidos porém nenhum código de implementação será feito nesta classe. Apenas o botão Sair será implementado pois o seu comportamento será comum a todos os cadastros. Neste caso, o botão fechará a tela. Note que os atributos da classe estão definidos como *protected* pois eles deverão ser usados nas subclasses para receber o código de implementação de cada botão.

```
public abstract class TelaCad extends javax.swing.JFrame {
    public TelaCad() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    private void initComponents() {
        botaoIncluir = new javax.swing.JButton();
        botaoConsultar = new javax.swing.JButton();
        botaoAlterar = new javax.swing.JButton();
```

```
            botaoExcluir = new javax.swing.JButton();
            botaoSair = new javax.swing.JButton();
        setDefaultCloseOperation(javax.swing.WindowConstants.DO_NOTHING_ON_CLOSE);
            setResizable(false);
            setType(java.awt.Window.Type.UTILITY);
            botaoIncluir.setText("Incluir");
            botaoConsultar.setText("Consultar");
            botaoAlterar.setText("Alterar");
            botaoExcluir.setText("Excluir");
            botaoSair.setText("Sair");
            botaoSair.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                    botaoSairActionPerformed(evt);
                }
            });

            javax.swing.GroupLayout      layout      =      new      javax.swing.
GroupLayout(getContentPane());
            getContentPane().setLayout(layout);
            layout.setHorizontalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                .addComponent(botaoIncluir)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(botaoConsultar)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(botaoAlterar)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(botaoExcluir)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(botaoSair)
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,      Short.
MAX_VALUE)));
            layout.setVerticalGroup(      layout.createParallelGroup(javax.swing.
GroupLayout.Alignment.LEADING)
                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,      layout.
createSequentialGroup()
                .addContainerGap(266,Short.MAX_VALUE)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
```

```
                .addComponent(botaoIncluir)
                .addComponent(botaoConsultar)
                .addComponent(botaoAlterar)
                .addComponent(botaoExcluir)
                .addComponent(botaoSair))
                .addContainerGap())
        );
        pack();
    }

    private void botaoSairActionPerformed(java.awt.event.ActionEvent evt) {
        this.setVisible(false);
    }


    protected javax.swing.JButton botaoAlterar;
    protected javax.swing.JButton botaoConsultar;
    protected javax.swing.JButton botaoExcluir;
    protected javax.swing.JButton botaoIncluir;
    private javax.swing.JButton botaoSair;
}
```

Listagem 9

A figura 5.9 é a TelaCadClube que na verdade é um objeto derivado de TelaCad. Os botões Incluir, Consultar, Alterar, Excluir e Sair foram definidos na TelaCad porém agora são implementados para cumprir suas funcionalidades. A Listagem 10 é a implementação da figura 5.9.

O botão Incluir serve para obter os dados dos dois *JTextField*, agrupá-los, criar um objeto da classe Atividade e gravá-lo em um banco de dados. Não estamos contemplando neste exemplo a persistência dos dados, ou seja, a gravação e recuperação no banco de dados, que faz parte da camada de Controle do padrão MVC que já citamos.

No nosso caso, quando o botão Incluir é acionado, o programa apenas cria o objeto e mostra seus dados na tela por meio de um *JOptionPane.showMesssage-Dialog e do método toString()*.

Os outros botões: Alterar, Excluir e Consultar apenas mostram um diálogo para informar que foram acionados. Não foi implementado nenhum código

específico porque eles estão altamente relacionados com a persistência, a qual não iremos tratar aqui neste exemplo.

Perceba que na classe Atividade e na classe Socio foi implementado um método chamado toString(). Este método é muito útil porque ele é uma forma padronizada para poder mostrar os valores das variáveis de instância em formato String. É recomendável que as classes do Modelo implementem este método, desta forma qualquer outro objeto que precisar imprimir os dados em String pode usar este método.

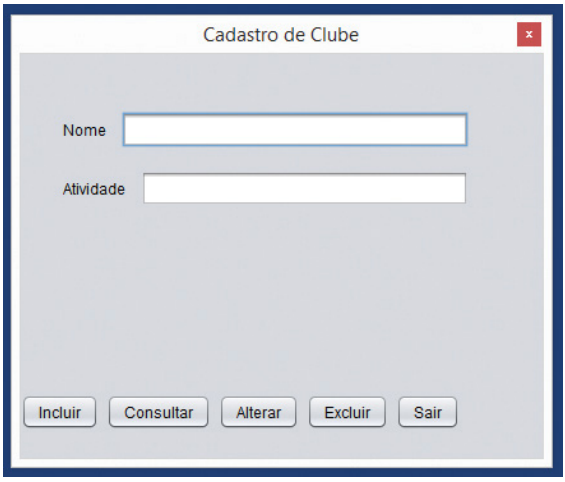


Figura 5.9 – Frame gerado a partir da Listagem 3

```
    import javax.swing.JOptionPane;
    public class TelaCadClube extends TelaCad {
       public TelaCadClube() {
          initComponents();
       }

       @SuppressWarnings("unchecked")
       private void initComponents() {
       jLabel1 = new javax.swing.JLabel();
       jLabel2 = new javax  wing.JLabel();
          txtNome = new javax.swing.JTextField();
          txtAtividade = new javax.swing.JTextField();
          setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
          setTitle("Cadastro de Clube");
          jLabel1.setText("Nome");
          jLabel2.setText("Atividade");
          jLabel2.setToolTipText("");

          javax.swing.GroupLayout      layout      =      new      javax.swing.
GroupLayout(getContentPane());
          getContentPane().setLayout(layout);
          layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
             .addGroup(layout.createSequentialGroup()
                .addGap(33, 33, 33)
    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING, false)
             .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel2)
    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                .addComponent(txtAtividade))
             .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel1)
    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                .addComponent(txtNome,
    javax.swing.GroupLayout.PREFERRED_SIZE, 272,
    javax.swing.GroupLayout.PREFERRED_SIZE)))
                .addContainerGap(58, Short.MAX_VALUE))
          );
```

```
          layout.setVerticalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
              .addGroup(layout.createSequentialGroup()
                  .addGap(45, 45, 45)
    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
BASELINE)
                      .addComponent(jLabel1)
                      .addComponent(txtNome,
    javax.swing.GroupLayout.PREFERRED_SIZE,     javax.swing.GroupLayout.DEFAULT_
SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                  .addGap(18, 18, 18)
    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
BASELINE)
                      .addComponent(jLabel2)
                      .addComponent(txtAtividade,
    javax.swing.GroupLayout.PREFERRED_SIZE,     javax.swing.GroupLayout.DEFAULT_
SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                  .addContainerGap(197, Short.MAX_VALUE))
          );
          botaoIncluir.addActionListener(new java.awt.event.ActionListener() {
              public void actionPerformed(java.awt.event.ActionEvent evt) {
                  botaoIncluirActionPerformed(evt);
              }
          });

          botaoAlterar.addActionListener(new java.awt.event.ActionListener() {
              public void actionPerformed(java.awt.event.ActionEvent evt) {
                  botaoAlterarActionPerformed(evt);
              }
          });

          botaoConsultar.addActionListener(new java.awt.event.ActionListener(){
              public void actionPerformed(java.awt.event.ActionEvent evt) {
                  botaoConsultarActionPerformed(evt);
              }
          });
```

```
            botaoExcluir.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                    botaoExcluirActionPerformed(evt);
                }
            });
            pack();
        }

        void botaoIncluirActionPerformed(java.awt.event.ActionEvent evt) {
            if(txtAtividade.getText()==null || txtAtividade.getText().isEmpty() ||
txtNome.getText()==null || txtNome.getText().isEmpty()){
                JOptionPane.showMessageDialog(null,"Preencha todos os campos");
            }
            else{
                int codigo = Integer.parseInt(txtAtividade.getText());
                String desc = txtNome.getText();
                Atividade ati = new Atividade(codigo,desc);
                JOptionPane.showMessageDialog(null,"Atividade       criada\n"+ati.
toString());
            }
        }

        void botaoAlterarActionPerformed(java.awt.event.ActionEvent evt) {
            JOptionPane.showMessageDialog(null,"Alterar");
        }

        void botaoConsultarActionPerformed(java.awt.event.ActionEvent evt) {
            JOptionPane.showMessageDialog(null,"Consultar");
        }

        void botaoExcluirActionPerformed(java.awt.event.ActionEvent evt) {
            JOptionPane.showMessageDialog(null,"Excluir");
        }
        private javax.swing.JLabel jLabel1;
        private javax.swing.JLabel jLabel2;
        private javax.swing.JTextField txtAtividade;
        private javax.swing.JTextField txtNome;
    }
```

Listagem 10

5.3.2 Criando as classes de negócio

A listagem 11 mostra as duas classes Atividade e Sócio, responsáveis por parte das classes do Modelo do nosso exemplo. Ainda falta a classe Alocação a qual não vamos mostrar aqui por questões de sua simplicidade. Estas duas classes são uma variação dos POJO que seguem o padrão EJB (veja o box), porém neste caso elas não estão implementando a interface *Serializable*.

Plain Old Java Objects (POJO) são objetos Java que tendem a ser os mais simples possíveis, em contraposição ao Enterprise Java Bean (EJB). Este é mais complexo e caracteriza-se por ser um objeto componente da plataforma JEE o qual roda em um *container* de um servidor de aplicação. Normalmente o POJO possui gets e *sets* para cada um de seus atributos, um método construtor sem parâmetros e outro construtor para inicializar os atributos, além de ser serializável, ou seja, pode ser transformado em *bytes* como por exemplo ser gravado em disco. A especificação dos POJOs possui outros detalhes os quais foram omitidos aqui por questão de escopo.

```
    public class Atividade {
        private int codAtv;
        private String descricao;

        public Atividade(int codAtv, String descricao) {
            this.codAtv = codAtv;
            this.descricao = descricao;
        }
        public int getCodAtv() { return codAtv; }
        public void setCodAtv(int codAtv) { this.codAtv = codAtv; }
        public String getDescricao() { return descricao; }
        public void setDescricao(String descricao) { this.descricao = descricao; }

        public String toString() {
            return "Atividade{" + "codAtv=" + codAtv + ", descricao=" + descricao
+ '}';
        }
    }
```

```
public class Socio {
    private int matricula;
    private String nome;

    public Socio(int matricula, String nome) {
        this.matricula = matricula;
        this.nome = nome;
    }

    public int getMatricula() { return matricula; }
    public void setMatricula(int matricula) { this.matricula = matricula; }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }

    @Override
    public String toString() {
        return "Socio{" + "matricula=" + matricula + ", nome=" + nome + '}';
    }
}
```

Listagem 11

O escopo do estudo de caso é este. Poderíamos acrescentar vários outros detalhes de implementação mas estes fogem do escopo proposto por este material e disciplina. Porém, você pode pesquisar sobre isso. O que é preciso para completar este exemplo:

• estudar o *Java Data Base Connectivity* (JDBC). O JDBC é um conjunto de classes que vão servir para fazer a interface com algum banco de dados. O JDBC envia os comandos SQL da aplicação para o banco de dados e desta forma permite que a persistência dos objetos seja feita.

CONEXÃO

Os links a seguir podem te ajudar a conhecer o JDBC. São tutoriais muito bons a respeito do assunto:

http://www.guj.com.br/articles/7

http://docs.oracle.com/javase/tutorial/jdbc/
http://docs.oracle.com/javase/tutorial/jdbc/basics/

• estudar sobre persistência de objetos: isto é importante porque a maioria dos bancos de dados usados nas aplicações atuais são relacionais e possuem um paradigma diferente da orientação a objetos. Sendo assim é importante saber como transformar um objeto em dados relacionais e vice-versa.

• implementar a associação entre os objetos da classe Atividade e da classe Clube e integrar com a classe Alocação. Esses detalhes foram omitidos mas você poderá inferir que dentro de uma alocação teremos que ter uma associação na classe contendo um objeto Sócio e um objeto Atividade.

• criar a tela da Alocação

Por enquanto é só. Porém não deixe de estudar. Lembre-se que nosso objetivo aqui é passar os principais conceitos. O aprimoramento é com você! Vamos fazer alguns exercícios.

ATIVIDADES

01. Liste 5 exemplos de exceções comuns.

02. Por que usamos os blocos finally?

03. Este código está correto?

```
try {

} finally {

}
```

04. Que tipos de exceções podem ser tratadas pelo seguinte tratador?

```
catch (Exception e) {

}
```

O que está errado ao usar este tipo de tratador de exceção?

05. Este código está correto? Ele será devidamente compilado?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

REFLEXÃO

Muitos projetos são executados sem muitos cuidados. Aprender a planejar, conhecer padrões de desenvolvimento bem como a linguagem de programação é importante. Atualmente percebe-se que os programadores tem evoluído em muitos conceitos relacionados com a codificação dos programas e repetem tutoriais e comandos que acham na *internet*. É importante que você aprenda e tire suas conclusões. Compartilhe o conhecimento, participe de fóruns, preze por desenvolvimentos mais inteligentes e produtivos. Todos tem a ganhar com isso.

LEITURA

Neste capítulo tratamos de alguns assuntos que não foram bem explorados porque não é o objetivo da disciplina e foge do escopo. Porém vamos indicar alguns links nos quais você poderá encontrar boas referências e terminar o estudo de caso com “chave de ouro”.

- http://www.gqferreira.com.br/artigos/ver/mvc-com-java-desktop-parte1: artigo dividido em 3 partes contendo dicas para o desenvolvimento de aplicações para *desktop* e o padrão MVC
- http://www.yaw.com.br/open/projetos/swing-jdbc-crud/: projeto de código aberto que trata de aplicações envolvendo banco de dados e *Swing*. Muito interessante!
- http://wiki.netbeans.org/JavaPersistenceApi: Trata sobre a biblioteca de persistência em Java
- https://platform.netbeans.org/tutorials/nbm-crud.html: Tutorial que mostra a criação de uma aplicação com banco de dados

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Bookman, 2002.

FLANAGAN, D. **Java o guia essencial**. Bookman, 2006.

GARY, C.; HORSTMANN, C. **Core Java 2**: Fundamentos. Makron Books, 2001.

GARY, C.; HORSTMANN, C. **Core Java 2**: Recursos Avançados. Makron Books, 2001.

HUBBARD, J. R. **Programação com Java**. Bookman, 2006.

GABARITO

Capítulo 1

01.

```
    import javax.swing.JOptionPane;

    public class Exercicio {
       public static void main(String[] args){
          String entrada,saida;
          int      numeroConta,      saldo,      totalItens,      totalCreditos,
limiteCredito,novoSaldo;
          entrada = JOptionPane.showInputDialog("Digite o numero da conta");
          numeroConta = Integer.parseInt(entrada);
          entrada = JOptionPane.showInputDialog("Digite o saldo");
          saldo = Integer.parseInt(entrada);
```

```
            entrada = JOptionPane.showInputDialog("Digite o total de itens");
            totalItens = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o total de creditos");
            totalCreditos = Integer.parseInt(entrada);

            entrada = JOptionPane.showInputDialog("Digite o limite");
            limiteCredito = Integer.parseInt(entrada);

            novoSaldo = saldo + totalItens - totalCreditos;

            if (novoSaldo > limiteCredito){
                JOptionPane.showMessageDialog(null, "O cliente da conta "+numero-
Conta+" excedeu o limite!");
            }
            else {
                JOptionPane.showMessageDialog(null, "O cliente da conta "+numero-
Conta+" NAO excedeu o limite!");
            }
        }
    }
```

02.

a)

```
    Segunda string
    Terceira string
```

b)

```
    public class Exercicio {
        public static void main(String[] args){
            int umNumero = 3;
            if (umNumero >= 0)
                if (umNumero == 0)
                    System.out.println("Primeira string");
```

```
        else System.out.println("Segunda string");
        System.out.println("Terceira string");
    }
}
```

c)

```
public class Exercicio {
    public static void main(String[] args){
        int umNumero = 3;
        if (umNumero >= 0)
            if (umNumero == 0)
                System.out.println("Primeira string");
        else
                System.out.println("Segunda string");
        System.out.println("Terceira string");
    }
}
```

d)

```
public class Exercicio {
    public static void main(String[] args){
        int umNumero = 3;
        if (umNumero >= 0){
            if (umNumero == 0)
                System.out.println("Primeira string");
        }
        else {
                System.out.println("Segunda string");
        }
        System.out.println("Terceira string");
    }
}
```

03.

a) 32
b) e
c) hannah.charAt(15)

04.

```
slá
oeá
owlá
oláoft
oláar
```

05.

Não “reinvente a roda”. Procure na API do Java por métodos já existentes. Vamos usar a classe Arrays:

```
import java.util.Arrays;
import javax.swing.JOptionPane;

public class Exercicio {
   public static void main(String[] args) {
       String s = JOptionPane.showInputDialog("Entre com a palavra 1");
       String r = JOptionPane.showInputDialog("Entre com a palavra 2");
       char[] a = s.toCharArray();
       char[] b = r.toCharArray();
       Arrays.sort(a);
       Arrays.sort(b);
       if (Arrays.equals(a, b)) {
           JOptionPane.showMessageDialog(null, "É anagrama");
       } else {
           JOptionPane.showMessageDialog(null, "NÃO É anagrama");
       }
   }
}
```

Capítulo 2

01.

a) x e y

b)

```
a.y = 5
b.y = 6
a.x = 1
b.x = 2
Erro pois a classe Exercicio1 não é estática
```

02.

```
public class Pessoa {
    private int idade;
    private String nome;

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public String toString() {
        return "Exercicio1 [idade=" + idade + ", nome=" + nome + "]";
    }

    public void fazAniversario(){
        this.idade++;
    }
}
```

Teste e objetos:

```
    Pessoa obj1 = new Pessoa();
    Pessoa obj2 = new Pessoa();

    obj1.setNome("Fabiano");
    obj1.setIdade(42);
    JOptionPane.showMessageDialog(null, "Antes do Aniversário\n"+obj1.toString());
    obj1.fazAniversario();
    JOptionPane.showMessageDialog(null,    "Depois    do    aniversário\n"+obj1.
toString());
    obj2.setNome("Vinicius");
    obj2.setIdade(9);
    JOptionPane.showMessageDialog(null, "Antes do aniversário\n"+obj2.toString());
    obj2.fazAniversario();
    JOptionPane.showMessageDialog(null,    "Depois    do    aniversário\n"+obj2.
toString());
```

03.

```
    import java.util.Scanner;

    public class Conta {
       private String nome;
       private int conta, saques;
       private double saldo;
       Scanner entrada = new Scanner(System.in);

       public Conta(String nome, int conta, double saldo_inicial){
           this.nome=nome;
           this.conta=conta;
           saldo=saldo_inicial;
           saques=0;
       }
       public void extrato(){
           System.out.println("\tEXTRATO");
           System.out.println("Nome: " + this.nome);
           System.out.println("Número da conta: " + this.conta);
           System.out.printf("Saldo atual: %.2f\n",this.saldo);
```

```
        System.out.println("Saques realizados hoje: " + this.saques + "\n");

    }

    public void sacar(double valor){
        if(saldo >= valor){
            saldo -= valor;
            saques++;
            System.out.println("Sacado: " + valor);
            System.out.println("Novo saldo: " + saldo + "\n");
        } else {
            System.out.println("Saldo insuficiente. Faça um depósito\n");
        }
    }

    public void depositar(double valor) {
        saldo += valor;
        System.out.println("Depositado: " + valor);
        System.out.println("Novo saldo: " + saldo + "\n");
    }

    public void iniciar(){
        int opcao;

        do{
            exibeMenu();
            opcao = entrada.nextInt();
            escolheOpcao(opcao);
        }while(opcao!=4);
    }

    public void exibeMenu(){

        System.out.println("\t Escolha a opção desejada");
        System.out.println("1 - Consultar Extrato");
        System.out.println("2 - Sacar");
        System.out.println("3 - Depositar");
        System.out.println("4 - Sair\n");
        System.out.print("Opção: ");
    }
```

```
    public void escolheOpcao(int opcao){
        double valor;
        switch( opcao ){
            case 1:
                extrato();
                break;
            case 2:
                if(saques<3){
                    System.out.print("Quanto deseja sacar: ");
                    valor = entrada.nextDouble();
                    sacar(valor);
                } else{
                    System.out.println("Limite de saques diários atingidos.\n");
                }
                break;

            case 3:
                System.out.print("Quanto deseja depositar: ");
                valor = entrada.nextDouble();
                depositar(valor);
                break;

            case 4:
                System.out.println("Sistema encerrado.");
                break;

            default:
                System.out.println("Opção inválida");
        }
    }
}
```

04.

```
import java.util.List;

public class Pais {

    private String descricao;
```

```
private String capital;
private String tamanho;
private List<Pais> vizinhos;

public Pais(String descricao, String capital, String tamanho,
       List<Pais> vizinhos) {
    super();
    this.descricao = descricao;
    this.capital = capital;
    this.tamanho = tamanho;
    this.vizinhos = vizinhos;
}

public Pais(String descricao, String capital, String tamanho) {
    super();
    this.descricao = descricao;
    this.capital = capital;
    this.tamanho = tamanho;
}



public String getDescricao() {
    return descricao;
}

public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public String getCapital() {
    return capital;
}

public void setCapital(String capital) {
    this.capital = capital;
}
```

```
    public String getTamanho() {
        return tamanho;
    }

    public void setTamanho(String tamanho) {
        this.tamanho = tamanho;
    }

    public List<Pais> getVizinhos() {
        return vizinhos;
    }

    public void setVizinhos(List<Pais> vizinhos) {
        this.vizinhos = vizinhos;
    }
}

import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {
        Pais uruguai = new Pais("Uruguai", "Monte", "pequeno");
        Pais argentina = new Pais("Argentina", "Buenos", "médio");
        ArrayList<Pais> vizinhos = new ArrayList<>();
        vizinhos.add(argentina);
        vizinhos.add(uruguai);
        Pais brasil = new Pais("Brasil", "Brasilia", "Grande", vizinhos);

    }
}
```

Capítulo 3

```
public class Produto {
    private int codigo;
    private String descricao;
    private Data validade;
```

```
    public Produto(int codigo, String descricao, Data validade) {
        super();
        this.codigo = codigo;
        this.descricao = descricao;
        this.validade = validade;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public Data getValidade() {
        return validade;
    }

    public void setValidade(Data validade) {
        this.validade = validade;
    }
}

public class Data {
    private int dia;
    private int mes;
    private int ano;
    private Produto produto;
        public Data(int dia, int mes, int ano, Produto produto) {
        this.dia = dia;
        this.mes = mes;
```

```
        this.ano = ano;
        this.produto = produto;
    }

    public int getDia() {
        return dia;
    }

    public void setDia(int dia) {
        this.dia = dia;
    }

    public int getMes() {
        return mes;
    }

    public void setMes(int mes) {
        this.mes = mes;
    }

    public int getAno() {
        return ano;
    }

    public void setAno(int ano) {
        this.ano = ano;
    }

    public Produto getProduto() {
        return produto;
    }

    public void setProduto(Produto produto) {
        this.produto = produto;
    }

    public String extenso() {
        return dia + " de " + mes + " de " + ano;
    }
}
```

01.

```
    import java.awt.Container;
    import java.awt.event.ActionEvent;
    import java.awt.event.ActionListener;
    import java.util.Locale;
    import javax.print.DocFlavor;
    import javax.swing.AbstractAction;
    import javax.swing.JButton;
    import javax.swing.JFrame;
    import static javax.swing.JFrame.EXIT_ON_CLOSE;
    import javax.swing.JLabel;
    import javax.swing.JOptionPane;
    import javax.swing.JTextField;

    public class Calculadora extends JFrame implements ActionListener{

       private JButton btnResult,btnLimpar, btn0,btn1, btn2, btn3, btn4, btn5,
btn6, btn7, btn8, btn9, btnmais, btnmenos, btndivide, btnmult, btnponto;

       private JLabel  resultado;
       String oOperador;

       String primeiro ="", segundo="", sinal ="";
       int cont =1 ;
       public Calculadora(){
          super("Calculadora");
          this.setDefaultCloseOperation(EXIT_ON_CLOSE);
          this.setSize(250, 380);
          this.setLocationRelativeTo(null);

          Container c = this.getContentPane();
          c.setLayout(null);

          final JTextField tbxPrincipal = new JTextField("");
          tbxPrincipal.setBounds(20, 40, 198, 20);
          c.add(tbxPrincipal);
```

```
btn7 = new JButton("7");
btn7.setBounds(20, 80, 45, 45);
btn7.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "7";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn7);
btn8 = new JButton("8");
btn8.setBounds(70, 80, 45, 45);
btn8.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "8";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn8);
btn9 = new JButton("9");
btn9.setBounds(120, 80, 45, 45);
btn9.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "9";
        if(cont == 1){
            primeiro = primeiro + n;
```

```
            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn9);
btn4 = new JButton("4");
btn4.setBounds(20, 130, 45, 45);
btn4.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "4";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn4);
btn5 = new JButton("5");
btn5.setBounds(70, 130, 45, 45);
btn5.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "5";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
```

```
});
c.add(btn5);
btn6 = new JButton("6");
btn6.setBounds(120, 130, 45, 45);
btn6.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "6";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn6);
btn1 = new JButton("1");
btn1.setBounds(20, 180, 45, 45);
btn1.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "1";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btn1);
btn2 = new JButton("2");
btn2.setBounds(70, 180, 45, 45);
btn2.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = "2";
```

```
            if(cont == 1){
                primeiro = primeiro + n;

                tbxPrincipal.setText(primeiro);
            } else if(cont==2) {
                segundo = segundo + n;
                tbxPrincipal.setText(segundo);
            }
        }
    });
    c.add(btn2);
    btn3 = new JButton("3");
    btn3.setBounds(120, 180, 45, 45);
    btn3.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String n = "3";
            if(cont == 1){
                primeiro = primeiro + n;
                tbxPrincipal.setText(primeiro);
            } else if(cont==2) {
                segundo = segundo + n;
                tbxPrincipal.setText(segundo);
            }
        }
    });
    c.add(btn3);
    btn0 = new JButton("0");
    btn0.setBounds(20, 230, 45, 45);
    btn0.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            String n = "0";
            if(cont == 1){
                primeiro = primeiro + n;
                tbxPrincipal.setText(primeiro);
            } else if(cont==2) {
                segundo = segundo + n;
                tbxPrincipal.setText(segundo);
            }
        }
```

```
});
c.add(btn0);
this.setVisible(true);
btnponto = new JButton(".");
btnponto.setBounds(70, 230, 45, 45);
btnponto.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        String n = ".";
        if(cont == 1){
            primeiro = primeiro + n;

            tbxPrincipal.setText(primeiro);
        } else if(cont==2) {
            segundo = segundo + n;
            tbxPrincipal.setText(segundo);
        }
    }
});
c.add(btnponto);
btnResult = new JButton("=");
btnResult.setBounds(120, 230, 45, 45);
btnResult.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e) {
        double soma =0;
        if(sinal == "+"){
            soma=Double.valueOf(primeiro)+ Double.valueOf(segundo);
            tbxPrincipal.setText(String.valueOf(soma));
            cont = 1;
            primeiro = "";
            segundo ="";
        } else  if(sinal == "-"){
            soma=Double.valueOf(primeiro)- Double.valueOf(segundo);

            tbxPrincipal.setText(String.valueOf(soma));
            cont = 1;
            primeiro = "";
            segundo ="";
        } else if(sinal == "*"){
            soma=Double.valueOf(primeiro)- Double.valueOf(segundo);
```

```
                tbxPrincipal.setText(String.valueOf(soma));
                cont = 1;
                primeiro = "";
                segundo ="";
            } else if(sinal == "*"){
                soma=Double.valueOf(primeiro)* Double.valueOf(segundo);
                tbxPrincipal.setText(String.valueOf(soma));
                cont = 1;
                primeiro = "";
                segundo ="";
            }else if(sinal == "/"){
                soma=Double.valueOf(primeiro)/ Double.valueOf(segundo);

                tbxPrincipal.setText(String.valueOf(soma));
                cont = 1;
                primeiro = "";
                segundo ="";
            } else {
                tbxPrincipal.setText("Sem valores para efetuar funções");
            }
        }
    });
    c.add(btnResult);
    btndivide = new JButton("/");
    btndivide.setBounds(170, 80, 45, 45);
    btndivide.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String valor = tbxPrincipal.getText();
            tbxPrincipal.setText(valor + "/");

            sinal ="/";
            cont =2;
        }
    });
    c.add(btndivide);
    btnmult = new JButton("*");
    btnmult.setBounds(170, 130, 45, 45);
    btnmult.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
```

```
            String valor = tbxPrincipal.getText();
            tbxPrincipal.setText(valor + "*");

            sinal ="*";
            cont =2;
        }
    });
    c.add(btnmult);
    btnmenos = new JButton("-");
    btnmenos.setBounds(170, 180, 45, 45);
    btnmenos.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String valor = tbxPrincipal.getText();
            tbxPrincipal.setText(valor + "-");

            sinal ="-";
            cont =2;
        }
    });
    c.add(btnmenos);
    btnmais = new JButton("+");
    btnmais.setBounds(170, 230, 45, 45);
    btnmais.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String valor = tbxPrincipal.getText();
            tbxPrincipal.setText(valor + "+");

            sinal ="+";
            cont =2;
        }
    });
    c.add(btnmais);
    btnLimpar = new JButton("CEE");
    btnLimpar.setBounds(20, 280, 85, 45);
    btnLimpar.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            String valor = tbxPrincipal.getText();
            tbxPrincipal.setText("");
```

```
                sinal ="CEE";
                cont =1;
            }
        });
        c.add(btnLimpar);
    }

    public static void main(String[] args) {
        Calculadora f = new Calculadora();
    }
    public void actionPerformed(ActionEvent e) {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
}
```

02.

Feito no Netbeans

```
public class Cadastro extends javax.swing.JFrame {
    public Cadastro() {
        initComponents();
    }

@SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
        jLabel3 = new javax.swing.JLabel();
        jLabel4 = new javax.swing.JLabel();
        jLabel5 = new javax.swing.JLabel();
        jLabel6 = new javax.swing.JLabel();
        jLabel7 = new javax.swing.JLabel();
        jLabel8 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jButton1 = new javax.swing.JButton();
```

```
        jTextField2 = new javax.swing.JTextField();
        jTextField3 = new javax.swing.JTextField();
        jTextField4 = new javax.swing.JTextField();
        jTextField5 = new javax.swing.JTextField();
        jTextField6 = new javax.swing.JTextField();
        jTextField7 = new javax.swing.JTextField();
        jButton2 = new javax.swing.JButton();
        jButton3 = new javax.swing.JButton();
        jButton4 = new javax.swing.JButton();
        jButton5 = new javax.swing.JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jLabel1.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
        jLabel1.setText("Desenvolvimento Aberto");
        jLabel2.setText("Pesquisa código:");
        jLabel3.setText("Código:");
        jLabel4.setText("Primeiro Nome:");
        jLabel5.setText("Sobrenome:");
        jLabel6.setText("Cargo:");
        jLabel7.setText("Salário");
        jLabel8.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
        jLabel8.setText("Porcentagem (%):");

        jTextField1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jTextField1ActionPerformed(evt);
        }
        });

        jButton1.setText("Pesquisar");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        jButton2.setText("Novo");
```

```
          jButton3.setText("Inserir");
          jButton4.setText("Alterar");
          jButton5.setText("Apagar");
          javax.swing.GroupLayout      layout      =      new      javax.swing.
GroupLayout(getContentPane());
          getContentPane().setLayout(layout);
          layout.setHorizontalGroup(
             layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
             .addGroup(layout.createSequentialGroup()
             .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.LEADING)
             .addGroup(layout.createSequentialGroup()
             .addGap(79, 79, 79)
             .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.TRAILING)
                .addComponent(jLabel4)
                .addComponent(jLabel3)
                .addComponent(jLabel5)
                .addComponent(jLabel6)
                .addComponent(jLabel7)
                .addComponent(jLabel8))
             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
             .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.LEADING, false)
                .addComponent(jTextField3)
                .addComponent(jTextField4)
                .addComponent(jTextField5)
                .addComponent(jTextField6)
                .addComponent(jTextField7)
                .addComponent(jTextField2,    javax.swing.GroupLayout.DEFAULT_
SIZE, 94, Short.MAX_VALUE)))
             .addGroup(layout.createSequentialGroup()
             .addGap(42, 42, 42)
             .addComponent(jLabel2)
             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
```

```
                .addComponent(jTextField1,   javax.swing.GroupLayout.PREFERRED_
SIZE, 156, javax.swing.GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                .addComponent(jButton1))
                .addGroup(layout.createSequentialGroup()
                .addGap(99, 99, 99)
                .addComponent(jLabel1))
                .addGroup(layout.createSequentialGroup()
                .addContainerGap()
                .addComponent(jButton2)
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
            .addComponent(jButton3)
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                .addComponent(jButton4)
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
                .addComponent(jButton5)))
                .addContainerGap(33, Short.MAX_VALUE))
            );
            layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
                .addGroup(layout.createSequentialGroup()
                .addContainerGap()
                .addComponent(jLabel1)
                .addGap(36, 36, 36)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel2)
                .addComponent(jTextField1,            javax.swing.GroupLayout.
PREFERRED_SIZE,  javax.swing.GroupLayout.DEFAULT_SIZE,  javax.swing.GroupLayout.
PREFERRED_SIZE)
            .addComponent(jButton1))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
UNRELATED)
```

```
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel3)
                .addComponent(jTextField2,
    javax.swing.GroupLayout.PREFERRED_SIZE,     javax.swing.GroupLayout.DEFAULT_
SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel4)
                .addComponent(jTextField3,
    javax.swing.GroupLayout.PREFERRED_SIZE,     javax.swing.GroupLayout.DEFAULT_
SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel5)
                .addComponent(jTextField4,            javax.swing.GroupLayout.
PREFERRED_SIZE,  javax.swing.GroupLayout.DEFAULT_SIZE,  javax.swing.GroupLayout.
PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel6)
                .addComponent(jTextField5,            javax.swing.GroupLayout.
PREFERRED_SIZE,  javax.swing.GroupLayout.DEFAULT_SIZE,  javax.swing.GroupLayout.
PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
RELATED)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel7)
                .addComponent(jTextField6,            javax.swing.GroupLayout.
PREFERRED_SIZE,  javax.swing.GroupLayout.DEFAULT_SIZE,  javax.swing.GroupLayout.
PREFERRED_SIZE))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
UNRELATED)
```

```
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jLabel8)
                .addComponent(jTextField7,            javax.swing.GroupLayout.
PREFERRED_SIZE,  javax.swing.GroupLayout.DEFAULT_SIZE,  javax.swing.GroupLayout.
PREFERRED_SIZE))
                .addGap(18, 18, 18)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
Alignment.BASELINE)
                .addComponent(jButton2)
                .addComponent(jButton3)
                .addComponent(jButton4)
                .addComponent(jButton5))
            .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,      Short.
MAX_VALUE))
        );

        pack();
    }// </editor-fold>

    private void jTextField1ActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    public static void main(String args[]) {
        try {
            for  (javax.swing.UIManager.LookAndFeelInfo  info  :  javax.swing.
UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
```

```
            } catch (ClassNotFoundException ex) {
                java.util.logging.Logger.getLogger(Cadastro.class.getName()).log(-
java.util.logging.Level.SEVERE, null, ex);
            } catch (InstantiationException ex) {
                java.util.logging.Logger.getLogger(Cadastro.class.getName()).log(-
java.util.logging.Level.SEVERE, null, ex);
            } catch (IllegalAccessException ex) {
                java.util.logging.Logger.getLogger(Cadastro.class.getName()).log(-
java.util.logging.Level.SEVERE, null, ex);
            } catch (javax.swing.UnsupportedLookAndFeelException ex) {
                java.util.logging.Logger.getLogger(Cadastro.class.getName()).log(-
java.util.logging.Level.SEVERE, null, ex);
            }
            //</editor-fold>

            /* Create and display the form */
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
                    new Cadastro().setVisible(true);
                }
            });
        }

        private javax.swing.JButton jButton1;
        private javax.swing.JButton jButton2;
        private javax.swing.JButton jButton3;
        private javax.swing.JButton jButton4;
        private javax.swing.JButton jButton5;
        private javax.swing.JLabel jLabel1;
        private javax.swing.JLabel jLabel2;
        private javax.swing.JLabel jLabel3;
        private javax.swing.JLabel jLabel4;
        private javax.swing.JLabel jLabel5;
        private javax.swing.JLabel jLabel6;
        private javax.swing.JLabel jLabel7;
        private javax.swing.JLabel jLabel8;
        private javax.swing.JTextField jTextField1;
        private javax.swing.JTextField jTextField2;
        private javax.swing.JTextField jTextField3;
```

```
        private javax.swing.JTextField jTextField4;
        private javax.swing.JTextField jTextField5;
        private javax.swing.JTextField jTextField6;
        private javax.swing.JTextField jTextField7;
        // End of variables declaration
    }
```

03.

```
    import javax.swing.JOptionPane;

    public class Main {
        public static void main(String[] args) {
            String stringCelsius = JOptionPane.showInputDialog (null,"Temperatu-
ra " + " em Celsius", "Calculadora de " + "Graus Fahrenheit", JOptionPane.
QUESTION_MESSAGE);
            double celsius = Double.parseDouble(stringCelsius);
            // convert fahrenheit into celcius and store it in variable
            double fahrenheit = celsius*1.8+32;
            JOptionPane.showMessageDialog(null, celsius + " graus celsius" + "
é igual a " + fahrenheit + " °F.", "Calculadora de "+ "Graus Fahrenheit",
JOptionPane.QUESTION_MESSAGE);
        }
    }
```

Capítulo 5

01.

IOException
SQLException
RunTimeException
IllegalArgumentException
NullPointerException

02.

O bloco finally sempre executa seu bloco de dados mesmo que uma exceção seja lançada. É útil para liberar recursos do sistema quando utilizamos, por exemplo, conexões de banco de dados e abertura de buffer para leitura ou escrita de arquivos.

03.

Sim, está correto. Um try não precisa ter um bloco catch se existir um finally. Se o código no tryu tiver vários pontos de saída e não há catch associado, o código no bloco finally é executado independente da forma que o bloco try sai.

04.

Este tratador pega exceções do tipo Exception, ou seja, qualquer tipo de exceção. Isto não é muito aconselhável porque o usuário fica sem saber qual foi o erro na execução do programa.

05.

O primeiro tratador pega exceções do tipo Exception, incluindo as ArithmeticException. O segundo tratador nunca será executado e portanto o código não será compilado.

ANOTAÇÕES