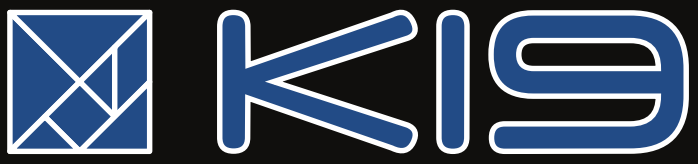


TREINAMENTOS

Orientação a Objetos em Java

Orientação a Objetos em Java

17 de novembro de 2010

Sumário

1 Introdução **1**

2 Lógica **3**
2.1 O que é um Programa? . . . 3
2.2 Linguagem de Programação VS Linguagem de Máquina . . . 3
2.3 Exemplo de programa Java . . . 4
2.4 Método Main - Ponto de Entrada . . . 5
2.5 Máquinas Virtuais . . . 6
2.6 Exercícios . . . 9
2.7 Variáveis . . . 10
2.8 Operadores . . . 13
2.9 IF-ELSE . . . 14
2.10 WHILE . . . 15
2.11 FOR . . . 15
2.12 Exercícios . . . 15

3 Orientação a Objetos **19**
3.1 Objetos . . . 19
3.2 Classes . . . 20
3.3 Referências . . . 21
3.4 Manipulando Atributos . . . 22
3.5 Agregação . . . 22
3.6 Exercícios . . . 23
3.7 Métodos . . . 27
3.8 Sobrecarga(Overloading) . . . 29
3.9 Exercícios . . . 30
3.10 Construtores . . . 31
3.10.1 Construtor Default . . . 33
3.10.2 Sobrecarga de Construtores . . . 33
3.10.3 Construtores chamando Construtores . . . 34
3.11 Exercícios . . . 34
3.12 Referências como parâmetro . . . 37
3.13 Exercícios . . . 37

4 Arrays **39**
4.1 Arrays de Arrays . . . 40
4.2 Percorrendo Arrays . . . 40
4.3 Operações . . . 41
4.4 Exercícios . . . 41

5 Eclipse **45**
5.1 Workspace . . . 45
5.2 Welcome . . . 46
5.3 Workbench . . . 46
5.4 Perspective . . . 47
5.5 Views . . . 48
5.6 Criando um projeto java . . . 48
5.7 Criando uma classe . . . 51
5.8 Criando o método main . . . 53
5.9 Executando uma classe . . . 54
5.10 Corrigindo erros . . . 55
5.11 Atalhos Úteis . . . 56
5.12 Save Actions . . . 56

6 Atributos e Métodos de Classe **59**
6.1 Atributos Estáticos . . . 59
6.2 Métodos Estáticos . . . 60
6.3 Exercícios . . . 60

7 Encapsulamento **63**
7.1 Atributos Privados . . . 63
7.2 Métodos Privados . . . 64
7.3 Métodos Públicos . . . 64
7.4 Implementação e Interface de Uso . . . 65
7.5 Escondendo a implementação . . . 65
7.6 Acesso e Alteração de atributos . . . 66
7.7 Exercícios . . . 67

8 Herança **71**
8.1 Reutilização de Código . . . 71
8.2 Reescrita de Método . . . 75
8.3 Construtores e Herança . . . 77
8.4 Exercícios . . . 77

9 Polimorfismo **83**
9.1 Modelagem das contas . . . 84
9.2 É UM (extends) . . . 85
9.3 Melhorando o gerador de extrato . . . 85
9.4 Exercícios . . . 86

10 Classes Abstratas **89**
10.1 Classes Abstratas . . . 89
10.2 Métodos Abstratos . . . 90
10.3 Exercícios . . . 91

11 Interfaces **93**
11.1 Padronização . . . 93
11.2 Contratos . . . 94
11.3 Exemplo . . . 94
11.4 Polimorfismo . . . 95
11.5 Interface e Herança . . . 95
11.6 Exercícios . . . 97

12 Pacotes **101**
12.1 Organização . . . 101
12.2 O comando package . . . 101
12.3 Sub Pacotes . . . 101
12.4 Classes ou Interfaces públicas . . . 102
12.4.1 Fully Qualified Name . . . 102
12.5 Import . . . 103
12.6 Níveis de visibilidade . . . 104
12.6.1 Privado . . . 104
12.6.2 Padrão . . . 104
12.6.3 Protegido . . . 104
12.6.4 Público . . . 104
12.7 Exercícios . . . 104

13 Exceptions **107**
13.1 Tipos de erros de execução . . . 108
13.2 Lançando erros . . . 108
13.2.1 Checked e Unchecked . . . 108
13.3 Capturando erros . . . 109
13.4 Exercícios . . . 110

14 Object **113**
14.1 Polimorfismo . . . 113
14.2 O método toString() . . . 114
14.3 O método equals() . . . 115
14.4 Exercícios . . . 116

15 Entrada e Saída **117**
15.1 Byte a Byte . . . 117
15.2 Scanner . . . 118
15.3 PrintStream . . . 119
15.4 Exercícios . . . 119

16 Collections **121**
16.1 Listas . . . 121
16.1.1 Método: add(Object) . . . 122
16.1.2 Método: add(int, Object) . . . 122
16.1.3 Método: size() . . . 122
16.1.4 Método: clear() . . . 123
16.1.5 Método: contains(Object) . . . 123
16.1.6 Método: remove(Object) . . . 123
16.1.7 Método: remove(int) . . . 124
16.1.8 Método: get(int) . . . 124
16.1.9 Método: indexOf(Object) . . . 124
16.1.10 Benchmarking . . . 125
16.2 Exercícios . . . 125
16.3 Conjuntos . . . 127
16.4 Coleções . . . 127
16.5 Exercícios . . . 128
16.6 Laço foreach . . . 129
16.7 Generics . . . 129
16.8 Exercícios . . . 130

17 Apêndice -Swing **133**
17.1 Componentes . . . 133
17.1.1 JFrame . . . 133
17.1.2 JPanel . . . 134
17.1.3 JTextField e JLabel . . . 134
17.1.4 JTextArea . . . 135
17.1.5 JPasswordField . . . 135
17.1.6 JButton . . . 136
17.1.7 JCheckBox . . . 136
17.1.8 JComboBox . . . 136
17.2 Layout Manager . . . 137
17.3 Events, Listeners e Sources . . . 138
17.3.1 Exemplo . . . 138
17.4 Exercícios . . . 139

18 Apêndice -Threads **141**
18.1 Definindo Tarefas - (Runnables) . . . 141
18.2 Executando Tarefas . . . 142
18.3 Exercícios . . . 142
18.4 Controlando a Execução das Tarefas . . . 143
18.4.1 sleep() . . . 143
18.4.2 join() . . . 144
18.5 Exercícios . . . 144

19 Apêndice - Socket **147**
19.1 Socket 147
19.2 ServerSocket 147
19.3 Exercícios 148

20 Apêndice - Chat K19 **151**
20.1 Arquitetura do Sistema 151
20.2 Aplicação servidora 151
20.2.1 Registrador 151
20.2.2 Receptor 151
20.2.3 Emissor 152
20.2.4 Distribuidor 152
20.3 Aplicação cliente 152
20.3.1 EmissorDeMensagem 152
20.3.2 ReceptorDeMensagem 152
20.4 Exercícios 153

Capítulo 1

Introdução

O principal objetivo deste treinamento é capacitar profissionais para atuar no mercado de trabalho na área de desenvolvimento de software. Conhecimentos sobre a plataforma Java e sobre o modelo de programação orientado a objetos são normalmente exigidos pelas empresas que desejam contratar. Portanto, os dois tópicos principais deste treinamento são: a plataforma Java e orientação a objetos. Resumidamente, apresentaremos a seguir cada um desses assuntos e as relações entre eles.

Orientação a objetos é um modelo de programação ou paradigma de programação, ou seja, é um conjunto de ideias, conceitos e abstrações que servem como um guia para construir um software.

A plataforma Java é composta por vários elementos. Neste momento os mais importantes são: a linguagem de programação e o ambiente de execução.

A linguagem de programação é utilizada para definir formalmente todas as partes de um sistema e como essas partes se relacionam. Além disso, a linguagem de programação Java é orientada a objetos, ou seja, ela permite que você aplique as ideias, conceitos e abstrações de orientação a objetos de uma maneira natural.

O ambiente de execução é composto basicamente de uma máquina virtual e de um conjunto de bibliotecas padronizado. A máquina virtual permite que um programa Java possa ser executado em ambiente diferentes. As bibliotecas facilitam a implementação dos sistemas.

O processo de desenvolver um sistema orientado a objetos em Java se dá da seguinte forma: As partes do sistema e o relacionamento entre elas são definidas utilizando o modelo de programação orientado a objetos e depois o código fonte do sistema é escrito na linguagem de programação Java.

Do ponto de vista do aprendizado, é interessante tentar definir o grau de importância dos dois assuntos principais abordados neste treinamento. Consideramos que a orientação a objetos é mais importante pois ela é utilizada frequentemente no desenvolvimento de software inclusive com outras linguagens de programação (C#, JavaScript, Ruby, C++, entre outras). As pessoas que possuem conhecimentos mais sólidos em orientação a objetos são mais preparadas para desenvolver sistemas na maioria das linguagens utilizadas pelas empresas que desenvolvem software.

Para compreender melhor o conteúdo desse treinamento, é importante saber para quais tipos de sistemas o modelo de programação orientado a objetos e a plataforma Java são mais adequados. As características desses sistemas são:

- Possui uma grande quantidade de funcionalidades, sendo necessária uma equipe de de-

senvolvedores para desenvolver e manter o funcionamento do sistema.

- Será utilizado por muito tempo e sofrerá alterações com o tempo.

Esse tipo de sistema é justamente o que muitas empresas que desenvolvem software estão interessadas em desenvolver. Por isso, elas acabam adotando orientação a objetos e Java.

Para que sistemas desse tipo sejam bem sucedidos algumas qualidades são necessárias. Durante o treinamento, discutiremos quais são essas qualidades e como consegui-las.

Capítulo 2

Lógica

2.1 O que é um Programa?

Um dos maiores benefícios da utilização de computadores é a automatização de processos que antes deveriam ser realizados por pessoas. Vejamos um exemplo prático:

Quando as apurações dos votos das eleições eram realizadas manualmente, o tempo para obter os resultados era alto e havia grandes chances de ocorrer uma falha humana. Esse processo foi automatizado e hoje é realizado por computadores. O tempo para obter os resultados e a chance de ocorrer uma falha humana diminuíram bastante.

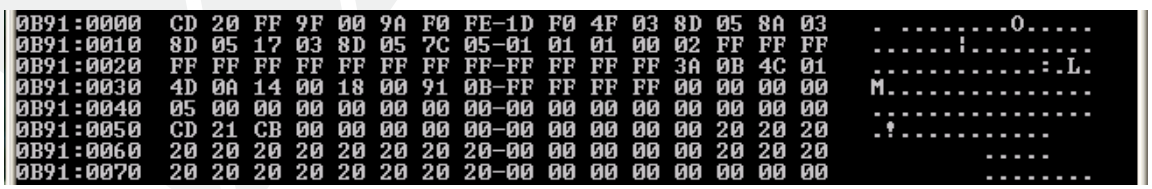
Os computadores são capazes de executar instruções. Com essa capacidade eles podem resolver tarefas complexas. Porém, eles não são suficientemente inteligentes para definir quais instruções devem ser executadas para resolver uma determinada tarefa. Em outras palavras, um computador sabe executar comandos mas não sabe quais comandos devem ser executados para solucionar um problema. Então, uma pessoa precisa criar um **roteiro** com as instruções que devem ser executadas pelo computador para que uma determinada tarefa seja realizada.

Esses roteiros devem ser colocados em arquivos no disco rígido dos computadores. Assim, quando as tarefas precisam ser realizadas, os computadores podem ler esses arquivos para saber quais instruções devem ser executadas.

Um arquivo contendo um roteiro de instruções que devem ser realizadas por um computador para resolver uma determinada tarefa é chamado de **programa** ou **executável**.

2.2 Linguagem de Programação VS Linguagem de Máquina

Os computadores só sabem ler programas escritos em **Linguagem de Máquina**. Um programa escrito em **Linguagem de Máquina** é uma sequência de números que representam as instruções que um computador deve executar.



Teoricamente, os programadores poderiam escrever os programas já em linguagem de

máquina. Na prática, ninguém faz isso pois é uma tarefa muito complicada e demorada. É muito complicado para uma pessoa ler ou escrever um programa escrito em linguagem de máquina.

Ao invés disso, os programadores escrevem os programas utilizando alguma **Linguagem de Programação**. As linguagens de programação tentam ser mais próximas das linguagens humanas. Para uma pessoa, é mais fácil ler um programa escrito em linguagem de programação.

Os programas escritos pelos programadores em alguma linguagem de programação são "traduzidos" para programas escritos em linguagem de máquina para que os computadores possam executá-los. Quem faz essa tradução são os **compiladores**.

Os programas escritos em linguagem de programação são chamados de **Código Fonte**. Os programas escritos em linguagem de máquina são chamados de **Código Executável** ou simplesmente **Executável**.

2.3 Exemplo de programa Java

Uma linguagem de programação muito utilizada no mercado de trabalho é a linguagem Java. Os códigos dos programas escritos em Java são colocados em arquivos com a extensão **.java**.

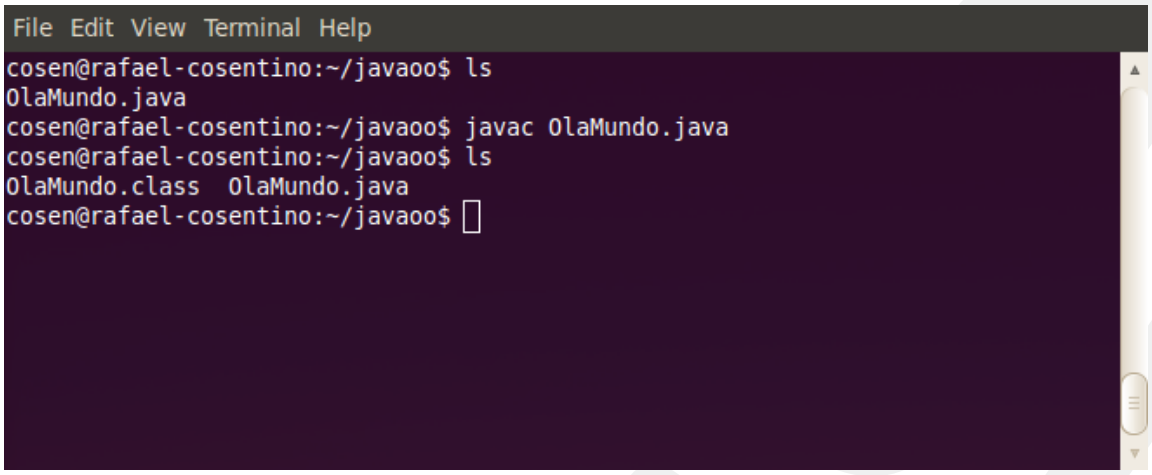
Observe o código do exemplo de programa escrito em Java que imprime uma mensagem na tela.

```
1 // arquivo: OlaMundo.java
2 class OlaMundo  {
3     public static void main(String[] args) {
4         System.out.println("Olá Mundo");
5     }
6 }
```

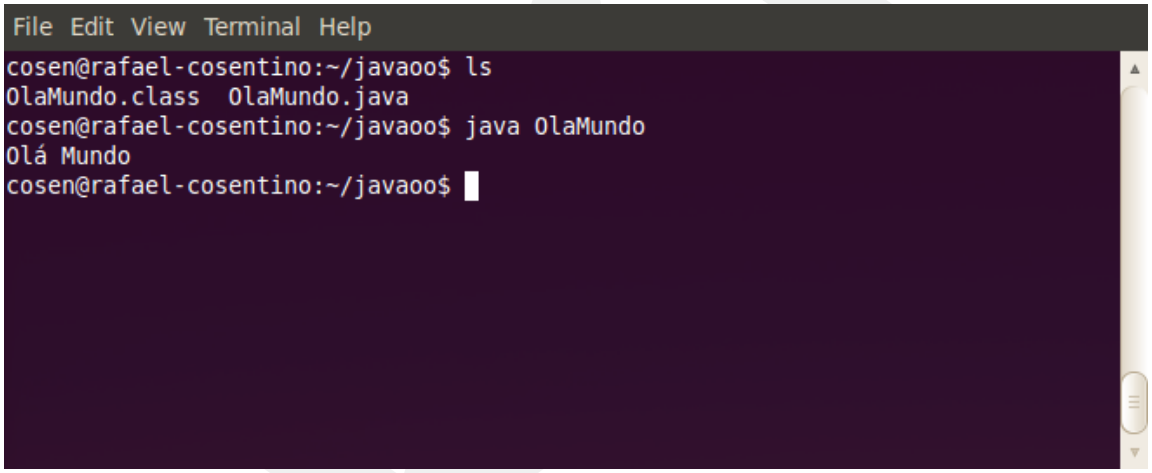
Agora, não é necessário entender todo esse código fonte. O importante é saber que todo programa escrito em Java para executar precisa ter o método especial **main**.

Esse código fonte precisa ser traduzido para um executável para que um computador possa executá-lo. Essa "tradução" é realizada por um compilador da linguagem Java.

Suponha que o código acima seja colocado no arquivo **OlaMundo.java**. O compilador padrão do Java (**javac**) pode ser utilizado para compilar esse arquivo. O **terminal** permite que o compilador seja executado.



O programa gerado pelo compilador é colocado em um arquivo chamado *OlaMundo.class* que podemos executar pelo terminal.



2.4 Método Main - Ponto de Entrada

Para um programa Java executar, é necessário definir um método especial para ser o ponto de entrada do programa, ou seja, para ser o primeiro método a ser chamado quando o programa for executado. O método main precisa ser **public**, **static**, **void** e receber um array de strings como argumento.

Algumas das possíveis versões do método main:

```
1 static public void main(String[] args)
2 public static void main(String[] args)
3 public static void main(String args[])
4 public static void main(String[] parametros)
```

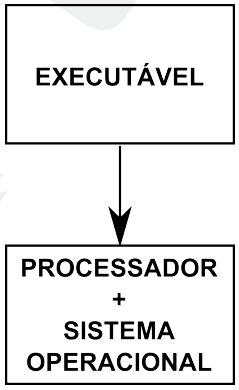
Os parâmetros do método main são passados pela linha de comando e podem ser manipulados dentro do programa. O código abaixo imprime cada parâmetro recebido em uma linha diferente. A execução do programa é mostrada na figura abaixo.

```
1  // arquivo: OlaMundo.java
2  class OlaMundo {
3      public static void main(String[] args) {
4          for(int i = 0; i < args.length; i++) {
5              System.out.println(args[i]);
6          }
7      }
8  }
```

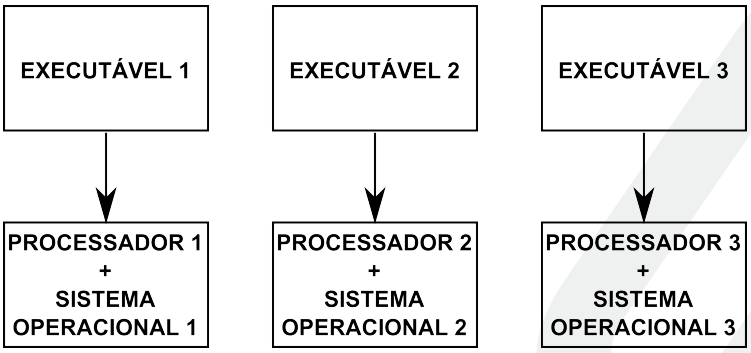
```
File  Edit  View  Terminal  Help
cosen@rafael-cosentino:~/javaoo$ ls
OlaMundo.class  OlaMundo.java
cosen@rafael-cosentino:~/javaoo$ java OlaMundo Rafael Alexandre Jonas
Rafael
Alexandre
Jonas
cosen@rafael-cosentino:~/javaoo$
```

2.5 Máquinas Virtuais

Os compiladores geram executáveis específicos para um determinado tipo de processador e um sistema operacional. Em outras palavras, esses executáveis funcionam corretamente em computadores que possuem um determinado tipo de processador e um determinado sistema operacional.

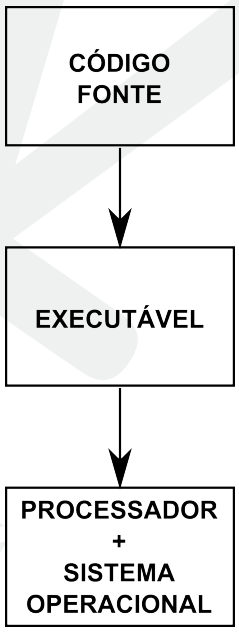


Como os computadores possuem processadores de tipos diferentes e sistemas operacionais diferentes, é necessário ter diversos executáveis do mesmo programa para poder executá-lo em computadores diferentes.

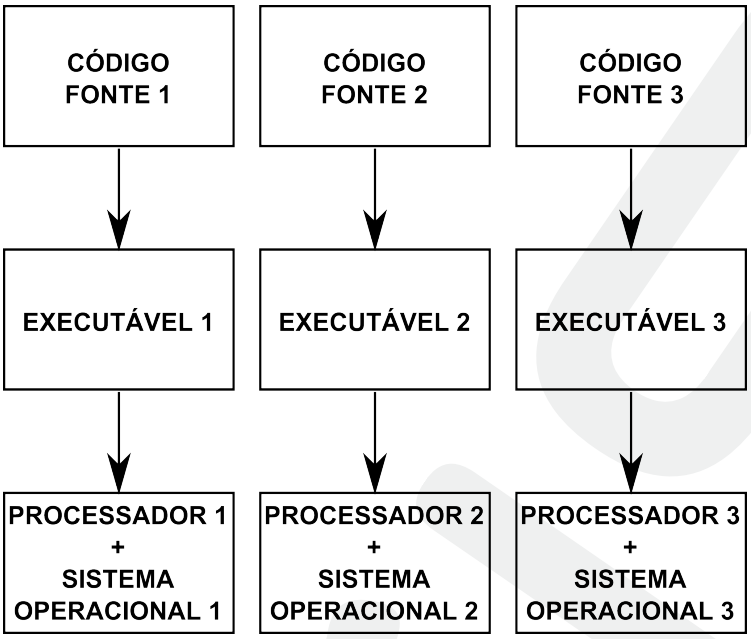


Do ponto de vista das empresas, manter atualizados e distribuir corretamente os diversos executáveis de um mesmo programa é custoso.

Além disso, ao escrever um programa, os programadores utilizam bibliotecas que já realizam parte do trabalho que deveria ser feito. Essas bibliotecas são chamadas diretamente do código fonte e elas normalmente só funcionam em um determinado Sistema Operacional e em um tipo específico de processador. Dessa forma, o código fonte de um programa fica acoplado a um determinado tipo de processador e um determinado sistema operacional.



Não basta manter diversos executáveis de um mesmo programa. É necessário ter diversos códigos fonte de um mesmo programa, um para cada tipo de processador e sistema operacional que serão utilizados. O alto custo do desenvolvimento e manutenção desses códigos fonte pode tornar inviável comercialização de um software.

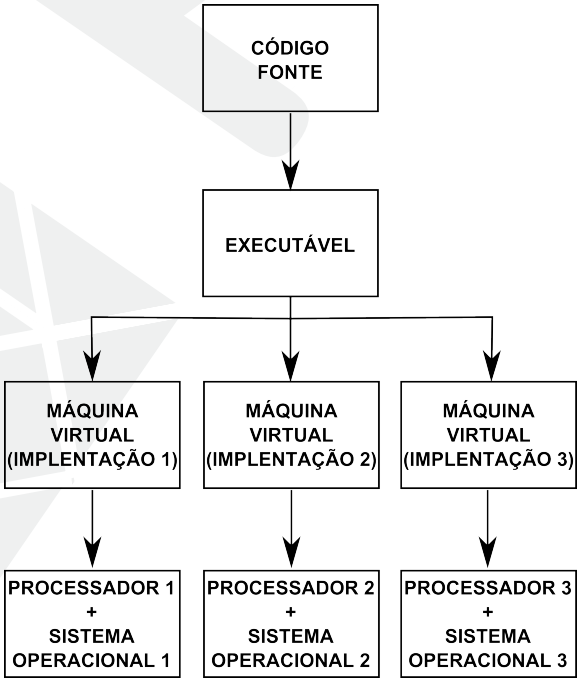


Para não ser necessário manter diversos executáveis e diversos códigos fonte de um mesmo programa, surgiu o conceito de máquina virtual.

Uma máquina virtual é um programa que funciona como um computador, ela sabe ler e executar instruções de programas escritos na sua própria linguagem de máquina. Além disso, as máquinas virtuais oferecem bibliotecas para criação de programas.

Para ser vantajoso, é necessário ter diversas implementações da mesma máquina virtual. Cada implementação deve funcionar em um sistema operacional e em um tipo de processador.

Dessa forma, um programador pode escrever o código fonte de um programa chamando diretamente as bibliotecas da máquina virtual. Depois, compilar esse código fonte para gerar um executável escrito na linguagem de máquina da máquina virtual.



Um benefício da utilização de máquinas virtuais é o ganho de portabilidade. Em outras palavras, para executar um programa em computadores diferentes, é necessário apenas um código fonte e um executável.

Uma desvantagem de utilizar uma máquina virtual para executar um programa e a diminuição de performance já que a própria máquina virtual consome recursos do computador. Além disso, as instruções do programa são processadas primeiro pela máquina virtual e depois pelo computador.

Por outro lado, as máquinas virtuais podem aplicar otimizações que aumentam a performance da execução de um programa. Inclusive, essas otimizações podem considerar informações geradas durante a execução. São exemplos de informações geradas durante a execução: a quantidade de uso da memória RAM e do processador do computador, a quantidade de acessos ao disco rígido, a quantidade de chamadas de rede e a frequência de execução de um determinado trecho do programa.

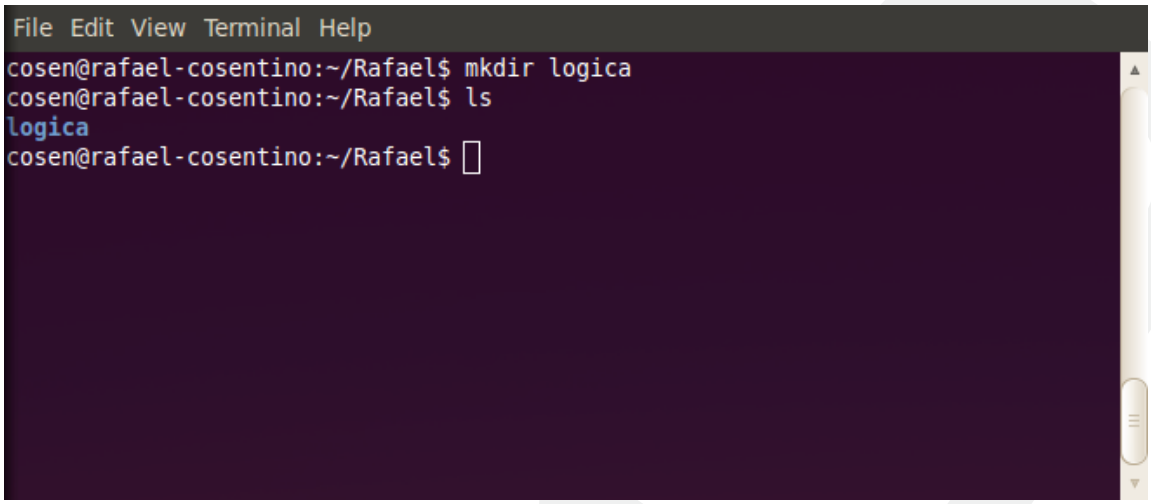
Algumas máquinas virtuais identificam os trechos do programa que estão sendo mais chamados em um determinado momento da execução para traduzi-los para a linguagem de máquina do computador. A partir daí, esses trechos podem ser executados diretamente no computador sem passar pela máquina virtual. Essa análise da máquina virtual é realizada durante toda a execução.

Com essas otimizações que consideram várias informações geradas durante a execução, um programa executado com máquina virtual pode até ser mais eficiente em alguns casos do que um programa executado diretamente no sistema operacional.

2.6 Exercícios

1. Abra um terminal e crie uma pasta com o seu nome. Você deve salvar os exercícios nessa pasta.

2. Dentro da sua pasta de exercícios crie uma pasta para os arquivos desenvolvidos nesse capítulo chamada **logica**.



3. Crie um programa que imprima uma mensagem na tela.

```
1 // arquivo: OlaMundo.java
2 class OlaMundo {
3     public static void main(String[] args) {
4         System.out.println("Olá Mundo");
5     }
6 }
```

Compile e execute a classe OLAMUNDO.

2.7 Variáveis

Um dos recursos fundamentais que as linguagens de programação oferecem são as variáveis. A função de uma variável é permitir que um programa armazene e manipule dados na memória RAM do computador.

Declaração

Na linguagem de programação Java, as variáveis devem ser declaradas para que possam ser utilizadas. A declaração de uma variável envolve definir um nome único (identificador) e um tipo de valor. As variáveis são acessadas pelos nomes e armazenam valores compatíveis com o seu tipo.

```
1 // tipo: int, nome: numero
2 int numero;
3
4 // tipo: double, nome: preco
5 double preco;
```

Declaração, Inicialização e Uso

Toda variável deve ser inicializada antes de ser utilizada pela primeira vez. Se isso não for realizado ocorrerá um erro de compilação. A inicialização é realizada através do operador de atribuição =. Esse operador guarda um valor na variável, ou melhor, no espaço de memória reservado para a variável.

```
1  // Declaracoes
2  int numero;
3  double preco;
4
5  // Inicializacao
6  numero = 10;
7
8  // Uso
9  System.out.println(numero);
10
11 // Erro de compilacao
12 System.out.println(preco);
```

Mais Declarações

A declaração de uma variável pode ser feita em qualquer linha de um bloco. Não é necessário declarar todas as variáveis no começo do bloco.

```
1  // Declaracao com Inicializacao
2  int numero = 10;
3
4  // Uso
5  System.out.println(numero);
6
7  // Outra Declaracao com Inicializacao
8  double preco = 137.6;
9
10 // Uso
11 System.out.println(preco);
```

Escopo

Toda variável pode ser utilizada dentro do bloco no qual ela foi declarada após a linha da sua inicialização.

```
1  // Aqui nao pode usar a variavel numero
2  int numero = 10;
3
4  // Aqui pode usar a variavel numero
5  System.out.println(numero);
```

Não é permitido declarar duas variáveis com o mesmo nome dentro de um mesmo bloco. Se isso ocorrer o compilador indicará um erro de compilação.

```
int numero;

// Erro de compilacao
int numero;
```

Dentro de um bloco é possível abrir um sub-bloco. As variáveis declaradas antes do sub-bloco podem ser acessadas também no sub-bloco. Assuma que o código abaixo está dentro do bloco do método main.

```
int numero = 10;

// sub-bloco do bloco do metodo main
{
    // Aqui pode usar a variavel numero
    System.out.println(numero);
}

// Aqui tambem pode usar o numero
System.out.println(numero);
```

Porém, variáveis declaradas em sub-blocos não podem ser acessadas do "bloco pai" nem de outro sub-bloco do "bloco pai".

```
// sub-bloco do bloco do metodo main
{
    int numero = 10;
    // Aqui pode usar a variavel numero
    System.out.println(numero);
}

// Erro de compilacao
System.out.println(numero);

// sub-bloco do bloco do metodo main
{
    // Erro de compilacao
    System.out.println(numero);
}
```

Dentro de sub-blocos diferentes de um mesmo bloco é permitido declarar variáveis com nomes iguais.

```
// sub-bloco do bloco do metodo main
{
    int numero = 10;
}

// sub-bloco do bloco do metodo main
{
    int numero = 10;
}
```

2.8 Operadores

Para manipular os valores das variáveis, podemos aplicar os operadores da linguagem Java. Há diversos tipos de operadores.

Aritméticos

Para realizar as operações matemáticas podemos aplicar os operadores: +(soma), -(subtração), *(multiplicação), /(divisão) e % (módulo).

```
1  int dois = 1 + 1;
2  int tres = 4 - 1;
3  int quatro = 2 * 2;
4  int cinco = 10 / 2;
5  int um = 5 % 2;
```

Igualdade e Desigualdade

Para verificar se o valor armazenado em uma variável é igual ou diferente a um determinado valor ou ao valor de outra variável, devem ser utilizados os operadores: ==(igual) e !=(diferente). Esses operadores devolvem valores do tipo **boolean**(true ou false).

```
1  int numero = 10;
2
3  boolean teste1 = numero == 10;
4  boolean teste2 = numero != 10;
5
6  // imprime: true
7  System.out.println(teste1);
8
9  // imprime: false
10 System.out.println(teste2);
```

Relacionais

As variáveis numéricas podem ser comparadas com o intuito de descobrir a ordem numérica dos seus valores. Os operadores relacionais são: >, <, > e <. Esses operadores devolvem valores do tipo BOOLEAN (true ou false).

```
int numero = 10;

boolean teste1 = numero > 5;
boolean teste2 = numero < 5;
boolean teste2 = numero >= 10;
boolean teste2 = numero <= 5;

// imprime: true
System.out.println(teste1);

// imprime: false
System.out.println(teste2);

// imprime: true
System.out.println(teste3);

// imprime: false
System.out.println(teste4);
```

Lógicos

Para verificar diversas condições é possível aplicar os operadores lógicos: && e ||.

```
int numero1 = 10;
int numero2 = 20;

boolean teste1 = numero1 > 1 && numero2 < 10;
boolean teste1 = numero1 > 1 || numero2 < 10;

// imprime: False
System.out.println(teste1);

// imprime: True
System.out.println(teste2);
```

2.9 IF-ELSE

Muitas vezes os programadores necessitam definir quais trechos de código devem ser executados de acordo com uma ou mais condições. Por exemplo, quando o pagamento de um boleto é feito em alguma agência bancária, o sistema do banco deve verificar a data de vencimento do boleto para aplicar a multa por atraso ou não.

```
GregorianCalendar vencimento = new GregorianCalendar(2010, 10, 20);
GregorianCalendar agora = new GregorianCalendar();

if (agora.after(vencimento)) {
    System.out.println("Calculando multa...");
} else {
    System.out.println("Sem multa");
}
```

O comando IF permite que valores booleanos sejam testados. Se o valor passado como parâmetro para o comando IF for TRUE o bloco do IF é executado caso contrário o bloco do ELSE é executado.

O parâmetro passado para o comando IF deve ser um valor booleano, caso contrário o código fonte não compila. O comando ELSE e o seu bloco são opcionais.

2.10 WHILE

Em alguns casos, é necessário repetir um trecho de código diversas vezes. Suponha que seja necessário imprimir 10 vezes na tela a mensagem: "Bom Dia". Isso poderia ser realizado colocando no código fonte 10 linhas iguais. Porém, se ao invés de 10 vezes fosse necessário imprimir 100 vezes já seriam 100 linhas iguais no código fonte.

Através do comando WHILE, é possível definir quantas vezes um determinado trecho de código deve ser executado pelo computador.

```
1 int contador = 0;
2
3 while(contador <= 100) {
4     System.out.println("Bom Dia");
5     contador++;
6 }
```

A variável CONTADOR indica quantas vezes a mensagem "Bom Dia" foi impressa na tela. O operador ++ incrementa a variável CONTADOR a cada rodada.

O parâmetro do comando WHILE tem que ser um valor booleano. Caso contrário ocorrerá um erro de compilação.

2.11 FOR

O comando FOR é análogo ao WHILE. A diferença entre esses dois comandos é que o FOR recebe três argumentos.

```
1 for(int contador = 0; contador <= 100; contador++) {
2     System.out.println("Bom Dia");
3 }
```

2.12 Exercícios

4. Imprima na tela o seu nome 100 vezes.

```
1 // arquivo: ImprimeNome.java
2 class ImprimeNome {
3     public static void main(String[] args) {
4         for(int contador = 0; contador < 100; contador++) {
5             System.out.println("Rafael Cosentino");
6         }
7     }
8 }
```

Compile e execute a classe IMPRIMENOME

5. Imprima na tela os números de 1 até 100.

```
// arquivo: ImprimeAte100.java
class ImprimeAte100 {
    public static void main(String[] args) {
        for(int contador = 1; contador <= 100; contador++) {
            System.out.println(contador);
        }
    }
}
```

Compile e execute a classe IMPRIMEATE100

6. Faça um programa que percorra todos os número de 1 até 100. Para os números ímpares deve ser impresso um "*" e para os números pares deve ser impresso dois "**".

```
*
**
*
**
*
**
```

```
// arquivo: ImprimePadrao1.java
class ImprimePadrao1 {
    public static void main(String[] args) {
        for(int contador = 1; contador <= 100; contador++) {
            int resto = contador % 2;
            if(resto == 1) {
                System.out.println("*");
            } else {
                System.out.println("**");
            }
        }
    }
}
```

Compile e execute a classe IMPRIMEPADRAO1

7. Faça um programa que percorra todos os número de 1 até 100. Para os números múltiplos de 4 imprima a palavra "PI" e para os outros imprima o próprio número.

```
1
2
3
PI
5
6
7
PI
```

```java
// arquivo: ImprimePadrao2.java
class ImprimePadrao2 {
    public static void main(String[] args) {
        for(int contador = 1; contador <= 100; contador++) {
            int resto = contador % 4;
            if(resto == 0) {
                System.out.println("PI");
            } else {
                System.out.println(contador);
            }
        }
    }
}
```

Compile e execute a classe IMPRIMEPADRAO2

8. Crie um programa que imprima na tela um triângulo de "*". Observe o padrão abaixo.

```
*
**
***
****
*****
```

```java
// arquivo: ImprimePadrao3.java
class ImprimePadrao3 {
    public static void main(String[] args) {
        String linha = "*";
        for(int contador = 1; contador <= 10; contador++) {
            System.out.println(linha);
            linha += "*";
        }
    }
}
```

Compile e execute a classe IMPRIMEPADRAO3

9. (Opcional) Crie um programa que imprima na tela vários triângulos de "*". Observe o padrão abaixo.

```
*
**
***
****
*
**
***
****
```

```
// arquivo: ImprimePadrao4.java
class ImprimePadrao4 {
    public static void main(String[] args) {
        String linha = "*";
        for(int contador = 1; contador <= 10; contador++) {
            System.out.println(linha);
            int resto = contador % 4;
            if(resto == 0) {
                linha = "*";
            } else {
                linha += "*";
            }
        }
    }
}
```

Compile e execute a classe IMPRIMEPADRAO4

10. (Opcional) A série fibonacci é uma sequência de números. O primeiro elemento da série é 0 e o segundo é 1. Os outros elementos são calculados somando os dois antecessores.

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...
```

Crie um programa para imprimir os 30 primeiros números da série de fibonacci.

```
// arquivo: ImprimePadrao5.java
class ImprimePadrao5 {
    public static void main(String[] args) {
        int penultimo = 0;
        int ultimo = 1;

        System.out.println(penultimo);
        System.out.println(ultimo);

        for(int contador = 0; contador < 28; contador++) {
            int proximo = anterior + ultimo;
            System.out.println(proximo);

            penultimo = ultimo;
            ultimo = proximo;
        }
    }
}
```

Compile e execute a classe IMPRIMEPADRAO5

Capítulo 3

Orientação a Objetos

Para apresentar os conceitos desse e dos próximos capítulos, simularemos a criação de um sistema bancário. No desenvolvimento de um sistema, um dos primeiros passos é identificar os elementos que devem existir no mesmo. O que deve existir num sistema bancário?

- Contas
- Funcionários
- Clientes
- Agências

O exercício de identificar os elementos que devem existir em um sistema é uma tarefa complicada e depende muito da experiência dos desenvolvedores e do conhecimento que eles tem sobre o domínio (o contexto do banco).

3.1 Objetos

Depois de identificar os elementos que devem existir num sistema bancário, devemos decidir como eles devem ser representados dentro do computador. Como o nosso objetivo é criar um sistema orientado a objetos, devemos utilizar objetos para representar os elementos do sistema bancário.

Para exemplificar, suponha que o banco tenha um cliente chamado João. Dentro do sistema, deve existir um objeto para representar o João. Conceitualmente, um objeto deve ser modelado afim de ficar o mais parecido possível com aquilo que ele está representando.

A função de um objeto é representar um elemento do domínio.

Algumas informações do João como nome, idade e sexo são importantes para o banco. O objeto que representa o João deve possuir os dados que são relevantes para o banco. Esses dados são armazenados nos **atributos** do objeto que representa o João.

Quando alterações nos valores dos atributos de um objeto devem ser realizadas, o próprio objeto deve realizá-las. As lógicas para essas alterações são definidas nos **métodos** do objeto.

Os métodos também são utilizados para possibilitar a interação entre os objetos que formam um determinado sistema. Por exemplo, quando um cliente requisita um saque através de um

caixa eletrônico do banco, o objeto que representa o caixa eletrônico deve interagir com o objeto que representa a conta do cliente para tentar realizar a operação.

Um objeto é composto por atributos e métodos.

Não é adequado usar o objeto que representa o João para representar outro cliente do banco, pois cada cliente possui os seus próprios dados. Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.

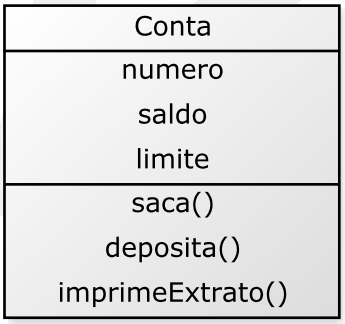
Os objetos não representam apenas coisas concretas como os clientes do banco. Eles também devem ser utilizados para representar coisas abstratas como uma conta de um cliente ou um serviço que o banco ofereça.

3.2 Classes

Em algum momento, os objetos devem ser criados. Por exemplo, quando alguém se torna cliente do banco, um novo objeto deve ser construído para representar esse cliente. A criação de um objeto novo envolve alocar um espaço na memória do computador para colocar os atributos e métodos do mesmo.

Um objeto é como se fosse uma casa ou um prédio, para ser construído precisa de um espaço físico, no caso dos objetos o espaço é algum trecho vago da memória RAM de algum computador, no caso das casas e dos prédios o espaço é algum terreno vazio em alguma cidade.

Um prédio é construído a partir de uma planta criada por um engenheiro. Para criar um objeto, é necessário algo semelhante a uma planta para que sejam definidos os atributos e métodos que objeto deve ter. Em orientação a objetos, a "planta"de um objeto é chamada de **classe**.



Uma classe funciona como uma forma para criar objetos. Inclusive, vários objetos podem ser criados a partir de uma única classe. Assim como vários prédios poderiam ser construídos a partir de uma única planta.

Os objetos criados a partir da classe CONTA terão todos os atributos e métodos mostrados no diagrama UML. As diferenças entre esses objetos são os valores dos atributos.

Classes Java

O conceito de classe apresentado anteriormente é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe CONTA poderia ser escrita utilizando a linguagem Java. Inicialmente, discutiremos apenas sobre os atributos. Os métodos serão abordados posteriormente.

```
1 class Conta {
2     double saldo;
3     double limite;
4     int numero;
5 }
```

A classe Java CONTA é declarada utilizando a palavra reservada **class**. No corpo dessa classe, são declaradas três variáveis que são os atributos que os objetos possuirão. Como a linguagem Java é estaticamente tipada os tipos dos atributos são definidos no código. O atributo **saldo** e **limite** são do tipo **double** que permite armazenar números com casas decimais e o atributo **numero** é do tipo **int** que permite armazenar números inteiros.

Criando objetos em Java

Após definir a classe CONTA, podemos criar objetos a partir dela. Esses objetos devem ser alocados na memória RAM do computador. Felizmente, todo o processo de alocação do objeto na memória é gerenciado pela máquina virtual. O gerenciamento da memória é um dos recursos mais importantes oferecidos pela máquina virtual.

Do ponto de vista da aplicação, basta utilizar um comando especial para criar objetos que a máquina virtual se encarrega do resto. O comando para criar objetos é o **new**.

```
1 class TestaConta {
2     public static void main(String[] args) {
3         new Conta();
4     }
5 }
```

A linha com o comando NEW poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe CONTA. A classe Java TESTACONTA serve apenas para colocarmos o método MAIN que é o ponto de partida da aplicação.

Chamar o comando NEW passando uma classe Java é como se estivéssemos contratando uma construtora e passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta.

3.3 Referências

Após criar um objeto, provavelmente, desejaremos utilizá-lo. Para isso precisamos acessá-lo de alguma maneira. Os objetos são acessados através de **referências**. Uma referência é "link" que aponta para um objeto.

Referências em Java

Ao utilizar o comando NEW, um objeto é alocado em algum lugar da memória RAM. Para que possamos acessar esse objeto, precisamos da referência dele. O comando NEW devolve a referência do objeto que foi criado.

Para guardar as referências devolvidas pelo comando NEW, podemos declarar variáveis cujo o propósito é justamente armazenar referências.

```
1 Conta ref = new Conta();
```

No código Java acima, a variável **ref** receberá a referência do objeto criado pelo comando NEW. Essa variável é do tipo **Conta**. Isso significa que ela só pode armazenar referência para objeto do tipo **Conta**.

3.4 Manipulando Atributos

Podemos alterar os valores dos atributos de um objeto se tivermos a referência dele. Os atributos são acessados pelo nome. No caso específico da linguagem Java, a sintaxe para acessar um atributo utiliza o operador "**.**".

```
1 Conta ref = new Conta();
2
3 ref.saldo = 1000.0;
4 ref.limite = 500.0;
5 ref.numero = 1;
6
7 System.out.println(ref.saldo);
8 System.out.println(ref.limite);
9 System.out.println(ref.numero);
```

No código acima, o atributo NOME recebe o valor 1000.0. O atributo LIMITE recebe o valor 500 e o NUMERO recebe o valor 1. Depois, os valores são impressos na tela através do comando SYSTEM.OUT.PRINTLN.

Valores Default

Poderíamos instanciar um objeto e utilizar o seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores default. Os atributos de tipos numéricos são inicializados com 0, os atributos do tipo BOOLEAN(true ou false) são inicializados com FALSE e os demais atributos com NULL (referência vazia).

A inicialização dos atributos com os valores default ocorre na instanciação, ou seja, quando o comando NEW é utilizado. Dessa forma, todo objeto "nasce"com os valores default. Em alguns casos, é necessário trocar esses valores. Para trocar o valor default de um atributo, devemos inicializá-lo na declaração.

```
1 class Conta {
2     double saldo;
3     double limite = 500;
4     int numero;
5 }
```

3.5 Agregação

Todo cliente do banco pode adquirir um cartão de crédito. Suponha que um cliente adquira um cartão de crédito. Dentro do sistema do banco, deve existir um objeto que represente o

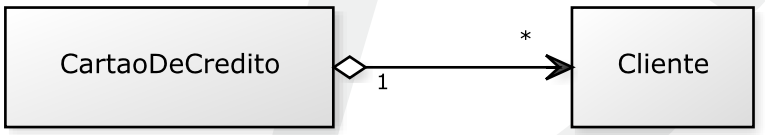
cliente e outro que represente o cartão de crédito. Para expressar a relação entre o cliente e o cartão de crédito, algum vínculo entre esses dois objetos deve ser estabelecido.

Duas classes deveriam ser criadas, uma para definir os atributos e métodos dos clientes e outra para os cartões de crédito. Para expressar o relacionamento entre cliente e cartão de crédito, podemos colocar um atributo do tipo CLIENTE na classe CARTAODECREDITO.

```
1 class Cliente {
2     String nome;
3 }
```

```
1 class CartaoDeCredito {
2     Cliente cliente;
3 }
```

Esse tipo de relacionamento é chamado de **Agregação**. Há uma notação gráfica na linguagem UML para representar uma agregação. Veja o diagrama abaixo.



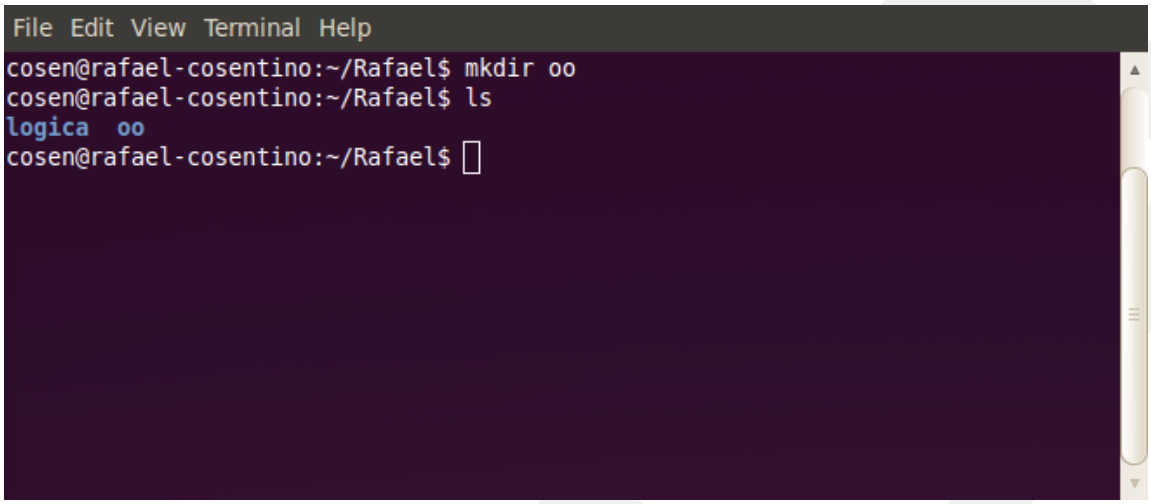
No relacionamento entre cartão de crédito e cliente, um cartão de crédito só pode se relacionar com um único cliente. Por isso, no diagrama acima, o 1 é colocado ao lado da classe CLIENTE. Por outro lado, um cliente pode se relacionar com muitos cartões de crédito. Por isso, no diagrama acima, o "*"é colocado ao lado da classe CARTAODECREDITO.

O relacionamento entre um objeto da classe CLIENTE e um objeto da classe CARTAODECREDITO só é concretizado quando a referência do objeto da classe CLIENTE é armazenada no atributo cliente do objeto da classe CARTAODECREDITO. Depois de relacionados, podemos acessar os atributos do cliente através da referência do objeto da classe CARTAODECREDITO.

```
1 // Criando um objeto de cada classe
2 CartaoDeCredito cdc = new CartaoDeCredito();
3 Cliente c = new Cliente();
4
5 // Ligando os objetos
6 cdc.cliente = c;
7
8 // Acessando o nome do cliente
9 cdc.cliente.nome = "Rafael Cosentino";
```

3.6 Exercícios

1. Dentro da sua pasta de exercícios crie uma pasta para os arquivos desenvolvidos nesse capítulo chamada **oo**.



2. No sistema de um banco algumas informações dos clientes como nome e código precisam ser armazenadas. Crie uma classe para modelar os objetos que representarão os clientes.

```
1  // arquivo: Cliente.java
2  class Cliente {
3      String nome;
4      int codigo;
5  }
```

3. Faça um teste criando dois objetos da classe CLIENTE.

```
1  // arquivo: TestaCliente.java
2  class TestaCliente {
3      public static void main(String[] args) {
4          Cliente c1 = new Cliente();
5          c1.nome = "Rafael Cosentino";
6          c1.codigo = 1;
7
8          Cliente c2 = new Cliente();
9          c2.nome = "Jonas Hirata";
10         c2.codigo = 2;
11
12         System.out.println(c1.nome);
13         System.out.println(c1.codigo);
14
15         System.out.println(c2.nome);
16         System.out.println(c2.codigo);
17     }
18 }
```

Compile e execute a classe TESTACLIENTE.

4. Os bancos oferecem aos clientes a possibilidade de obter um cartão de crédito que pode ser utilizados para fazer compras. Um cartão de crédito possui um número e uma data de validade. Crie uma classe para modelar os objetos que representarão os cartões de crédito.

```
1  // arquivo: CartaoDeCredito.java
2  class CartaoDeCredito {
3      int numero;
4      String dataDeValidade;
5  }
```

5. Faça um teste criando dois objetos da classe CARTAODECREDITO. Altere e imprima os atributos desses objetos.

```
1  // arquivo: TestaCartaoDeCredito.java
2  class TestaCartaoDeCredito {
3      public static void main(String[] args) {
4          CartaoDeCredito cdc1 = new CartaoDeCredito();
5          cdc1.numero = 111111;
6          cdc1.dataDeValidade = "01/01/2012";
7
8          CartaoDeCredito cdc2 = new CartaoDeCredito();
9          cdc2.numero = 222222;
10         cdc2.dataDeValidade = "01/01/2014";
11
12         System.out.println(cdc1.numero);
13         System.out.println(cdc1.dataDeValidade);
14
15         System.out.println(cdc2.numero);
16         System.out.println(cdc2.dataDeValidade);
17     }
18 }
```

Compile e execute a classe TESTACARTAODECREDITO.

6. Defina um vínculo entre a classe CLIENTE e a classe CARTAODECREDITO. Para isso você deve alterar a classe CARTAODECREDITO.

```
1  // arquivo: CartaoDeCredito.java
2  class CartaoDeCredito {
3    int numero;
4    String dataDeValidade;
5
6    // ADICIONE A LINHA ABAIXO
7    Cliente cliente;
8  }
```

7. Teste o relacionamento entre clientes e cartões de crédito.

```
// arquivo: TestaClienteECartao.java
class TestaClienteECartao {
    public static void main(String[] args) {
        // Criando alguns objetos
        Cliente c = new Cliente();
        CartaoDeCredito cdc = new CartaoDeCredito();

        // Carregando alguns dados
        c.nome = "Rafael Cosentino";
        cdc.numero = 111111;

        // Ligando os objetos
        cdc.cliente = c;

        System.out.println(cdc.numero);
        System.out.println(cdc.cliente.nome);
    }
}
```

Compile e execute a classe TESTACLIENTEECARTAO.

8. As agências bancárias possuem um número. Crie uma classe para modelar as agências.

```
// arquivo: Agencia.java
class Agencia {
    int numero;
}
```

9. As contas do banco possuem saldo e estão vinculadas a uma agência. Crie uma classe para modelar as contas e estabeleça um vínculo com a classe AGENCIA.

```
// arquivo: Conta.java
class Conta {
    double saldo;
    Agencia agencia;
}
```

10. Teste o relacionamento entre contas e agências.

```
// arquivo: TestaContaEAgencia.java
class TestaContaEAgencia {
    public static void main(String[] args) {
        // Criando alguns objetos
        Agencia a = new Agencia();
        Conta c = new Conta();

        // Carregando alguns dados
        a.numero = 178;
        c.saldo = 1000.0;

        // Ligando os objetos
        c.agencia = a;

        System.out.println(c.agencia.numero);
        System.out.println(c.saldo);
    }
}
```

Compile e execute a classe TESTACLIENTEECARTAO.

11. Faça um teste que imprima os atributos de um objeto da classe CONTA logo após a instanciação.

```
1 // arquivo: TestaValoresDefault.java
2 class TestaValoresDefault {
3     public static void main(String[] args) {
4         Conta c = new Conta();
5 
6         System.out.println(c.saldo);
7         System.out.println(c.agencia);
8     }
9 }
```

Compile e execute a classe TESTAVALORESDEFAULT.

12. Altere a classe CONTA para que todos os objetos criados a partir dessa classe possuam 100 de saldo inicial.

```
1 // arquivo: Conta.java
2 class Conta {
3     double saldo = 100.0;
4     Agencia agencia;
5 }
```

Compile e execute a classe TESTAVALORESDEFAULT.

3.7 Métodos

Quando um depósito é realizado, o valor depositado deve ser acrescentado ao saldo da conta correspondente. No sistema do banco, as contas são representadas por objetos e os saldos são atributos desses objetos. Alterações nos valores dos atributos de um objeto devem ser realizadas pelo próprio objeto através dos seus métodos.

Supondo que tivéssemos um objeto da classe CONTA, a sintaxe para chamar um método seria relativamente parecida à sintaxe de acessar atributos. A diferença é que devemos passar uma lista de parâmetros na chamada de um método. No caso do método que implementa a operação de depositar é interessante passar o valor que está sendo depositado como parâmetro.

```
1 Conta c = new Conta();
2 
3 // Chamando um metodo
4 c.deposita(1000);
```

Contudo, os objetos da classe CONTA não possuem o método DEPOSITA() até que este seja definido na classe CONTA.

```
class Conta {
    double saldo;
    double limite;

    void deposita(double valor) {
        this.saldo += valor;
    }
}
```

Com o DEPOSITA() definido na classe CONTA todo objeto criado a partir dessa classe possuirá esse método. Na declaração de um método, a lista de parâmetros deve ser definida, ou seja, os tipos e a ordem dos valores que o método deve receber precisa ser especificada. Os atributos do objeto que receber uma chamada ao método DEPOSITA() são acessados de dentro do corpo do método através da palavra reserva **this**. Como o método DEPOSITA() não deve devolver nenhuma resposta ele é marcado com **void**.

Da maneira que o método DEPOSITA() foi implementado, os saldos das contas podem assumir valores incorretos. Isso ocorreria se um valor negativo fosse depositado.

```
c.deposita(-1000);
```

Para evitar essa situação, algum tipo de validação pode ser implementada no método DEPOSITA().

```
void deposita(double valor) {
    if(valor > 0) {
        this.saldo += valor;
    }
}
```

De acordo com essa implementação o método DEPOSITA() simplesmente não faz nada se o valor passado como parâmetro é negativo. Uma desvantagem dessa abordagem é que o chamador do método não saberá se houve algo de errado no processamento do DEPOSITA(). Para resolver esse problema, o método DEPOSITA() pode devolver uma resposta.

```
boolean deposita(double valor) {
    if(valor > 0) {
        this.saldo += valor;
        return true;
    } else {
        return false;
    }
}
```

Ao chamar o método DEPOSITA() a resposta pode ser armazenada em uma variável.

```
Conta c = new Conta();
boolean reposta = c.deposita(1000);

if(resposta){
    System.out.println("O depósito foi efetuado");
} else {
    System.out.println("O depósito não foi efetuado");
}
```

3.8 Sobrecarga(Overloading)

É natural que um cliente do banco queira consultar periodicamente algumas informações das suas contas. Para obter essas informações o cliente pode pedir um extrato. Como dados das contas devem ser utilizados para gerar os extratos, é interessante definir um método na classe CONTA para implementar a lógica para imprimir extratos.

```
class Conta {
    double saldo;
    double limite;

    void imprimeExtrato(){
        System.out.println("Saldo: " + this.saldo);
        System.out.println("Limite: " + this.limite);
    }
}
```

Normalmente, quando requisitamos extratos para os bancos, temos a possibilidade de passar uma quantidade de dias que devem ser considerados ou utilizar a quantidade padrão definida pelo banco (ex: 15 dias). Para implementar essas possibilidades, podemos definir dois métodos na classe CONTA.

```
class Conta {

    void imprimeExtrato(){
        // Logica para gerar um extrato dos ultimos 15 dias
    }

    void imprimeExtrato(int dias){
        // Logica para gerar um extrato com
        // uma quantidade variavel de dias
    }
}
```

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão pelo banco para gerar os extratos. O segundo recebe um valor inteiro como parâmetro e deve considerar essa quantidade de dias para gerar os extratos. Os dois métodos possuem o mesmo nome e lista de parâmetros diferentes. Quando dois ou mais métodos são definidos na mesma classe com o mesmo nome, dizemos que houve uma **sobrecarga** de métodos. Uma sobrecarga de métodos só é válida se as listas de parâmetros dos métodos são diferentes entre si.

No caso dos dois métodos que geram extratos, poderíamos evitar repetição de código fazendo um método chamar o outro. Normalmente, o método mais específico chama o método mais genérico.

```
class Conta {

    void imprimeExtrato(){
        this.imprimeExtrato(15);
    }

    void imprimeExtrato(int dias){
        // Logica para gerar um extrato com
        // uma quantidade variavel de dias
    }
}
```

3.9 Exercícios

13. Acrescente alguns métodos na classe CONTA para definir as lógicas de depositar, sacar e imprimir extrato.

```
// arquivo: Conta.java
class Conta {
    double saldo;
    Agencia agencia;

    void deposita(double valor) {
        this.saldo += valor;
    }

    void saca(double valor) {
        this.saldo -= valor;
    }

    void imprimeExtrato() {
        System.out.println("SALDO: " + this.saldo);
    }
}
```

14. Teste os métodos da classe CONTA.

```
// arquivo: TestaMetodosConta.java
class TestaMetodosConta {
    public static void main(String[] args) {
        Conta c = new Conta();

        c.deposita(1000);
        c.imprimeExtrato();

        c.saca(100);
        c.imprimeExtrato();
    }
}
```

Compile e execute a classe TESTAMETODOSCONTA.

15. Crie uma classe para modelar as faturas dos cartões de crédito.

```
1  // arquivo: Fatura.java
2  class Fatura {
3      double total;
4
5      void adiciona(double valor) {
6          this.total += valor;
7      }
8
9      double calculaMulta() {
10         return this.total * 0.2;
11     }
12
13     void imprimeDados() {
14         System.out.println("Total: " + this.total);
15     }
16 }
```

16. Teste a classe FATURA.

```
1  // arquivo: TestaMetodosFatura.java
2  class TestaMetodosFatura {
3      public static void main(String[] args) {
4          Fatura f = new Fatura();
5
6          f.adiciona(100);
7          f.adiciona(200);
8
9          f.imprimeDados();
10
11         double multa = f.calculaMulta();
12
13         System.out.println(multa);
14     }
15 }
```

Compile e execute a classe TESTAMETODOSFATURA.

3.10 Construtores

Na criação de um cartão de crédito é fundamental que o número dele seja definido. Na criação de uma agência é necessário que o número dela seja escolhido. Uma conta quando criada deve ser associada a uma agência.

Após criar um objeto para representar um cartão de crédito, poderíamos definir um valor para o atributo NUMERO. De maneira semelhante, podemos definir um número para um objeto da classe AGENCIA e uma agência para um objeto da classe CONTA.

```
1  CartaoDeCredito cdc = new CartaoDeCredito();
2  cdc.numero = 12345;
```

```
1  Agencia a = new Agencia();
2  a.numero = 11111;
```

```
Conta c = new Conta();
c.agencia = a;
```

Definir os valores dos atributos obrigatórios de um objeto logo após a criação dele resolveria as restrições do sistema do banco. Porém, nada garante que todos os desenvolvedores sempre lembrem de inicializar esses valores.

Para resolver esse problema, podemos utilizar o conceito de CONSTRUTOR que permite que um determinado trecho de código seja executado toda vez que um objeto é criado. Um construtor pode receber parâmetros como os métodos mas não pode devolver uma resposta. Além disso, em Java, para definir um construtor devemos utilizar o mesmo nome da classe na qual o construtor foi colocado.

```
// arquivo: CartaoDeCredito.java
class CartaoDeCredito {
    int numero;

    CartaoDeCredito(int numero) {
        this.numero = numero;
    }
}
```

```
// arquivo: Agencia.java
class Agencia {
    int numero;

    Agencia(int numero) {
        this.numero = numero;
    }
}
```

```
// arquivo: Conta.java
class Conta {
    Agencia agencia;

    Conta(Agencia agencia) {
        this.agencia = agencia;
    }
}
```

Na criação de um objeto com o comando NEW, os argumentos passados devem ser compatíveis com a lista de parâmetro de algum construtor definido na classe que está sendo instanciada. Caso contrário, um erro de compilação ocorrerá para avisar o desenvolvedor dos valores obrigatórios que devem ser passados para criar um objeto.

```
// Passando corretamente os parâmetros para os construtores
CartaoDeCredito cdc = new CartaoDeCredito(1111);

Agencia a = new Agencia(1234);

Conta c = new Conta(a1);
```

```
1 // ERRO DE COMPILACAO
2 CartaoDeCredito cdc = new CartaoDeCredito();
3
4 // ERRO DE COMPILACAO
5 Agencia a = new Agencia();
6
7 // ERRO DE COMPILACAO
8 Conta c = new Conta(a1);
```

3.10.1 Construtor Default

Toda vez que um objeto é criado, um construtor da classe correspondente deve ser chamado. Mesmo quando nenhum construtor foi definido explicitamente, há um construtor padrão que é inserido pelo próprio compilador. O construtor padrão só é inserido se nenhum construtor foi explicitamente definido e ele não possui parâmetros.

Portanto, para instanciar uma classe que não possui construtores definidos no código fonte, devemos utilizar o construtor padrão já que este é inserido automaticamente pelo compilador.

```
1 // arquivo: Conta.java
2 class Conta {
3
4 }
```

```
1 // Chamando o construtor padrão
2 Conta c = new Conta();
```

Lembrando que o construtor padrão só é inserido pelo compilador se nenhum construtor foi definido no código fonte. Dessa forma, se você fizer um construtor com parâmetros então não poderá utilizar o comando NEW sem passar argumentos pois um erro de compilação ocorrerá.

```
1 // arquivo: Agencia.java
2 class Agencia {
3     int numero;
4
5     Agencia(int numero) {
6         this.numero = numero;
7     }
8 }
```

```
1 // ERRO DE COMPILACAO
2 Agencia a = new Agencia();
```

3.10.2 Sobrecarga de Construtores

O conceito de sobrecarga de métodos pode ser aplicado para construtores. Dessa forma, podemos definir diversos construtores na mesma classe.

```
// arquivo: Pessoa.java
class Pessoa {
    String rg;
    int cpf;

    Pessoa(String rg){
        this.rg = rg;
    }

    Pessoa(int cpf){
        this.cpf = cpf;
    }
}
```

Quando dois ou mais construtores são definidos, há duas ou mais opções no momento de utilizar o comando NEW.

```
// Chamando o primeiro construtor
Pessoa p1 = new Pessoa("123456X");
// Chamando o segundo construtor
Pessoa p2 = new Pessoa(123456789);
```

3.10.3 Construtores chamando Construtores

Assim como podemos encadear métodos também podemos encadear construtores.

```
// arquivo: Conta.java
class Conta {
    int numero;
    double limite;

    Conta(int numero) {
        this.numero = numero;
    }

    Conta(int numero, double limite) {
        this(numero);
        this.limite = limite;
    }
}
```

3.11 Exercícios

17. Acrescente um construtor na classe AGENCIA para receber um número como parâmetro.

```
// arquivo: Agencia.java
class Agencia {
    int numero;

    // ADICIONE O CONSTRUTOR ABAIXO
    Agencia(int numero) {
        this.numero = numero;
    }
}
```

18. Tente compilar novamente o arquivo TESTACONTAEAGENCIA. Observe o erro de compilação.

19. Altere o código da classe TESTACONTAEAGENCIA para que o erro de compilação seja resolvido.

    Substitua a linha abaixo:

```
1 Agencia a = new Agencia();
```

    Por:

```
1 Agencia a = new Agencia(1234);
```

    Compile novamente o arquivo TESTACONTAEAGENCIA.

20. Acrescente um construtor na classe CARTAODECREDITO para receber um número como parâmetro.

```
1  // arquivo: CartaoDeCredito.java
2  class CartaoDeCredito {
3      int numero;
4      String dataDeValidade;
5
6      Cliente cliente;
7
8      // ADICIONE O CONSTRUTOR ABAIXO
9      CartaoDeCredito(int numero) {
10         this.numero = numero;
11     }
12 }
```

21. Tente compilar novamente os arquivos TESTACARTAODECREDITO e TESTACLIENTEECARTAO. Observe os erros de compilação.

22. Altere o código da classe TESTACARTAODECREDITO e TESTACLIENTEECARTAO para que os erros de compilação sejam resolvidos.

    Substitua trechos de código semelhantes ao trecho abaixo:

```
1 CartaoDeCredito cdc1 = new CartaoDeCredito();
2 cdc1.numero = 111111;
```

    Por trechos de código semelhantes ao trecho abaixo:

```
1 CartaoDeCredito cdc = new CartaoDeCredito(111111);
```

Compile novamente os arquivos TESTACARTAODECREDITO e TESTACLIENTEECARTAO.

23. Acrescente um construtor na classe CONTA para receber uma referência como parâmetro.

```
1  // arquivo: Conta.java
2  class Conta {
3      double saldo = 100.0;
4      Agencia agencia;
5
6      // ADICIONE O CONSTRUTOR ABAIXO
7      Conta(Agencia agencia) {
8          this.agencia = agencia;
9      }
10
11     void deposita(double valor) {
12         this.saldo += valor;
13     }
14     void saca(double valor) {
15         this.saldo -= valor;
16     }
17     void imprimeExtrato() {
18         System.out.println("SALDO: " + this.saldo);
19     }
20 }
```

24. Tente compilar novamente os arquivos TESTACONTAEAGENCIA, TESTAMETODOSCONTA e TESTAVALORESDEFAULT. Observe os erros de compilação.

25. Altere o código da classe TESTACONTAEAGENCIA, TESTAMETODOSCONTA e TESTAVALORESDEFAULT para que o erros de compilação sejam resolvidos.

    Substitua trechos de código semelhantes ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta();
```

    Por trechos de código semelhantes ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta(a);
```

    Também substitua trechos de código semelhantes ao trecho abaixo:

```
1  Conta c = new Conta();
```

    Por trechos de código semelhantes ao trecho abaixo:

```
1  Agencia a = new Agencia(1234);
2  Conta c = new Conta(a);
```

Compile novamente os arquivos TESTACONTAEAGENCIA, TESTAMETODOSCONTA e TESTAVALORESDEFAULT.

3.12 Referências como parâmetro

Assim como podemos passar valores primitivos como argumentos para um método ou construtor também podemos passar valores não primitivos (referências).

Suponha um método na classe CONTA que implemente a lógica de transferência de valores entre contas. Esse método deve receber como argumento além do valor a ser transferido a conta que receberá o dinheiro.

```
1 void transfere(Conta destino, double valor) {
2     this.saldo -= valor;
3     destino.saldo += valor;
4 }
```

Na chamada do método TRANSFERE, devemos ter duas referências de contas: uma para chamar o método e outra para passar como parâmetro.

```
1 Conta origem = new Conta();
2 origem.saldo = 1000;
3
4 Conta destino = new Conta();
5
6 origem.transfere(destino, 500);
```

Quando a variável DESTINO é passada como parâmetro, somente a referência armazenada nessa variável é enviada para o método TRANSFERE e não o objeto correspondente a referência. Em outras palavras, somente o "link" para a conta que receberá o valor da transferência é enviado para o método TRANSFERE.

3.13 Exercícios

26. Acrescente um método na classe CONTA para implementar a lógica de transferência de valores entre contas.

```
1 void transfere(Conta destino, double valor) {
2     this.saldo -= valor;
3     destino.saldo += valor;
4 }
```

27. Faça um teste para verificar o funcionamento do método transfere.

```
1  // arquivo: TestaMetodoTransfere.java
2  class TestaMetodoTransfere {
3      public static void main(String[] args) {
4          Agencia a = new Agencia(1234);
5
6          Conta origem = new Conta(a);
7          origem.saldo = 1000;
8
9          Conta destino = new Conta(a);
10         destino.saldo = 1000;
11
12         origem.transfere(destino, 500);
13
14         System.out.println(origem.saldo);
15         System.out.println(destino.saldo);
16     }
17 }
```

Compile e execute a classe TESTAMETODOTRANSFERE.

Capítulo 4

Arrays

Suponha que o sistema de uma loja virtual precisa armazenar os preços dos produtos que ela comercializa. Não seria nada prático declarar para cada produto uma variável que armazena o preço do mesmo. Para isso, é possível utilizar as estruturas chamadas **arrays**.

Um array é uma estrutura que armazena um ou mais valores de um determinado tipo. Um array pode ser criado com o comando NEW.

```
1 double[] precos = new double[100];
```

A variável PRECOS armazena a referência de um array criado na memória RAM do computador. Na memória, o espaço ocupado por esse array está dividido em 100 "pedaços" iguais numerados de 0 até 99. Cada "pedaço" pode armazenar um valor do tipo DOUBLE. Quando um array é criado, as posições dele são inicializadas com os valores default.

```
1 double[] precos = new double[100];
2 precos[0] = 130.5;
3 precos[99] = 17.9;
```

Acessar posições fora do intervalo de índices de um array gera um erro de execução.

No momento da criação de um array, os seus valores podem ser definidos.

```
1 double[] precos = new double[2]{100.4,87.5};
```

```
1 double[] precos = new double[]{100.4,87.5};
```

```
1 double[] precos = {100.4,87.5};
```

4.1 Arrays de Arrays

Podemos criar arrays de arrays.

```
1 double[][] arrays = new double[3][];
2
3 arrays[0] = new double[2];
4 arrays[1] = new double[3];
5 arrays[2] = new double[4];
6
7 arrays[0][0] = 10.5;
```

```
1 double[][] arrays = {
2     new double[]{1.5, 3.7},
3     new double[]{6.6, 9.7, 6.7},
4     new double[]{8.5, 2.8, 7.5, 8.6}
5 };
6 System.out.println(arrays[0][0]);
```

4.2 Percorrendo Arrays

Os arrays podem ser facilmente preenchidos utilizando o comando WHILE ou o FOR.

```
1 double[] precos = new double[100];
2 for(int i = 0; i < 100; i++) {
3     precos[i] = i;
4 }
```

```
1 double[][] tabela = new double[100][50];
2 for(int i = 0; i < 100; i++) {
3     for(int j = 0; j < 50; j++) {
4         tabela[i][j] = i * j;
5     }
6 }
```

A quantidade de elementos em cada dimensão dos arrays pode ser obtida através do atributo LENGTH.

```
1 double[][] tabela = new double[100][50];
2 for(int i = 0; i < tabela.length; i++) {
3     for(int j = 0; j < tabela[i].length; j++) {
4         tabela[i][j] = i * j;
5     }
6 }
```

Para acessar todos os elementos de um array, é possível aplicar o comando FOR com uma sintaxe um pouco diferente.

```
1 double[] precos = new double[100];
2 for(int i = 0; i < precos.length; i++) {
3     precos[i] = i;
4 }
5
6 for(double x : precos)
7 {
8     System.out.println(x);
9 }
```

4.3 Operações

Ordenando um Array

Suponha um array de strings criado para armazenar alguns nomes de pessoas. É comum querer ordenar esses nomes utilizando a ordem alfabética. Essa tarefa pode ser realizada através do método ARRAYS.SORT.

```
1 String[] nomes = new String[]{"rafael cosentino", "jonas hirata", "marcelo martins" };
2 Arrays.sort(nomes);
3
4 for(String nome : nomes)
5 {
6     System.out.println(nome);
7 }
```

Copiando um Array

Para copiar o conteúdo de um array para outro com maior capacidade, podemos utilizar o método ARRAYS.COPYOF.

```
1 String[] nomes = new String[] {"rafael", "jonas", "marcelo" };
2 String[] nomes2 = Arrays.copyOf(nomes, 10);
```

4.4 Exercícios

1. Dentro da sua pasta de exercícios crie uma pasta para os arquivos desenvolvidos nesse capítulo chamada **arrays**.

2. Crie um programa que imprima na tela os argumentos passados na linha de comando para o método MAIN.

```
1  // arquivo: Imprime.java
2  class Imprime {
3      public static void main(String[] args) {
4          for(String arg : args) {
5              System.out.println(arg);
6          }
7      }
8  }
```

Compile e execute a classe IMPRIME. Na execução não esqueça de passar alguns parâmetros na linha de comando.

```
java Imprime Rafael Alex Daniel Jonas
```

3. Faça um programa que ordene o array de strings recebido na linha de comando.

```
1  // arquivo: Ordena.java
2  class Ordena {
3      public static void main(String[] args) {
4          java.util.Arrays.sort(args);
5
6          for(String arg : args) {
7              System.out.println(arg);
8          }
9      }
10 }
```

Compile e execute a classe ORDENA. Na execução não esqueça de passar alguns parâmetros na linha de comando.

```
java Ordena Rafael Alex Daniel Jonas
```

4. (Opcional) Faça um programa que calcule a média dos elementos recebidos na linha de comando. Dica: para converter strings para double pode ser utilizado um código semelhante a este:

```
String s = "10";
double d = Double.parseDouble(s);
```

5. (Opcional) Crie um programa que encontre o maior número entre os valores passados na linha de comando.

6. (Opcional) Crie um array de arrays na forma de triângulo com os valores do padrão abaixo e imprima na tela esses valores.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Capítulo 5

Eclipse

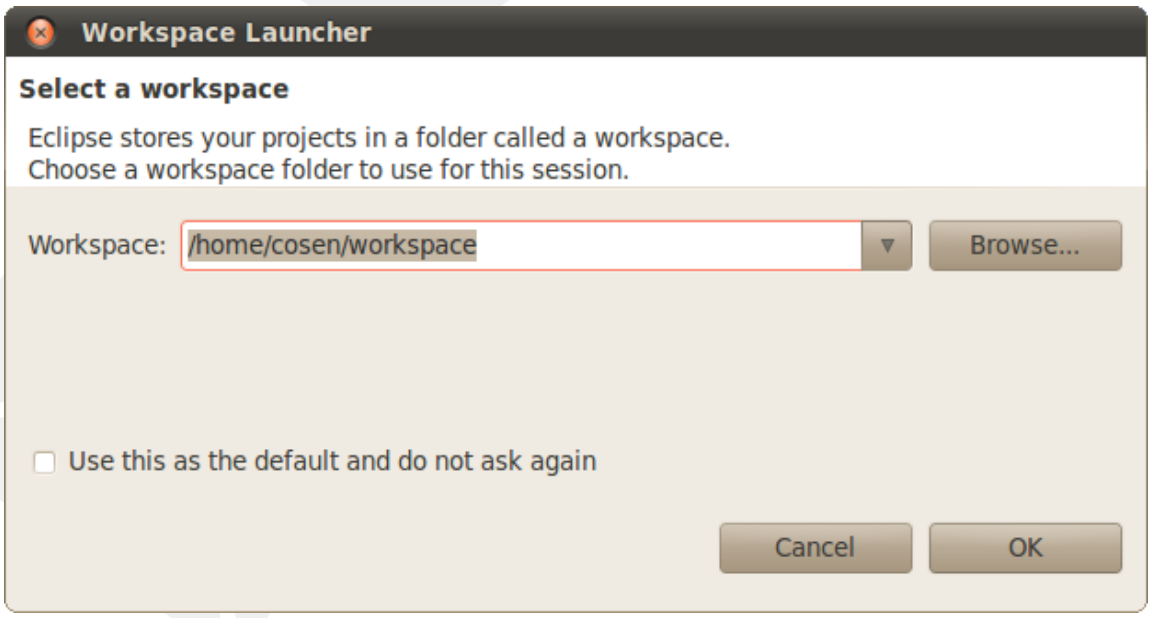
Na prática, alguma ferramenta de desenvolvimento é adotada para aumentar a produtividade. Essas ferramentas são chamadas **IDE**(Integrated Development Environment - Ambiente de Desenvolvimento Integrado). Uma IDE é uma ferramenta que provê facilidades para o desenvolvedor realizar as principais tarefas relacionadas ao desenvolvimento de um software.

No caso específico da plataforma Java, a IDE mais utilizada e o **Eclipse**. Essa ferramenta é bem abrangente e oferece recursos sofisticados para o desenvolvimento de uma aplicação Java. Além disso, ela é gratuita.

As diversas distribuições do Eclipse podem ser obtidas através do site: `http://www.eclipse.org/`.

5.1 Workspace

Uma workspace é uma pasta que normalmente contém projetos e configurações do Eclipse. Logo que é executado, o Eclipse permite que o usuário selecione uma pasta como workspace.

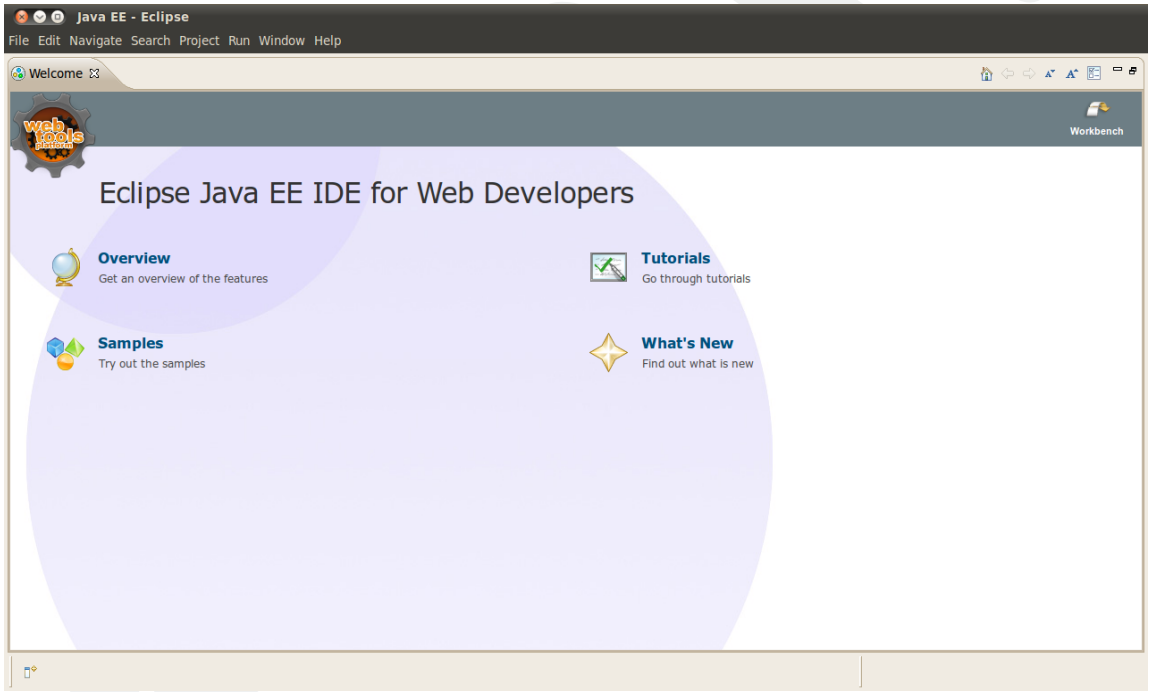


Podemos ter várias worspaces para organizar conjuntos projetos e configurações indepen-

dentemente.

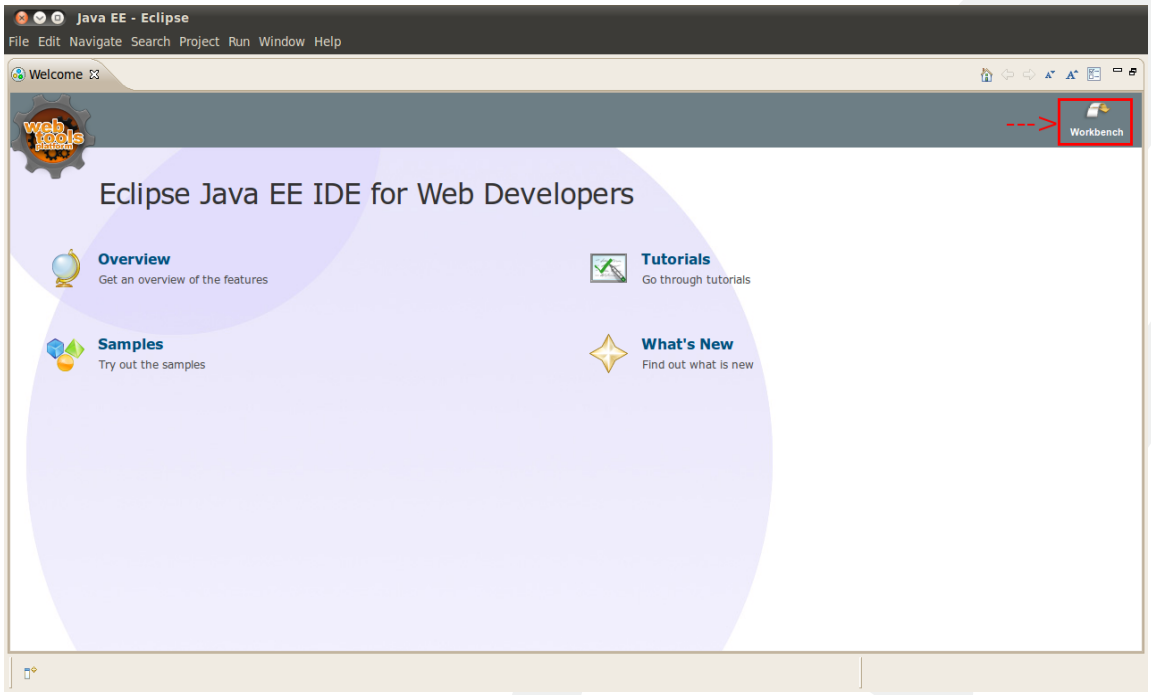
5.2 Welcome

A primeira tela do Eclipse (welcome) mostra “links” para alguns exemplos, tutorias, visão geral da ferramenta e novidades.



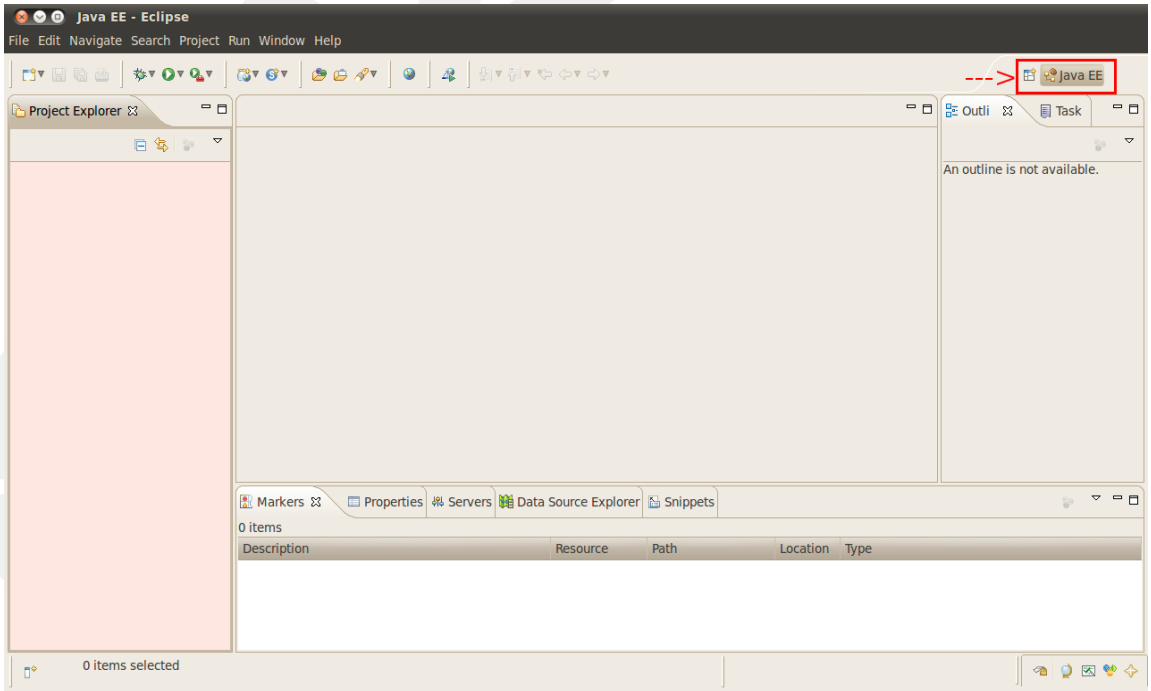
5.3 Workbench

Da tela welcome podemos ir para workbench que é a área de trabalho do Eclipse clicando no ícone destacado na figura abaixo ou simplesite fechando a tela welcome.



5.4 Perspective

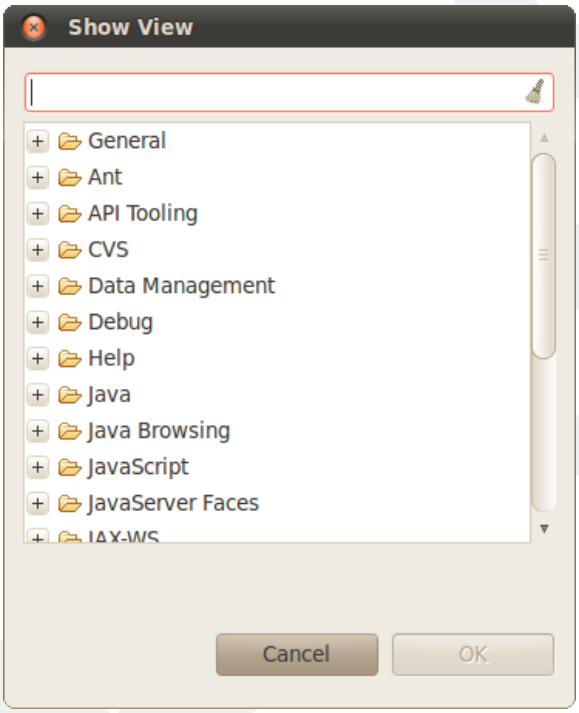
O Eclipse oferece vários modos de trabalho ao desenvolvedor. Cada modo de trabalho é adequado para algum tipo de tarefa. Esses modos de trabalhos são chamados de perspectives (perspectivas). Podemos abrir uma perspectiva através do ícone no canto superior direito da workbench.



Na maioria das vezes utilizaremos a perspectiva Java.

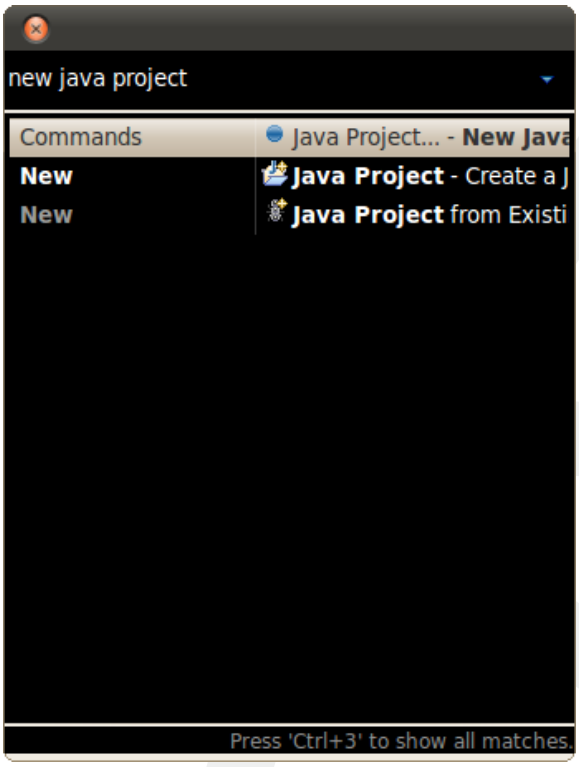
5.5 Views

As telas que são mostradas na workbench são chamadas de views. O desenvolvedor pode abrir, fechar ou mover qualquer view ao seu gosto e necessidade. Uma nova view pode ser aberta através do menu **Window->Show View->Other**.

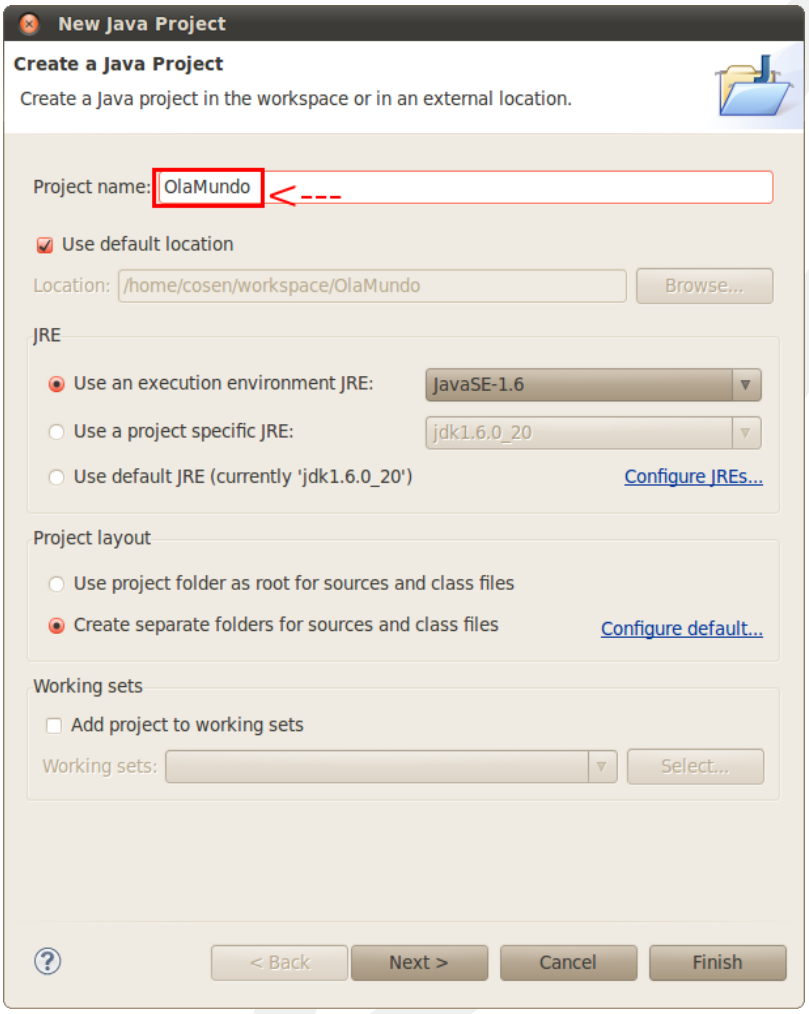


5.6 Criando um projeto java

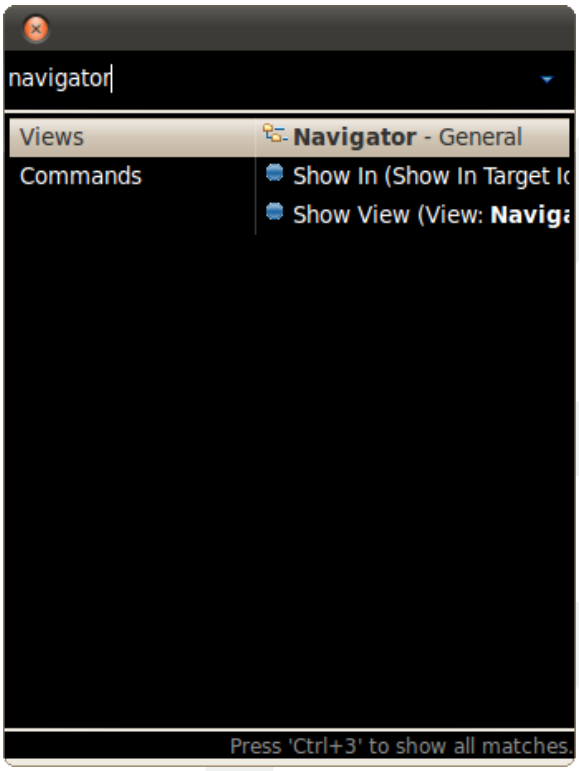
Podemos utilizar os menus para criar projetos porém a maneira mais prática é utilizar o **Quick Acess** através do atalho **CTRL+3**. O Quick Acess permite que o desenvolvedor busque as funcionalidades do Eclipse pelo nome.

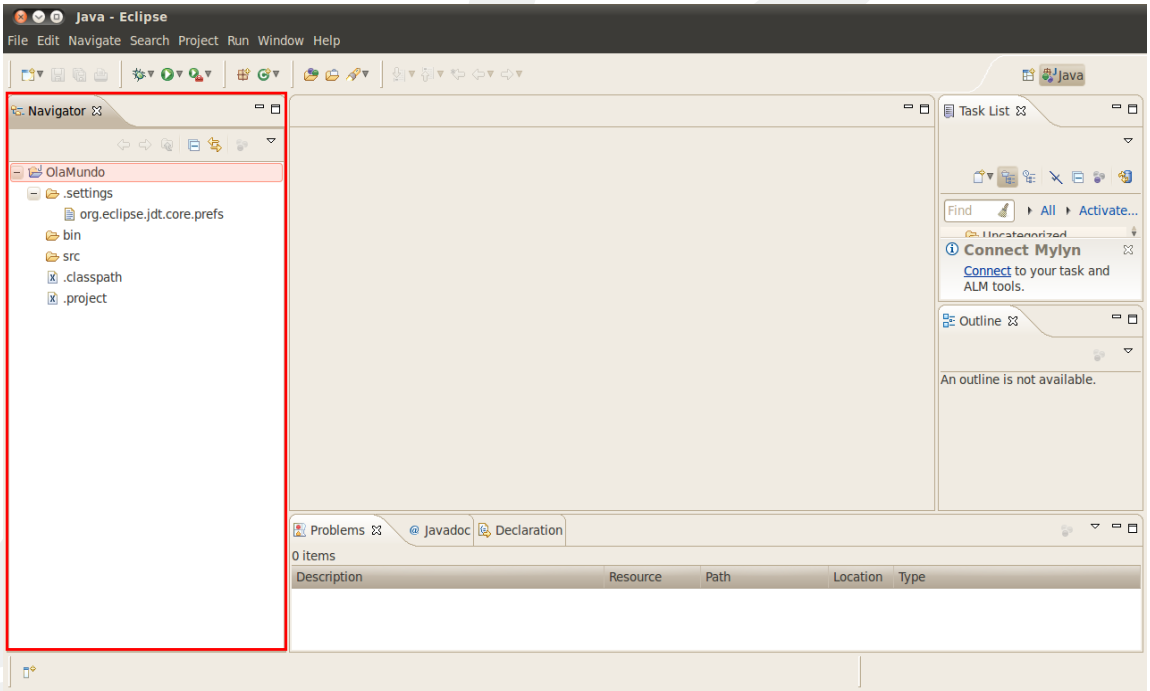


Na tela de criação de projeto java devemos escolher um nome para o projeto.



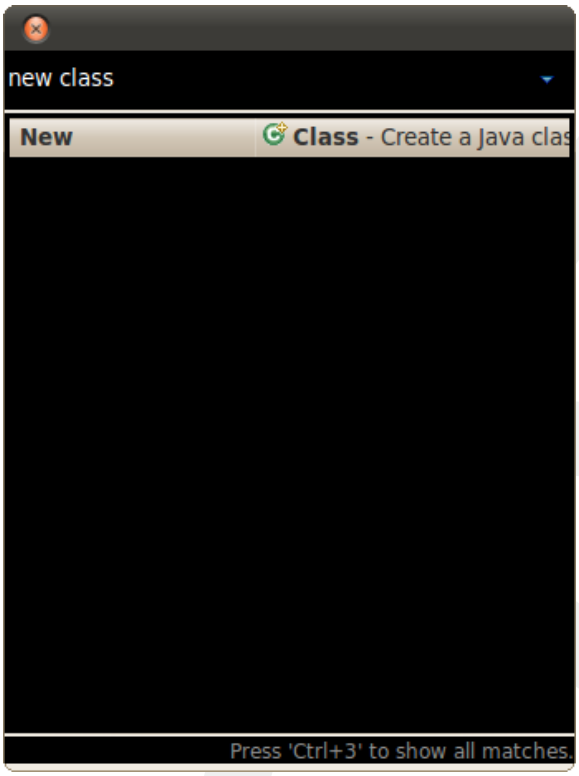
A estrutura do projeto pode ser vista através da view **Navigator** que pode ser aberta com Quick Acess.



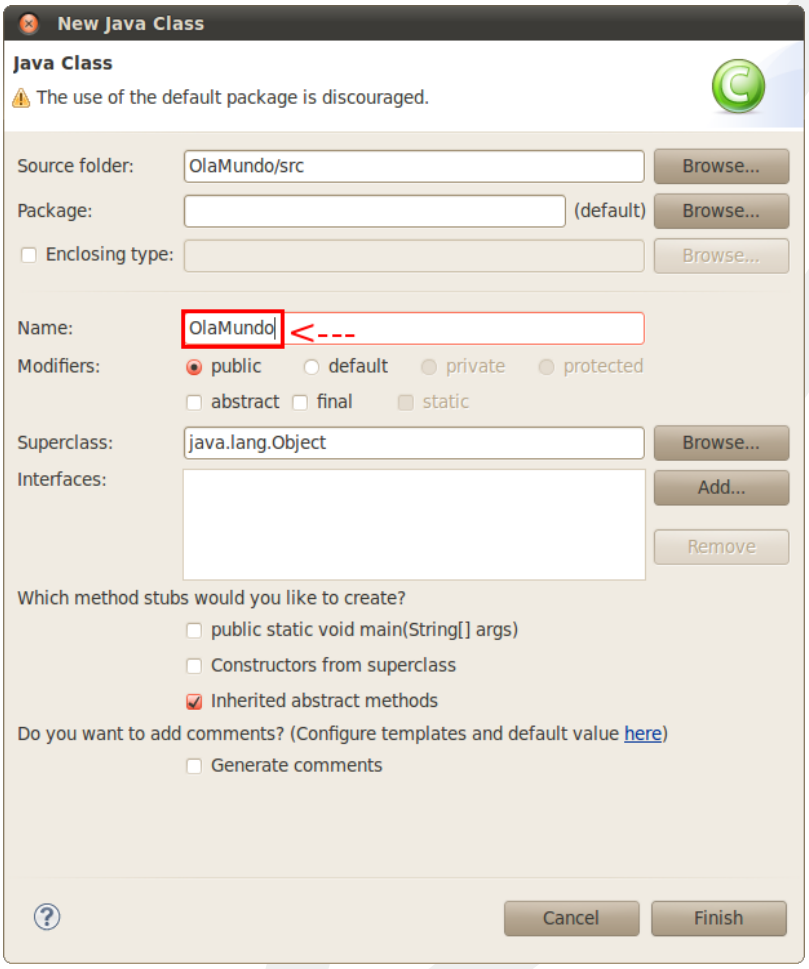


5.7 Criando uma classe

Após a criação de um projeto podemos criar uma classe também através do Quick Acess.



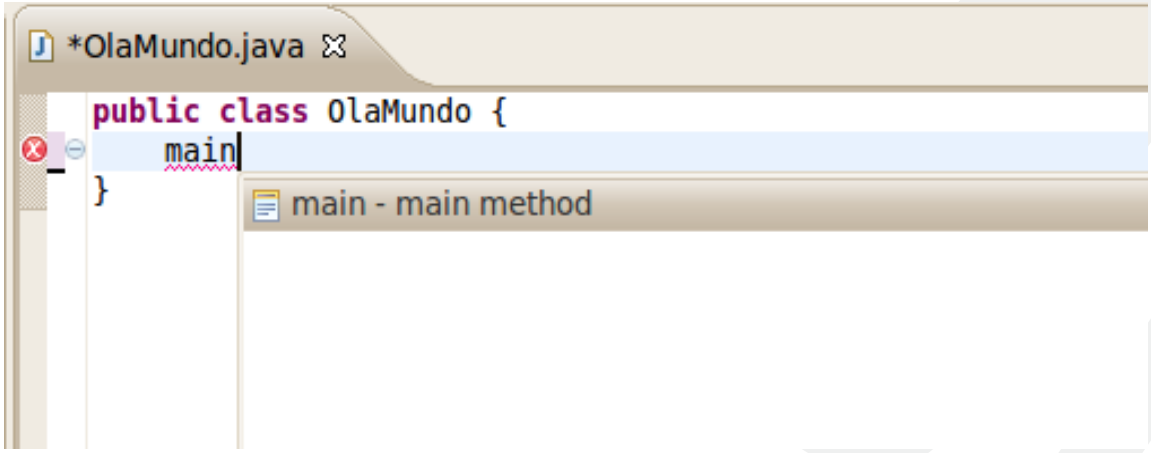
Na tela de criação de classe devemos escolher um nome.



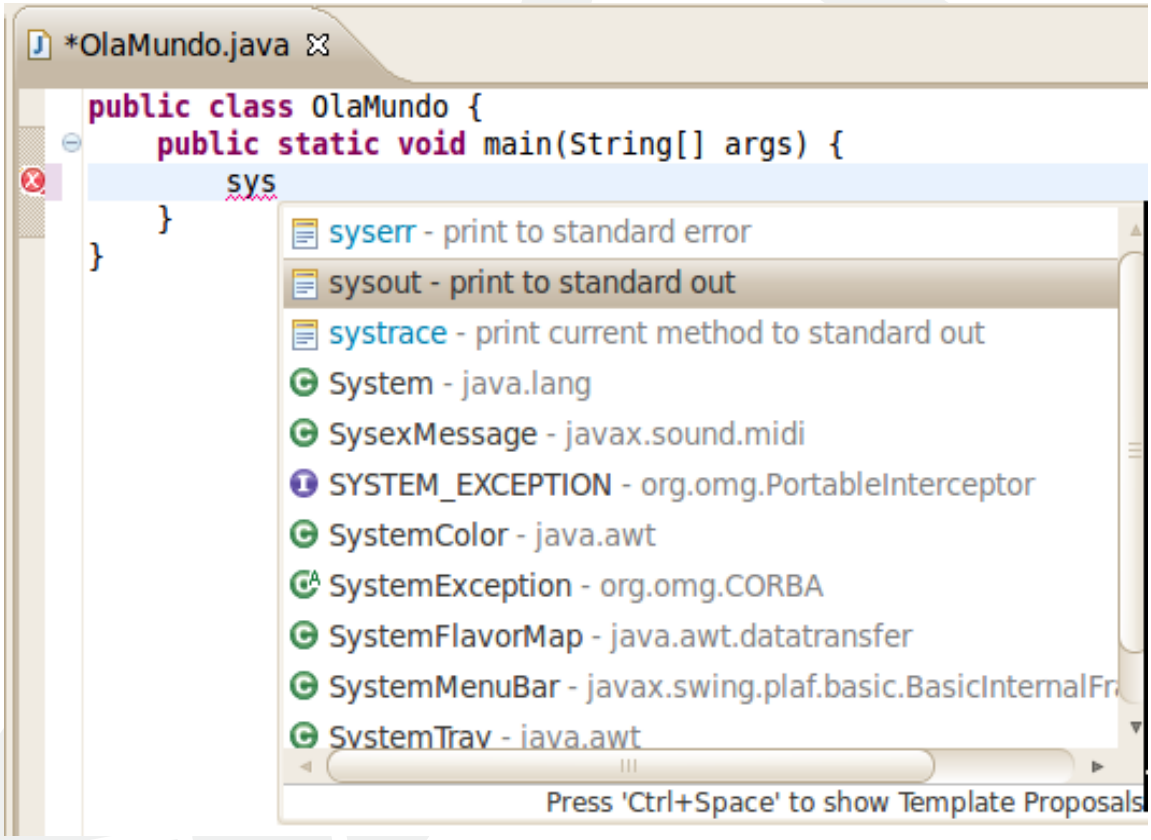
Um arquivo com o esqueleto da classe é criado na pasta **src** e automaticamente o Eclipse compila e salva o código compilado na pasta **bin**.

5.8 Criando o método main

O método main pode ser gerado utilizando **Content Assist** através do atalho **CTRL+ESPACO**. Basta digitar "main" seguido de CTRL+ESPACO e aceitar a sugestão do template para o método main.

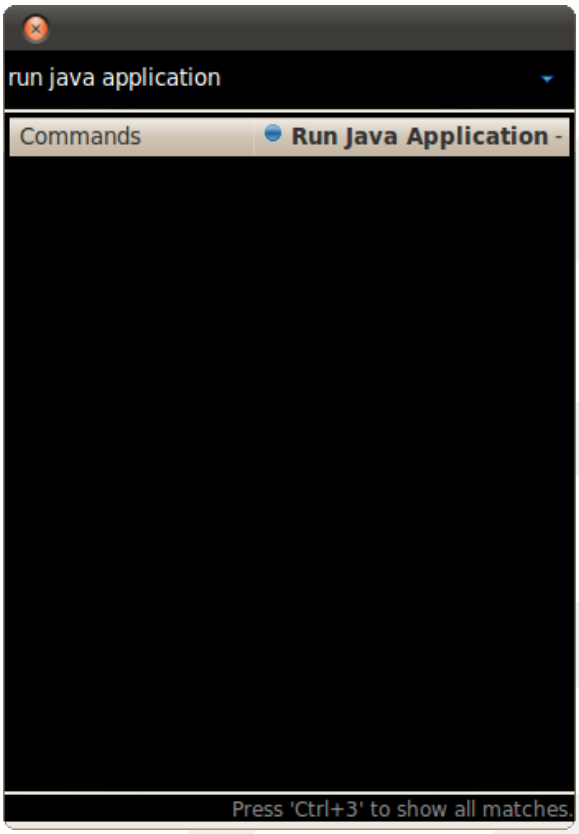


Dentro do método main podemos gerar o código necessário para imprimir uma mensagem na tela com o Content Assist. Basta digitar "sys" seguido de CTRL+ESPACO e escolher a sugestão adequada.



5.9 Executando uma classe

Podemos executar uma classe que possui main através do Quick Acess.

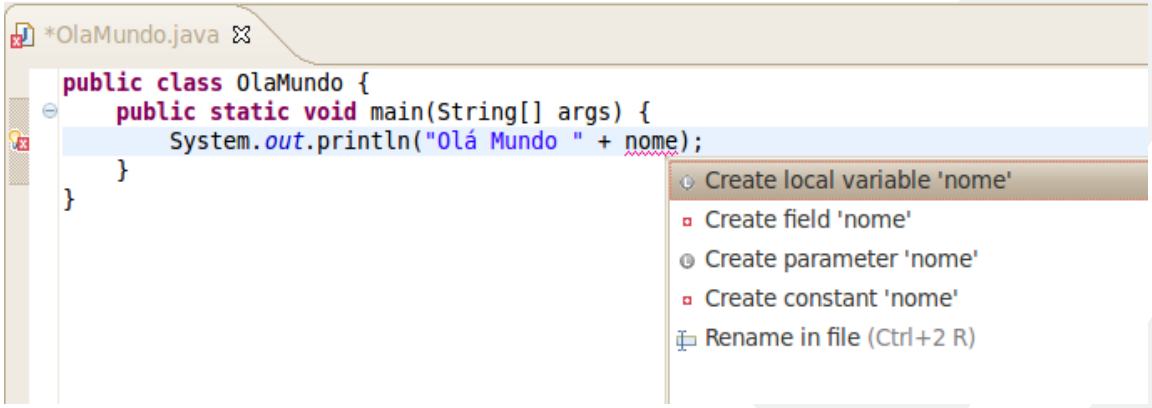


Também podemos utilizar o botão **run** na barra de ferramentas do eclipse.

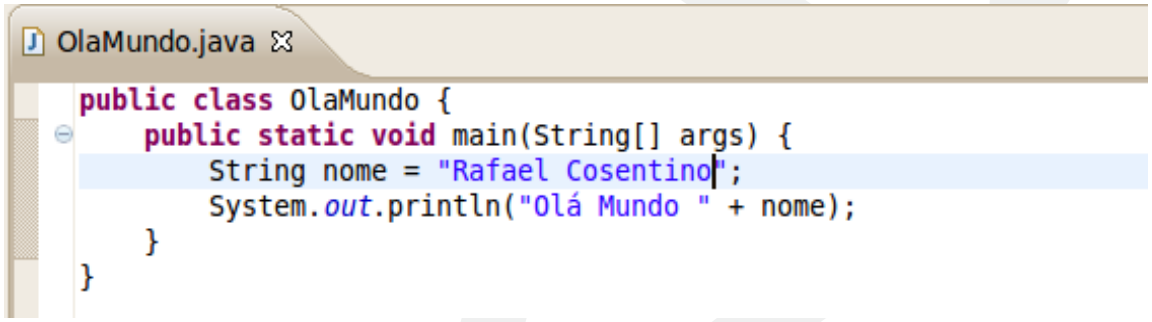
A saída do programa é mostrada na tela **Console**.

5.10 Corrigindo erros

Erros de compilação podem ser corrigidos com o **Quick Fix** através do atalho **CTRL+1**.



No exemplo, o Quick Fix gera uma variável local chamada NOME. Depois, basta definir um valor para essa variável.



5.11 Atalhos Úteis

CTRL+1 (Quick Fix) : Lista sugestões para consertar erros.

CTRL+3 (Quick Acess) : Lista para acesso rápido a comandos ou menus.

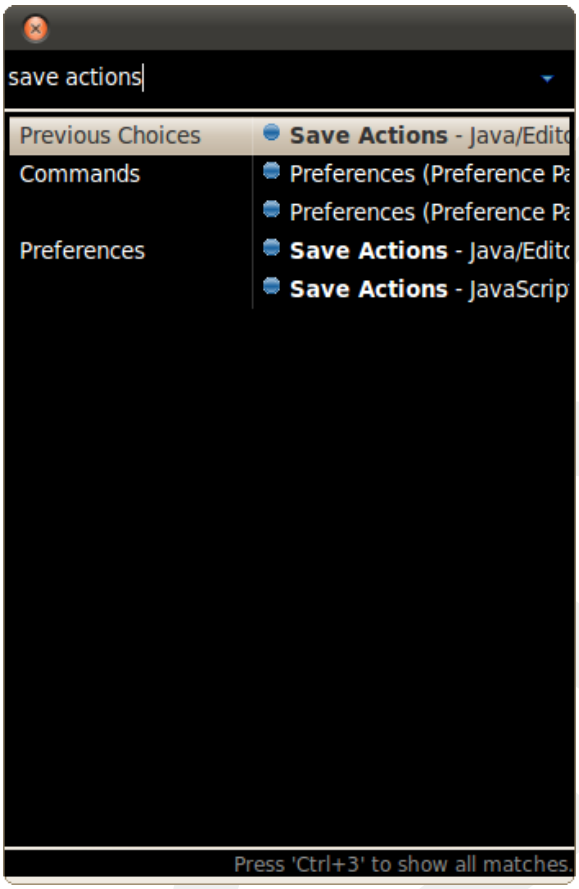
CTRL+ESPACO (Content Assist) : Lista sugestões para completar código.

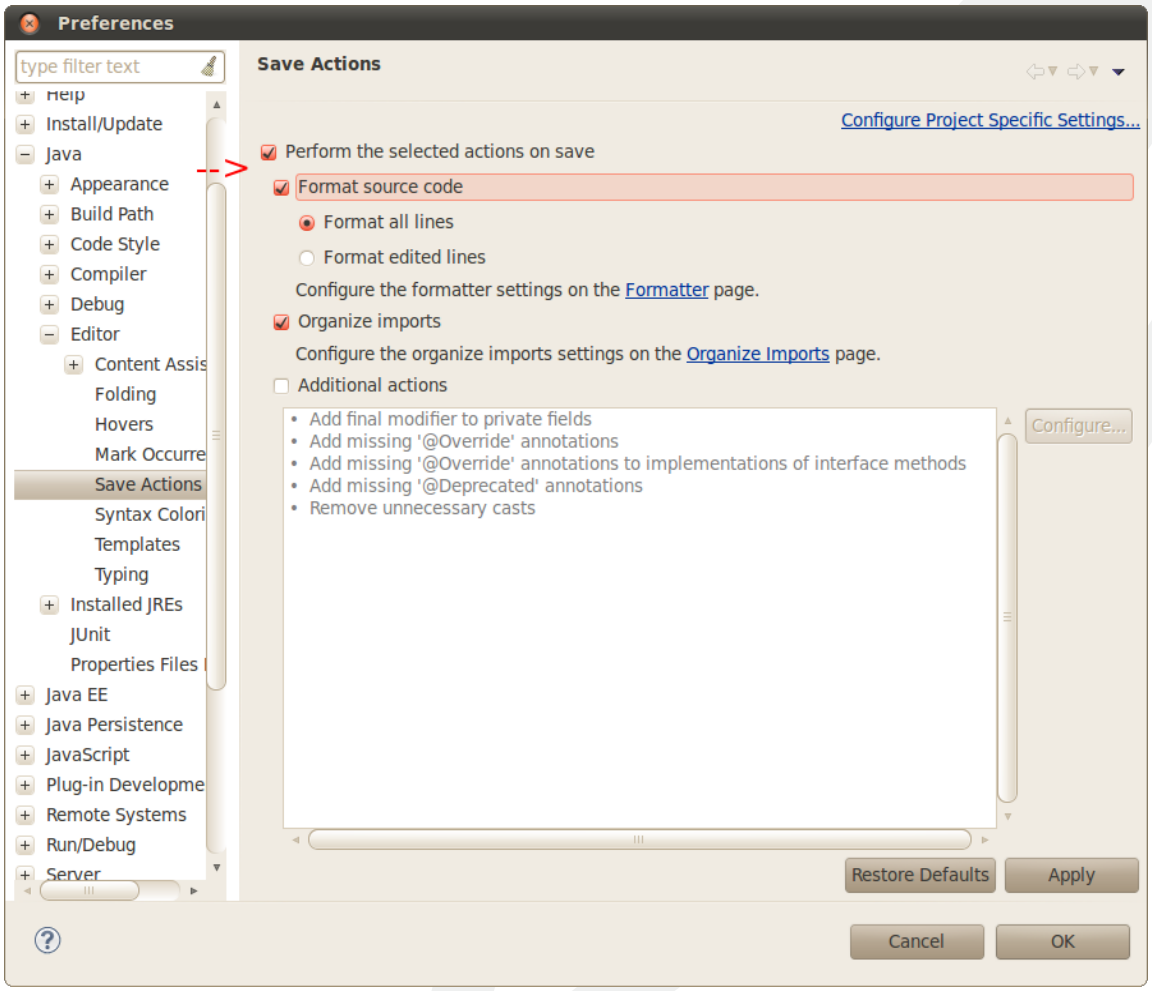
CTRL+SHIFT+F : Formata o código.

CTRL+ : Comenta o código selecionado.

5.12 Save Actions

Podemos escolher algumas ações para serem executadas no momento em que um arquivo com código java é salvo. Uma ação muito útil é a de formatar o código.


save actions
Previous Choices
Commands
Preferences
Save Actions - Java/Edit
Preferences (Preference Pa
Preferences (Preference Pa
Save Actions - Java/Edit
Save Actions - JavaScrip
Press 'Ctrl+3' to show all matches.


Preferences
type filter text
Help
Install/Update
Java
Appearance
Build Path
Code Style
Compiler
Debug
Editor
Content Assis
Folding
Hovers
Mark Occurre
Save Actions
Syntax Colori
Templates
Typing
Installed JREs
JUnit
Properties Files
Java EE
Java Persistence
JavaScript
Plug-in Developme
Remote Systems
Run/Debug
Server
Save Actions
Configure Project Specific Settings...
Perform the selected actions on save
Format source code
Format all lines
Format edited lines
Configure the formatter settings on the Formatter page.
Organize imports
Configure the organize imports settings on the Organize Imports page.
Additional actions
Add final modifier to private fields
Add missing '@Override' annotations
Add missing '@Override' annotations to implementations of interface methods
Add missing '@Deprecated' annotations
Remove unnecessary casts
Configure...
Restore Defaults
Apply
Cancel
OK

Capítulo 6

Atributos e Métodos de Classe

6.1 Atributos Estáticos

Num sistema bancário, provavelmente, criaríamos uma classe para especificar os objetos que representariam os funcionários do banco.

```
1  class Funcionario {
2      String nome;
3      double salario;
4
5      void aumentaSalario(double aumento) {
6          this.salario += aumento;
7      }
8  }
```

Supondo que o banco possui um piso salarial para os seus funcionários e que o sistema tem que guardar esse valor, poderíamos definir um atributo na classe FUNCIONARIO para tal propósito.

```
1  class Funcionario {
2      String nome;
3      double salario;
4      double pisoSalarial;
5
6      void aumentaSalario(double aumento) {
7          this.salario += aumento;
8      }
9  }
```

O atributo PISOSALARIAL é de instância, ou seja, cada objeto criado a partir da classe FUNCIONARIO teria o seu próprio atributo PISOSALARIAL. Porém, não faz sentido ter o valor do piso salarial repetido em todos os objetos já que esse valor é único para todos os funcionários.

Para que o atributo PISOSALARIAL não se repita em cada objeto da classe FUNCIONARIO, devemos torná-lo um atributo de classe ou invés de atributo de instância. Para isso devemos aplicar o modificador **static** na declaração do atributo.

```
1  class Funcionario {
2      String nome;
3      double salario;
4      static double pisoSalarial;
5
6      void aumentaSalario(double aumento) {
7          this.salario += aumento;
8      }
9  }
```

Um atributo de classe deve ser acessado através do nome da classe na qual ele foi declarado.

```
1  Funcionario.pisoSalarial = 2000;
```

6.2 Métodos Estáticos

Da mesma forma que definimos métodos para implementar as lógicas que manipulam os valores dos atributos de instância, podemos fazer o mesmo para os atributos de classe.

Suponha que o banco tenha um procedimento para reajustar o piso salarial baseado em uma taxa. Poderíamos definir um método na classe FUNCIONARIO para implementar esse reajuste.

```
1  void reajustaPisoSalarial(double taxa) {
2      Funcionario.pisoSalarial += Funcionario.pisoSalarial * taxa;
3  }
```

O método REAJUSTAPISOSALARIAL() é de instância, ou seja, deve ser invocado a partir da referência de um objeto da classe FUNCIONARIO. Porém, como o reajuste do piso salarial não depende dos dados de um funcionário em particular não faz sentido precisar de uma referência de um objeto da classe FUNCIONARIO para poder fazer o reajuste.

Neste caso, poderíamos definir o REAJUSTAPISOSALARIAL() como método de classe ao invés de método de instância através do modificador STATIC. Dessa forma, o reajuste poderia ser executado independentemente da existência de objetos da classe FUNCIONARIO.

```
1  static void reajustaPisoSalarial(double taxa) {
2      Funcionario.pisoSalarial += Funcionario.pisoSalarial * taxa;
3  }
```

Um método de classe deve ser invocado através do nome da classe na qual ele foi definido.

```
1  Funcionario.reajustaPisoSalarial(0.1);
```

6.3 Exercícios

1. Crie um projeto no eclipse chamado **Static**.

2. Crie uma classe chamada **Conta** no projeto STATIC. Defina um atributo de classe para contabilizar o número de objetos instanciados a partir da classe CONTA. Esse atributo deve ser incrementado toda vez que um objeto é criado. Você pode utilizar construtores para fazer o incremento.

```
1 public class Conta {
2     // ATRIBUTO DE CLASSE
3     static int contador;
4
5     // CONSTRUTOR
6     Conta() {
7         Conta.contador++;
8     }
9 }
```

3. Faça um teste criando dois objetos da classe CONTA e imprimindo o valor do contador de contas antes, entre e depois da criação dos objetos.

```
1 public class Teste {
2     public static void main(String[] args) {
3         System.out.println("Contador: " + Conta.contador);
4         Conta c1 = new Conta();
5         System.out.println("Contador: " + Conta.contador);
6         Conta c2 = new Conta();
7         System.out.println("Contador: " + Conta.contador);
8     }
9 }
```

4. O contador de contas pode ser utilizado para gerar um número único para cada conta. Acrescente na classe CONTA um atributo de instância para guardar o número das contas e implemente no construtor a lógica para gerar esses números de forma única através do contador de contas.

```
1  public class Conta {
2      // ATRIBUTO DE CLASSE
3      static int contador;
4
5      // ATRIBUTO DE INSTANCIA
6      int numero;
7
8      // CONSTRUTOR
9      Conta() {
10         Conta.contador++;
11         this.numero = Conta.contador;
12     }
13 }
```

5. Altere o teste para imprimir o número de cada conta criada.

```
1  public class Teste {
2      public static void main(String[] args) {
3          System.out.println("Contador: " + Conta.contador);
4          Conta c1 = new Conta();
5          System.out.println("Numero da primeira conta: " + c1.numero);
6
7          System.out.println("Contador: " + Conta.contador);
8          Conta c2 = new Conta();
9          System.out.println("Numero da segunda conta: " + c2.numero);
10
11         System.out.println("Contador: " + Conta.contador);
12     }
13 }
```

6. (Opcional) Defina um método de classe na classe CONTA para zerar o contador e imprimir o total de contas anterior.

```
1  // METODO DE CLASSE
2  static void zeraContador() {
3      System.out.println("Contador: " + Conta.contador);
4      System.out.println("Zerando o contador de contas...");
5      Conta.contador = 0;
6  }
```

7. (Opcional) Altere o teste para utilizar o método ZERACONTADOR().

```
1  public class Teste {
2      public static void main(String[] args) {
3          Conta c1 = new Conta();
4          System.out.println("Numero da primeira conta: " + c1.numero);
5
6          Conta c2 = new Conta();
7          System.out.println("Numero da segunda conta: " + c2.numero);
8
9          Conta.zeraContador();
10     }
11 }
```

8. (Opcional) Crie uma classe para modelar os funcionários do banco. Defina nessa classe um atributo para armazenar o piso salarial.

9. (Opcional) Faça um teste para verificar o funcionamento do piso salarial.

10. (Opcional) Defina um método para reajustar o piso salarial a partir de uma taxa.

Capítulo 7

Encapsulamento

7.1 Atributos Privados

No sistema do banco, cada objeto da classe FUNCIONARIO possuiria um atributo para guardar o salário do funcionário que ele representa.

```
1 class Funcionario {
2     double salario;
3 }
```

O atributo SALARIO de um objeto da classe FUNCIONARIO pode ser acessado ou modificado por código escrito em qualquer classe do mesmo pacote da classe FUNCIONARIO (Veremos pacotes posteriormente). Portanto, o controle do atributo SALARIO é descentralizado dificultando a detecção de erros relacionados à manipulação dos salários dos funcionários.

Para obter um controle centralizado, podemos fazer o atributo SALARIO ser **privado** na classe FUNCIONARIO e criar métodos para implementar as lógicas que utilizam o valor desse atributo.

```
1 class Funcionario {
2     private double salario;
3 
4     void aumentaSalario(double aumento) {
5         // lógica para aumentar o salário
6     }
7 }
```

Um atributo privado só pode ser acessado ou alterado por código escrito na classe na qual ele foi definido. Se algum código fora da classe FUNCIONARIO tenta acessar ou alterar o valor do atributo privado SALARIO de um objeto da classe FUNCIONARIO um erro de compilação é gerado.

Definir todos os atributos como privados e métodos para implementar as lógicas de acesso e alteração é quase uma regra da orientação a objetos. O intuito é ter sempre um controle centralizado do dados dos objetos para facilitar a manutenção do sistema.

7.2 Métodos Privados

O papel de alguns métodos de uma classe pode ser auxiliar outros métodos da mesma classe. E muitas vezes, não é correto chamar esses métodos auxiliares diretamente.

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos fazê-los privados.

```
1  class Conta {
2      private double saldo;
3
4      void deposita(double valor) {
5          this.saldo += valor;
6          this.descontaTarifa();
7      }
8
9      void saca(double valor) {
10         this.saldo -= valor;
11         this.descontaTarifa();
12     }
13
14     private descontaTarifa(){
15         this.saldo -= 0.1;
16     }
17 }
```

No exemplo acima, o método DESCONTATARIFA() é um método auxiliar dos métodos DEPOSITA() e SACA(). Além disso, ele não deve ser chamado diretamente pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

7.3 Métodos Públicos

Os métodos que devem ser acessados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade **public**.

```
1  class Conta {
2      private double saldo;
3
4      public void deposita(double valor) {
5          this.saldo += valor;
6          this.descontaTarifa();
7      }
8
9      public void saca(double valor) {
10         this.saldo -= valor;
11         this.descontaTarifa();
12     }
13
14     private descontaTarifa(){
15         this.saldo -= 0.1;
16     }
17 }
```

7.4 Implementação e Interface de Uso

Dentro de um sistema orientado a objetos cada objeto realiza um conjunto de tarefas de acordo com as suas responsabilidades. Por exemplo, os objetos da classe CONTA realizam as operações de sacar, depositar, transferir ou gerar extrato.

Para descobrir o que um objeto pode fazer basta olhar para as assinaturas dos métodos públicos definidos na classe desse objeto. A assinatura de um método é composta pelo seu tipo de retorno, seu nome e seus parâmetros. As assinaturas dos métodos públicos de um objeto formam a sua **interface de uso**.

Por outro lado, para descobrir **como** um objeto da classe CONTA realiza as suas operações devemos observar o corpo de cada um dos métodos da classe CONTA. Os corpos dos métodos definem a **implementação** das operações dos objetos.

7.5 Escondendo a implementação

Uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento. A manutenção é favorecida pelo encapsulamento porque quando a implementação de um objeto tem que ser alterada basta modificar a classe do objeto. O desenvolvimento é favorecido pelo encapsulamento pois podemos separar o trabalho dos desenvolvedores de forma mais independente.

Exemplos

O conceito do encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostraremos alguns desses exemplos para esclarecer melhor a ideia.

Celular

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus do celular forma a **interface de uso** do mesmo. Em outras palavras, o usuário interage com o celular através dos botões, da tela e dos menus. Os dispositivos internos do celular e os processos que transformam o som capturado pelo microfone em ondas que podem ser transmitidas para uma antena da operadora de telefonia móvel formam a **implementação** do celular.

Do ponto de vista do usuário do celular, para fazer uma ligação, basta digitar o número do telefone desejado e clicar no botão que efetua ligação. Porém, diversos processos complexos são realizados pelo celular para que as pessoas possam conversar através dele. Se os usuários tivessem que ter conhecimento de todo o funcionamento interno do celular certamente a maioria das pessoas não teria celular.

No contexto da orientação a objetos, aplicamos o encapsulamento para criar objetos mais simples de serem utilizados em qualquer parte do sistema.

Carro

A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do cambio, etc).

A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Nos carros mais antigos, o dispositivo interno que leva o combustível para o motor é o carburador. Nos carros mais novos, o carburador foi substituído pelo sistema de injeção eletrônica. Inclusive, algumas oficinas especializadas substituem o carburador pelo sistema de injeção eletrônica. Essa alteração na implementação do carro não afeta a maneira que o motorista dirige. Todo mundo que sabe dirigir um carro com carburador sabe dirigir um carro com injeção eletrônica.

Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa. Quando a interface de uso do carro é alterada, a maneira de dirigir é afetada fazendo com que as pessoas que sabem dirigir tenham que se adaptar.

No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam "escondidas"então podemos modificá-las sem afetar a maneira de utilizar esses objetos. Por outro lado, se alterarmos a interface de uso que está exposta afetaremos a maneira de usar os objetos. Suponha uma troca de nome de método público, todas as chamadas a esse método devem ser alteradas.

Máquinas de Porcarias

Estamos acostumados a utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Em geral, essas máquinas oferecem uma interface de uso composta por:

- Entradas para moedas ou cédulas.
- Botões para escolher o produto desejado.
- Saída do produto.
- Saída para o troco (opcional).

Normalmente, essas máquinas são extremamente protegidas, para garantir que nenhum usuário mal intencionado ou não tente alterar a implementação da máquina, ou seja, tente alterar como a máquina funciona por dentro.

Levando essa ideia para um sistema orientado a objetos, um objeto deve ser bem protegido para que outros objetos não prejudiquem o funcionamento interno do objeto.

7.6 Acesso e Alteração de atributos

Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Assim, os atributos não podem ser acessados ou alterados por código escrito fora da classe na qual eles

foram definidos. Porém, muitas vezes, as informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema. Nesse caso, podemos disponibilizar métodos para consultar os valores dos atributos.

```
1  class Cliente {
2      private String nome;
3
4      public String consultaNome() {
5          return this.nome;
6      }
7  }
```

Da mesma forma, eventualmente, queremos alterar o valor de um atributo a partir de qualquer lugar do sistema. Nesse caso, também poderíamos criar um método para essa tarefa.

```
1  class Cliente {
2      private String nome;
3
4      public void alteraNome(String nome){
5          this.nome = nome;
6      }
7  }
```

Muitas vezes, é necessário consultar e alterar o valor de um atributo a partir de qualquer lugar do sistema. Nessa situação, podemos definir os dois métodos discutidos anteriormente. Mas, o que é melhor? Criar os dois métodos (um de leitura outro de escrita) ou deixar o atributo público?

Utilizando os métodos, podemos controlar como as alterações ou as consultas são feitas. Ou seja, temos um controle maior.

Na linguagem Java, para facilitar o trabalho em equipe, os nomes dos métodos discutidos anteriormente são padronizados. Os nomes dos métodos que consultam os valores dos atributos possuem o prefixo "get"seguido do nome do atributo. Os que alteram os valores dos atributos possuem o prefixo "set"seguido do nome do atributo.

```
1  class Cliente {
2      private String nome;
3
4      public String getNome() {
5          return this.nome;
6      }
7
8      public void setNome(String nome) {
9          this.nome = nome;
10     }
11 }
```

7.7 Exercícios

1. Crie um projeto no eclipse chamado **Encapsulamento**.

2. Defina uma classe para representar os funcionários do banco com um atributo para guardar

os salários e outro para os nomes.

```
1  class Funcionario {
2      double salario;
3      String nome;
4  }
```

3. Teste a classe FUNCIONARIO criando um objeto e manipulando os atributos.

```
1  class Teste {
2      public static void main(String[] args) {
3          Funcionario f = new Funcionario();
4
5          f.nome = "Rafael Cosentino";
6          f.salario = 2000;
7
8          System.out.println(f.nome);
9          System.out.println(f.salario);
10     }
11 }
```

Perceba que a classe TESTE pode acessar ou modificar os valores dos atributos de um objeto da classe FUNCIONARIO. Execute o teste!

4. Aplique a ideia do encapsulamento tornando os atributos definidos na classe FUNCIONARIO privados.

```
1  class Funcionario {
2      private double salario;
3      private String nome;
4  }
```

Observe que a classe TESTE não compila mais. Lembre que um atributo privado só pode ser acessado por código escrito na própria classe do atributo.

5. Crie métodos de acessado com nome padronizados para os atributos da classe FUNCIONARIO.

```
1  class Funcionario {
2      private double salario;
3      private String nome;
4
5      public double getSalario() {
6          return this.salario;
7      }
8
9      public String getNome() {
10         return this.nome;
11     }
12
13     public void setSalario(double salario) {
14         this.salario = salario;
15     }
16
17     public void setNome(String nome) {
18         this.nome = nome;
19     }
20 }
```

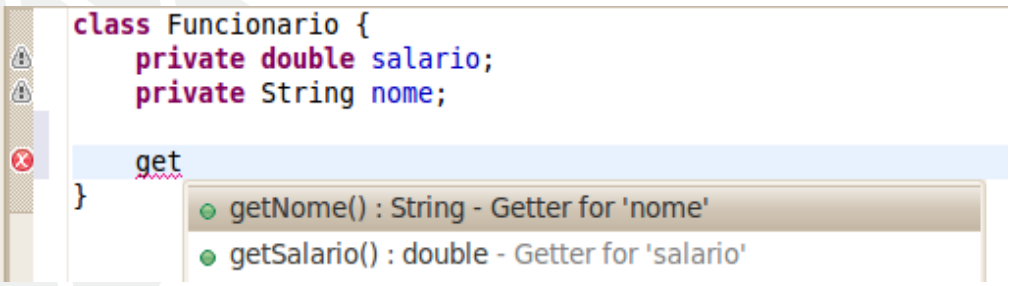
6. Altere a classe TESTE para que ela utilize os métodos de acesso ao invés de manipular os atributos do objeto funcionário diretamente.

```
1  class Teste {
2      public static void main(String[] args) {
3          Funcionario f = new Funcionario();
4
5          f.setNome("Rafael Cosentino");
6          f.setSalario(2000);
7
8          System.out.println(f.getNome());
9          System.out.println(f.getSalario());
10     }
11 }
```

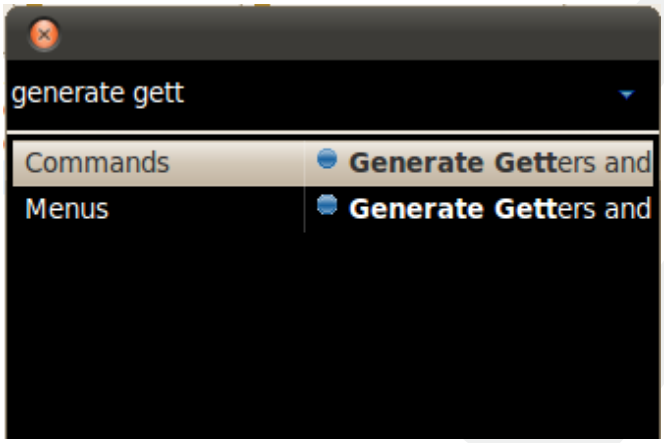
Execute o teste!

7. Gere os getters e setters com os recursos do eclipse.

Digite get ou set seguidos de CTRL+ESPAÇO para completar o código.



Outra possibilidade é utilizar o Quick Acess para executar o comando **generate getters and setters**


generate gett
Commands
Menus
Generate Getters and
Generate Getters and

Generate Getters and Setters
Select getters and setters to create:
nome
salario
Select All
Deselect All
Select Getters
Select Setters

Capítulo 8

Herança

8.1 Reutilização de Código

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Suponha que todos os serviços possuem basicamente: o cliente que contratou, o funcionário responsável pela contratação e data de contratação. O sistema do banco deve estar preparado para gerenciar esses serviços.

Com o intuito de ser produtivo, a modelagem dos serviços do banco deve diminuir a repetição de código. A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito **Don't Repeat Yourself**. Em outras palavras, devemos minimizar ao máximo a utilização do "copiar e colar". O aumento da produtividade e a diminuição do custo de manutenção são as principais motivações do **DRY**.

Em seguida, vamos discutir algumas modelagens possíveis para os serviços do banco. Buscaremos seguir a ideia do *DRY* na criação dessas modelagens.

Uma classe para todos os serviços

Poderíamos definir apenas uma classe para modelar todos os tipos de serviços que o banco oferece.

```
1 class Servico {
2     private Cliente contratante;
3     private Funcionario responsavel;
4     private String dataDeContratacao;
5
6     // getters e setters
7 }
```

Empréstimo

O empréstimo é um dos serviços que o banco oferece. Quando um cliente contrata esse serviço são definidos o valor e a taxa de juros mensal do empréstimo. Devemos acrescentar dois atributos na classe SERVICO, um para o valor outro para a taxa de juros do serviço de empréstimo.

```
class Servico {
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataDeContratacao;

    private double valor;
    private double taxa;

    // getters e setters
}
```

Seguro de veículos

Outro serviço oferecido pelo banco é o seguro de veículos. Para esse serviço devem ser definidas as seguintes informações: veículo segurado, valor do seguro e a franquia. Devemos adicionar três atributos na classe SERVICO.

```
class Servico {
    // GERAL
    private Cliente contratante;
    private Funcionario responsavel;
    private String dataDeContratacao;

    // EMPRESTIMO
    private double valor;
    private double taxa;

    // SEGURO DE VEICULO
    private Veiculo veiculo;
    private double valorDoSeguroDeVeiculo;
    private double franquia;

    // getters e setters
}
```

Apesar de seguir a ideia do *DRY*, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento. Supondo que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços então eles provavelmente codificariam a mesma classe concorrentemente. Além disso, os desenvolvedores, principalmente os recém chegados no projeto do banco, ficariam confusos com o código extenso da classe SERVICO.

Outro problema é que um objeto da classe SERVICO possui atributos para todos os serviços que o banco oferece mas na verdade deveria possuir apenas os atributos relacionados a um serviço. Do ponto de vista de performance, essa abordagem causaria um consumo desnecessário de memória.

Uma classe para cada serviço

Para modelar melhor os serviços, evitando uma quantidade grande de atributos e métodos desnecessários, criaremos uma classe para cada serviço.

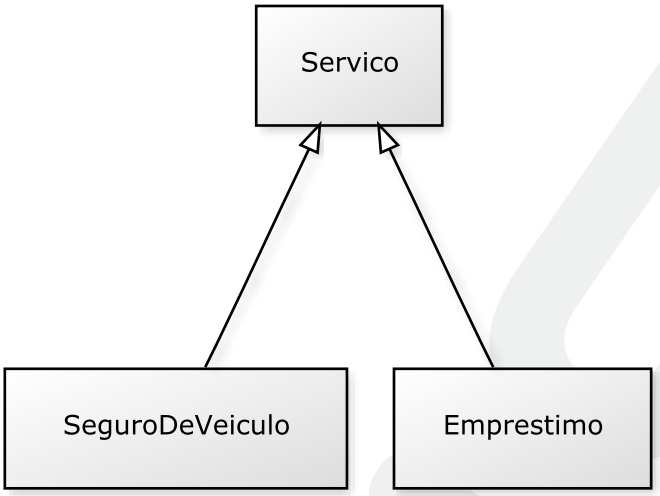
```
1  class SeguroDeVeiculo {
2      // GERAL
3      private Cliente contratante;
4      private Funcionario responsavel;
5      private String dataDeContratacao;
6
7      // SEGURO DE VEICULO
8      private Veiculo veiculo;
9      private double valorDoSeguroDeVeiculo;
10     private double franquia;
11
12     // getters e setters
13 }
```

```
1  class Emprestimo {
2      // GERAL
3      private Cliente contratante;
4      private Funcionario responsavel;
5      private String dataDeContratacao;
6
7      // EMPRESTIMO
8      private double valor;
9      private double taxa;
10
11     // getters e setters
12 }
```

Criar uma classe para cada serviço, torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do *DRY*. Além disso, qualquer alteração que deva ser realizada em todos os serviços precisa ser implementada em cada uma das classes.

Uma classe genérica e várias específicas

Na modelagem dos serviços do banco, podemos aplicar um conceito da orientação a objetos chamado **Herança**. A ideia é reutilizar o código de uma determinada classe em outras classes. Teríamos a classe SERVICO com os atributos e métodos que todos os serviços devem ter e uma classe para cada serviço com os atributos e métodos específicos do determinado serviço. As classes específicas seriam "ligadas"de alguma forma à classe SERVICO para reaproveitar o código nela definido. Esse relacionamento entre as classes é representado em UML pelo diagrama abaixo:



Os objetos das classes específicas: EMPRESTIMO e SEGURODEVEICULO) possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe SERVICO.

```
1 Emprestimo e = new Emprestimo();
2 e.setDataDeContratacao("10/10/2010");
3 e.setValor(10000);
```

As classes específicas são vinculadas com a classe genérica utilizando o comando *extends* e não precisam redefinir o conteúdo já declarado na classe genérica.

```
1 class Servico {
2     private Cliente contratante;
3     private Funcionario responsavel;
4     private String dataDeContratacao;
5 }
```

```
1 class Emprestimo extends Servico {
2     private double valor;
3     private double taxa;
4 }
```

```
1 class SeguroDeVeiculo extends Servico {
2     private Veiculo veiculo;
3     private double valorDoSeguroDeVeiculo;
4     private double franquia;
5 }
```

A classe genérica é denominada de **super classe**, **classe base** ou **classe mãe**. As classes específicas são denominadas **sub classes**, **classes derivadas** ou **classes filhas**.

8.2 Reescrita de Método

Preço Fixo

Suponha que todo serviço do banco possui uma taxa administrativa que deve ser paga pelo cliente que contratar o serviço. Inicialmente, vamos considerar que o valor dessa taxa é igual para todos os serviços do banco. Neste caso, poderíamos implementar um método na classe SERVICO para calcular o valor da taxa. Este método será reaproveitado por todas as classes que herdam da classe SERVICO.

```
1 class Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 10;
6     }
7 }
```

Alguns preços diferentes

Suponha que o valor da taxa administrativa do serviço de empréstimo é diferente dos outros serviços pois ele é calculado a partir do valor emprestado ao cliente. Como esta lógica é específica para o serviço de de empréstimo devemos acrescentar um método para implementar esse cálculo na classe EMPRESTIMO.

```
1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxaDeEmprestimo() {
5         return this.valor * 0.1;
6     }
7 }
```

Para os objetos da classe EMPRESTIMO devemos chamar o método CALCULATAXADEEMPRESTIMO(). Para todos os outros serviços devemos chamar o método CALCULATAXA().

Mesmo assim, nada impediria que o método CALCULATAXA() fosse chamado num objeto da classe EMPRESTIMO, pois ela herda esse método da classe SERVICO. Dessa forma, corremos o risco de alguém se equivocar chamando o método errado.

Na verdade, gostaríamos de "substituir"a implementação do método CALCULATAXA() herdado da classe SERVICO na classe EMPRESTIMO para evitar uma chamada errada de método. Para isso, basta escrever o método CALCULATAXA() também na classe EMPRESTIMO com a mesma assinatura que ele possui na classe SERVICO.

```
1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return this.valor * 0.1;
6     }
7 }
```

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Em outras palavras, se o método chamado existe na classe filha ele será chamado, caso contrário o método será procurado na classe mãe.

Fixo + Específico

Suponha que o preço de um serviço é a soma de um valor fixo mais um valor que depende do tipo do serviço. Por exemplo, o preço do serviço de empréstimo é 5 reais mais uma porcentagem do valor emprestado ao cliente. O preço do serviço de seguro de veículo 5 reais mais uma porcentagem do valor do veículo segurado. Em cada classe específica, podemos reescrever o método CALCULATAXA().

```
1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 5 + this.valor * 0.1;
6     }
7 }
```

```
1 class SeguraDeVeiculo extends Servico {
2     // ATRIBUTOS
3
4     public double calculaTaxa() {
5         return 5 + this.veiculo.getValor() * 0.05;
6     }
7 }
```

Se o valor fixo dos serviços for atualizado todas as classes específicas devem ser modificadas. Outra alternativa, seria criar um método na classe SERVICO para calcular o valor fixo de todos os serviços e chamá-lo dos métodos reescritos nas classes específicas.

```
1 class  Servico {
2     public double calculaTaxa() {
3         return  5 ;
4     }
5 }
```

```
1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public override double calculaTaxa() {
5         return  super.calculaValor() + this.valor * 0.1;
6     }
7 }
```

8.3 Construtores e Herança

Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal. Pelo menos um construtor de cada classe de uma mesma sequência hierárquica deve ser chamado ao instanciar um objeto. Por exemplo, quando um objeto da classe EMPRESTIMO, pelo menos um construtor da própria classe EMPRESTIMO e um da classe SERVICO devem ser executados. Além disso, os construtores da classes mais genéricas são chamados antes dos construtores das classes específicas.

```
1 class  Servico {
2     public Servico() {
3         System.out.println("Servico");
4     }
5 }
```

```
1 class Emprestimo extends Servico {
2     // ATRIBUTOS
3
4     public Emprestimo() {
5         System.out.println("Emprestimo");
6     }
7 }
```

Por padrão, todo construtor chama o construtor sem argumentos da classe mãe se não existir nenhuma chamada de construtor explícita.

8.4 Exercícios

1. Crie um projeto no eclipse chamado **Heranca**.

2. Defina uma classe para modelar os funcionários do banco. Sabendo que todo funcionário possui nome e salário. Inclua os getters e setters dos atributos.

```
1 class Funcionario {
2     private String nome;
3     private double salario;
4
5     // GETTERS AND SETTERS
6 }
```

3. Crie uma classe para cada tipo específico de funcionário herdando da classe FUNCIONARIO. Suponha somente três tipos específicos de funcionários: gerentes, telefonistas e secretarias. Os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. As telefonistas possuem um código de estação de trabalho. As secretarias possuem um número de ramal.

```
class Gerente extends Funcionario {
    private String usuario;
    private String senha;

    // GETTERS AND SETTERS
}
```

```
class Telefonista extends Funcionario {
    private int estacaoDeTrabalho;

    // GETTERS AND SETTERS
}
```

```
class Secretaria extends Funcionario {
    private int ramal;

    // GETTERS AND SETTERS
}
```

4. Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: GERENTE, TELEFONISTA e SECRETARIA.

```
class TestaFuncionarios {
    public static void main(String[] args) {
        Gerente g = new Gerente();
        g.setNome("Rafael Cosentino");
        g.setSalario(2000);
        g.setUsuario("rafael.cosentino");
        g.setSenha("12345");

        Telefonista t = new Telefonista();
        t.setNome("Carolina Mello");
        t.setSalario(1000);
        t.setEstacaoDeTrabalho(13);

        Secretaria s = new Secretaria();
        s.setNome("Tatiane Andrade");
        s.setSalario(1500);
        s.setRamal(198);

        System.out.println("GERENTE");
        System.out.println("Nome: " + g.getNome());
        System.out.println("Salário: " + g.getSalario());
        System.out.println("Usuário: " + g.getUsuario());
        System.out.println("Senha: " + g.getSenha());

        System.out.println("TELEFONISTA");
        System.out.println("Nome: " + t.getNome());
        System.out.println("Salário: " + t.getSalario());
        System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());

        System.out.println("SECRETARIA");
        System.out.println("Nome: " + s.getNome());
        System.out.println("Salário: " + s.getSalario());
        System.out.println("Ramal: " + s.getRamal());

    }
}
```

Execute o teste!

5. Suponha que todos os funcionários possuam uma bonificação de 10% do salário. Acrescente um método na classe FUNCIONARIO para calcular essa bonificação.

```
class Funcionario {
    private String nome;
    private double salario;

    public double calculaBonificacao() {
        return this.salario * 0.1;
    }

    // GETTERS AND SETTERS
}
```

6. Altere a classe TESTAFUNCIONARIOS para imprimir a bonificação de cada funcionário além dos dados que já foram impressos. Depois, execute o teste novamente.

7. Suponha que os gerentes possuam uma bonificação maior que os outros funcionários. Reescreva o método CALCULABONIFICACAO() na classe GERENTE. Depois, execute o teste novamente.

```
1  class Gerente extends Funcionario {
2      private String usuario;
3      private String senha;
4
5      public double calculaBonificacao() {
6          return this.getSalario() * 0.6 + 100;
7      }
8
9      // GETTERS AND SETTERS
10 }
```

8. (Opcional) Defina na classe FUNCIONARIO um método para imprimir alguns dados na tela.

```
1  class Funcionario {
2      private String nome;
3      private double salario;
4
5      public double calculaBonificacao() {
6          return this.salario * 0.1;
7      }
8
9      public void mostraDados() {
10         System.out.println("Nome: " + this.nome);
11         System.out.println("Salário: " + this.salario);
12         System.out.println("Bonificação: " + this.calculaBonificacao());
13     }
14
15     // GETTERS AND SETTERS
16 }
```

9. (Opcional) Reescreva o método MOSTRADADOS() nas classes GERENTE, TELEFONISTA e SECRETARIA para acrescentar a impressão dos dados específicos de cada tipo de funcionário. Veja o exemplo abaixo:

```
1  class Gerente extends Funcionario {
2      private String usuario;
3      private String senha;
4
5      public double calculaBonificacao() {
6          return this.getSalario() * 0.6 + 100;
7      }
8
9      public void mostraDados() {
10         super.mostraDados();
11         System.out.println("Usuário: " + this.usuario);
12         System.out.println("Senha: " + this.senha);
13     }
14
15     // GETTERS AND SETTERS
16 }
```

10. (Opcional) Modifique a classe TESTAFUNCIONARIO para utilizar o método MOSTRADADOS().

```
1  class TestaFuncionarios {
2      public static void main(String[] args) {
3          Gerente g = new Gerente();
4          g.setNome("Rafael Cosentino");
5          g.setSalario(2000);
6          g.setUsuario("rafael.cosentino");
7          g.setSenha("12345");
8
9          Telefonista t = new Telefonista();
10         t.setNome("Carolina Mello");
11         t.setSalario(1000);
12         t.setEstacaoDeTrabalho(13);
13
14         Secretaria s = new Secretaria();
15         s.setNome("Tatiane Andrade");
16         s.setSalario(1500);
17         s.setRamal(198);
18
19         System.out.println("GERENTE");
20         g.mostraDados();
21
22         System.out.println("TELEFONISTA");
23         t.mostraDados();
24
25         System.out.println("SECRETARIA");
26         s.mostraDados();
27     }
28 }
```

11. (Opcional) Defina classes para modelar diversos tipos de contas bancárias: poupança, corrente, conta salário, entre outros.

Capítulo 9

Polimorfismo

Os clientes de um banco podem consultar algumas informações das suas contas através de extratos impressos nos caixas eletrônicos. Eventualmente, poderíamos criar uma classe para definir um tipo de objeto capaz de gerar extratos de diversos tipos.

```
1  class GeradorDeExtrato {
2
3      public void imprimeExtratoBasico(ContaPoupanca cp) {
4          SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
5          Date agora = new Date();
6
7          System.out.println("DATA: "  + sdf.format(agora));
8          System.out.println("SALDO: "  + cp.getSaldo());
9      }
10 }
```

O método IMPRIMEEXTRATOBASICO() gera extratos básicos somente de contas do tipo poupança. O sistema do banco deve ser capaz de gerar extratos básicos para os outros tipos de contas também. Poderíamos implementar outros métodos na classe GERADORDEEXTRATO para trabalhar com os outros tipos de conta, um para cada tipo de conta.

```
class GeradorDeExtrato {

    public void imprimeExtratoBasico(ContaPoupanca cp) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + cp.getSaldo());
    }

    public void imprimeExtratoBasico(ContaCorrente cc) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + cc.getSaldo());
    }

    public void imprimeExtratoBasico(ContaSalario cs) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + cs.getSaldo());
    }
}
```

Como o conteúdo do extrato básico é o mesmo independentemente do tipo de conta então os métodos da classe GERADORDEEXTRATO seriam praticamente iguais. Qualquer tipo de mudança no conteúdo do extrato básico causaria modificações em diversos métodos.

Além disso, se o banco definir um novo tipo de conta então um método praticamente idêntico aos que já existem teria que ser adicionado na classe GERADORDEEXTRATO. Analogamente, se o banco extinguir um tipo de conta então o método correspondente de gerar extrato básico deve ser retirado do gerador.

9.1 Modelagem das contas

Com o intuito inicial de reutilizar código podemos modelar os diversos tipos de contas do banco utilizando o conceito de herança.

```
class Conta {
    private double saldo;

    // MAIS ATRIBUTOS E MÉTODOS
}
```

```
class ContaPoupanca extends Conta {
    private int diaDoAniversario;

    // MAIS ATRIBUTOS E MÉTODOS
}
```

```
class ContaCorrente extends Conta {
    private double limite;

    // MAIS ATRIBUTOS E MÉTODOS
}
```

9.2 É UM (extends)

Além de gerar reaproveitamento de código, a utilização de herança permite que objetos criados a partir das classes específicas sejam tratados como objetos da classe genérica. Em outras palavras, a herança entre as classes que modelam as contas permite que objetos criados a partir das classes CONTAPOUPANCA ou CONTACORRENTE sejam tratados como objetos da classe CONTA.

No código da classe CONTAPOUPANCA utilizamos a palavra **extends**. Ela pode ser interpretada com a expressão: **É UM** ou **É UMA**.

```
class ContaPoupanca extends Conta
//TODA ContaPoupanca É UMA Conta
```

Como está explícito no código que toda conta poupança é uma conta então podemos criar um objeto da classe CONTAPOUPANCA e tratá-lo como um objeto da classe CONTA.

```
// Criando um objeto da classe ContaPoupanca
ContaPoupanca cp = new ContaPoupanca();

// Tratando o objeto como um objeto da classe Conta
Conta c = cp;
```

Em alguns lugares do sistema do banco será mais vantajoso tratar um objeto da classe CONTAPOUPANCA como um objeto da classe CONTA.

9.3 Melhorando o gerador de extrato

Do ponto de vista do gerador de extrato, o tipo de conta não faz diferença na hora de gerar extratos básicos. Então, ao invés de criar um método de gerar extratos básicos para cada tipo de conta, vamos criar um método genérico que aceite qualquer tipo de conta como parâmetro.

```
class GeradorDeExtrato {

    public void imprimeExtratoBasico(Conta c) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + c.getSaldo());
    }
}
```

Como o método IMPRIMEEXTRATOBASICO() recebe uma conta como parâmetro ele pode receber objetos da classe CONTAPOUPANCA, CONTACORRENTE ou qualquer outra classe que herde direta ou indiretamente da classe CONTA.

A capacidade de tratar objetos criados a partir das classes específicas como objetos de uma classe genérica é chamada de **Polimorfismo**.

Aplicando a ideia do polimorfismo no gerador de extratos melhoramos a manutenibilidade da classe GERADORDEEXTRATO pois agora qualquer alteração no formato ou no conteúdo dos extratos básicos causa modificações em apenas um método. Fora isso, novos tipos de contas podem ser criadas sem a necessidade de qualquer alteração na classe GERADORDEEXTRATO. Analogamente, se algum tipo de conta for extinto nada precisará ser modificado na classe GERADORDEEXTRATO.

9.4 Exercícios

1. Crie um projeto no eclipse chamado **Polimorfismo**.

2. Defina uma classe genérica para modelar as contas do banco.

```
1 class Conta {
2     private double saldo;
3
4     // GETTERS AND SETTERS
5 }
```

3. Defina duas classes específicas para dois tipos de contas do banco: poupança e corrente.

```
1 class ContaPoupanca extends Conta {
2     private int diaDoAniversario;
3
4     // GETTERS AND SETTERS
5 }
```

```
1 class ContaCorrente extends Conta {
2     private double limite;
3
4     // GETTERS AND SETTERS
5 }
```

4. Defina uma classe para especificar um gerador de extratos.

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3
4  class GeradorDeExtrato {
5
6      public void imprimeExtratoBasico(Conta c) {
7          SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
8          Date agora = new Date();
9
10         System.out.println("DATA: "  + sdf.format(agora));
11         System.out.println("SALDO: "  + c.getSaldo());
12     }
13 }
```

Não se preocupe com os comando de "import", discutiremos sobre ele posteriormente.

5. Faça um teste para o gerador de extratos.

```
1  class TestaGeradorDeExtrato {
2
3      public static void main(String[] args) {
4          GeradorDeExtrato gerador = new GeradorDeExtrato();
5
6          ContaPoupanca cp = new ContaPoupanca();
7          cp.setSaldo(1000);
8
9          ContaCorrente cc = new ContaCorrente();
10         cc.setSaldo(1000);
11
12         gerador.imprimeExtratoBasico(cp);
13         gerador.imprimeExtratoBasico(cc);
14     }
15 }
```

Capítulo 10

Classes Abstratas

10.1 Classes Abstratas

Suponha que no banco todas as contas possuem um tipo específico, por exemplo, conta poupança, conta corrente ou conta salário e que elas são modelas dentro do nosso sistema pelas seguintes classes.

```
1 class Conta {
2     private double saldo;
3
4     // MAIS ATRIBUTOS E MÉTODOS
5 }
```

```
1 class ContaPoupanca extends Conta {
2     private int diaDoAniversario;
3
4     // MAIS ATRIBUTOS E MÉTODOS
5 }
```

```
1 class ContaCorrente extends Conta {
2     private double limite;
3
4     // MAIS ATRIBUTOS E MÉTODOS
5 }
```

Para cada conta do domínio do banco devemos criar um objeto da classe correspondente ao tipo da conta. Por exemplo, se existe uma conta poupança no domínio do banco devemos criar um objeto da classe CONTAPOUPANCA.

```
1 ContaPoupanca cp = new ContaPoupanca();
```

Faz sentido criar objetos da classe CONTAPOUPANCA pois existem contas poupança no domínio do banco. Dizemos que a classe CONTAPOUPANCA é uma classe concreta pois criaremos objetos a partir dela.

Por outro lado, a classe CONTA não define uma conta que de fato existe no domínio do banco. Ela apenas serve como "base"para definir as contas concretos.

Não faz sentido criar um objeto da classe CONTA pois estaríamos instanciado um objeto que não é suficiente para representar uma conta que pertença ao domínio do banco. Mas, a princípio não há nada proibindo a criação de objetos dessa classe. Para adicionar essa restrição no sistema, devemos declarar a classe CONTA como abstrata.

Uma classe concreta pode ser utilizada para instanciar objetos. Por outro lado, uma classe abstrata não pode. Todo código que tenta criar um objeto de uma classe abstrata não compila.

Para definir uma classe abstrata, basta adicionar o modificador **abstract**.

```
1 abstract class Conta {
2 }
```

10.2 Métodos Abstratos

Suponha que o banco ofereça extrato detalhado das contas e para cada tipo de conta as informações e o formato desse extrato detalhado são diferentes. Além disso, a qualquer momento o banco pode mudar os dados e o formato do extrato detalhado de um dos tipos de conta.

Neste caso, não vale apena ter uma implementação de método na classe CONTA para gerar extratos detalhados porque essa implementação seria reescrita nas classes específicas sem nem ser reaproveitada.

Poderíamos, simplesmente, não definir nenhum método para gerar extratos detalhados na classe CONTA. Porém, nada garantiria que as classes que derivam direta ou indiretamente da classe CONTA possuam uma implementação para gerar extratos detalhados nem que uma mesma assinatura de método seja respeitada, ou seja, as classes derivadas poderiam definir métodos com nomes, parâmetros ou retorno diferentes entre si.

Para garantir que toda classe concreta que deriva direta ou indiretamente da classe CONTA tenha uma implementação de método para gerar extratos detalhados e além disso que uma mesma assinatura de método seja utilizada, devemos utilizar o conceito de métodos abstratos.

Na classe CONTA, definimos um método abstrato para gerar extratos detalhados. Um método abstrato possui somente assinatura, ou seja, não possui corpo (implementação).

```
1 abstract class Conta {
2     private double saldo;
3
4     // GETTERS AND SETTERS
5
6     public abstract void imprimeExtratoDetalhado();
7 }
```

```
class ContaPoupanca extends Conta {
    private int diaDoAniversario;

    public void imprimeExtratoDetalhado() {
        System.out.println("EXTRATO DETALHADO DE CONTA POUPANÇA");

        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + this.getSaldo());
        System.out.println("ANIVERSÁRIO: "  + this.diaDoAniversario);
    }
}
```

Toda classes concreta que deriva direta ou indiretamente da classe CONTA será obrigada a ter uma implementação desse método. Caso contrário a classe não compila.

```
// ESSA CLASSE NÃO COMPILA
class ContaPoupanca extends Conta {
    private int diaDoAniversario;

}
```

10.3 Exercícios

1. Crie um projeto no eclipse chamado **Classes-Abstratas**.

2. Defina uma classe genérica para modelar as contas do banco.

```
class Conta {
    private double saldo;

    // GETTERS AND SETTERS
}
```

3. Crie um teste simples para utilizar objetos da classe CONTA.

```
class TestaConta {
    public static void main(String[] args) {
        Conta c = new Conta();

        c.setSaldo(1000);

        System.out.println(c.getSaldo());
    }
}
```

4. Torne a classe CONTA abstrata e verifique o que acontece na classe de teste.

5. Defina uma classe para modelar as contas poupança do nosso banco.

```
class ContaPoupanca extends Conta {
    private int diaDoAniversario;

    // GETTERS E SETTERS
}
```

6. Altere a classe TESTACONTA para corrigir o erro de compilação.

```
class TestaConta {
    public static void main(String[] args) {
        Conta c = new ContaPoupanca();

        c.setSaldo(1000);

        System.out.println(c.getSaldo());
    }
}
```

7. Defina um método abstrato na classe CONTA para gerar extratos detalhados.

```
abstract class Conta {
    private double saldo;

    // GETTERS AND SETTERS

    public abstract void imprimeExtratoDetalhado();
}
```

8. O que acontece com a classe CONTAPOUPANCA?

9. Defina uma implementação do método IMPRIMEEXTRATODETALHADO() na classe CONTAPOUPANCA.

```
import java.text.SimpleDateFormat;
import java.util.Date;

class ContaPoupanca extends Conta {
    private int diaDoAniversario;

    public void imprimeExtratoDetalhado() {
        System.out.println("EXTRATO DETALHADO DE CONTA POUPANÇA");

        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();

        System.out.println("DATA: "  + sdf.format(agora));
        System.out.println("SALDO: "  + this.getSaldo());
        System.out.println("ANIVERSÁRIO: "  + this.diaDoAniversario);
    }
}
```

Capítulo 11

Interfaces

11.1 Padronização

Recentemente, a TV Digital chegou ao Brasil. Muitos debates, testes e pesquisas foram realizados para escolher qual padrão de TV Digital deveria ser adotado pelo Brasil antes dessa nova tecnologia ser implantada.

Quando o brasil iniciou o processo de definição de um padrão para TV Digital, alguns países já haviam definido um padrão próprio. Os principais eram o americano e o japonês.

O Brasil poderia criar o seu próprio padrão ou adotar um dos já existentes. Essa escolha é complexa pois afeta fortemente vários setores do país. As emissoras de TV precisam adquirir novos equipamentos e capacitar pessoas para operá-los. Os fabricantes de aparelhos de TV devem desenvolver tecnologia para se adaptar. A população em geral precisa adquirir novos aparelhos ou adaptadores.

Esse problema, pode ser levado a escala mundial. A escolha de um padrão diferente dos outros países implica em dificuldades técnicas para utilizar produtos importados. Além disso, os produtos exportados pelo Brasil também podem enfrentar problemas técnicos para serem utilizados no exterior. Isso certamente afeta as relações comerciais do Brasil com o resto do mundo.

O Brasil acabou adotando um padrão fortemente baseado no padrão japonês. Esse padrão define o que cada dispositivo que participa da TV Digital **deve fazer**, desde do decodificador das TV's até as antenas que transmitem o sinal digital.

Essa padronização possibilita que aparelhos de TV de diversos fabricantes "conversem"com as antenas das emissoras de TV. Mais, do que isso, ele permite que aparelhos móveis como por exemplo celulares possam interagir de uma maneira uniforme com as antenas.

É claro que isso tudo funciona porque as empresas que fabricam os dispositivos que participam da TV Digital respeitam as especificações do padrão. Além disso, não é interessante para um fabricante de TV não seguir as especificações pois os aparelhos dele não funcionariam e consequentemente ninguém compraria.

Padronização através especificações é algo bem comum. Há organizações nacionais e internacionais que definem padrões para diversas áreas (veja `http://www.iso.org`). Normalmente, essas organizações além de definir especificações elas emitem certificados para os produtos ou serviços que estão de acordo com determinadas especificações. Esses certificados são importantes pois garantem qualidade e modo de utilização dos produtos ou serviços.

Eventualmente, padrões são substituídos por outros. Recentemente, o Brasil trocou o

padrão das tomadas elétricas afetando a maior parte da população. As pessoas tiveram que substituir tomadas ou comprar adaptadores.

11.2 Contratos

Num sistema Orientado a Objetos, os objetos interagem entre si através de chamadas de métodos (troca de mensagens). Assim como os aparelhos de TV e dispositivos móveis interagem com as antenas.

Para os objetos de um sistema "conversarem" entre si facilmente é importante padronizar o conjunto de métodos oferecidos por eles. Assim como as operações das antenas e dos aparelhos de TV são padronizadas.

Um padrão é definido através de especificações ou contratos. No contexto da orientação a objetos, um contrato é uma **interface**. Uma interface é um "contrato" que define assinaturas de métodos que devem ser implementados pelas classes que "assinarem" o mesmo.

11.3 Exemplo

No sistema do banco, podemos definir uma interface (contrato) para padronizar as assinaturas dos métodos oferecidos pelos objetos que representam as contas do banco.

```
1 interface Conta {
2     void deposita(double valor);
3     void saca(double valor);
4 }
```

Observe que somente assinaturas de métodos são declaradas no corpo de uma interface. Todos os métodos de uma interface são públicos e abstratos. Os modificadores PUBLIC e ABSTRACT são opcionais.

As classes que definem os diversos tipos de contas que existem no banco devem implementar (assinar) a interface CONTA.

```
1 class ContaPoupanca implements Conta {
2     public void deposita(double valor) {
3         // implementacao
4     }
5     public void saca(double valor) {
6         // implementacao
7     }
8 }
```

```
1 class ContaCorrente implements Conta {
2     public void deposita(double valor) {
3         // implementacao
4     }
5     public void saca(double valor) {
6         // implementacao
7     }
8 }
```

As classes concretas que implementam uma interface são obrigadas a possuir uma implementação para cada método declarado na interface. Caso contrário, ocorrerá um erro de compilação.

```
1 // Esta classe não compila porque ela não implementou o método saca()
2 class ContaCorrente implements Conta {
3     public void deposita(double valor) {
4         // implementacao
5     }
6 }
```

A primeira vantagem de utilizar uma interface é a padronização das assinaturas dos métodos oferecidos por um determinado conjunto de classes. A segunda vantagem é garantir que determinadas classes implementem certos métodos.

11.4 Polimorfismo

Se uma classe implementa uma interface, podemos aplicar a ideia do polimorfismo assim como em herança. Dessa forma, outra vantagem da utilização de interfaces é o ganho do polimorfismo.

Como exemplo, suponha que a classe CONTACORRENTE implemente a interface CONTA. Podemos guardar a referência de um objeto do tipo CONTACORRENTE em uma variável do tipo CONTA.

```
1 Conta c = new ContaCorrente();
```

Além disso podemos passar uma variável do tipo CONTACORRENTE para um método que o parâmetro seja do tipo CONTA.

```
1 GeradorDeExtrato g = new GeradorDeExtrato();
2 ContaCorrente c = new ContaCorrente();
3 g.geraExtrato(c);
```

```
1 class GeradorDeExtrato {
2     public void geraExtrato(Conta c) {
3         // implementação
4     }
5 }
```

O método GERAEXTRATO() pode ser aproveitado para objetos criados a partir de classes que implementam direta ou indiretamente a interface CONTA.

11.5 Interface e Herança

As vantagens e desvantagens entre interface e herança, provavelmente, é o tema mais discutido nos blogs, fóruns e revistas que abordam desenvolvimento de software orientado a objetos.

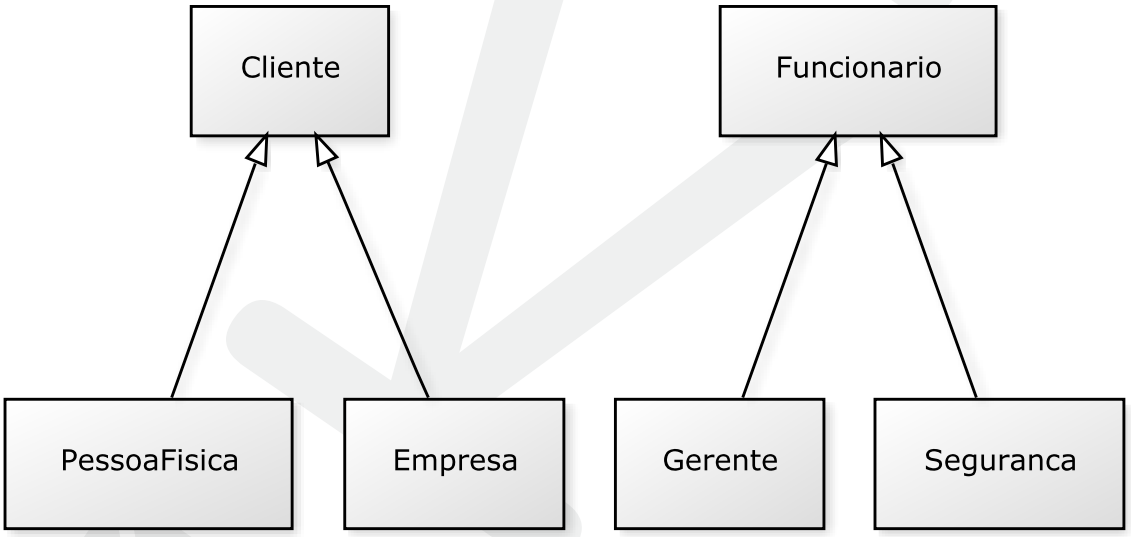
Muitas vezes, os debates sobre este assunto se estendem mais do que a própria importância deste tema. Inclusive, muitas pessoas se posicionam de forma radical defendendo a utilização de interfaces ao invés de herança em qualquer situação. Normalmente, os debates sobre este tópico estão interessados em analisar o que é melhor para manutenção, utilizar interfaces ou aplicar herança.

A grosso modo, priorizar a utilização de interfaces permite que alterações pontuais em determinados trechos do código fonte sejam feitas mais facilmente pois diminui as ocorrências de efeitos colaterais indesejados no resto da aplicação. Por outro lado, priorizar a utilização de herança pode diminuir a quantidade de código escrito no início do desenvolvimento de um projeto.

Algumas pessoas propõem a utilização de interfaces juntamente com composição para substituir totalmente o uso de herança. De fato, esta é uma alternativa interessante pois possibilita que um trecho do código fonte de uma aplicação possa ser alterado sem causar efeito colateral no restante do sistema além de permitir a reutilização de código mais facilmente.

Do ponto de vista prático, em Java, como não há herança múltipla, muitas vezes, interfaces são apresentadas como uma alternativa para obter um grau maior de polimorfismo.

Por exemplo, suponha duas árvores de herança independentes.



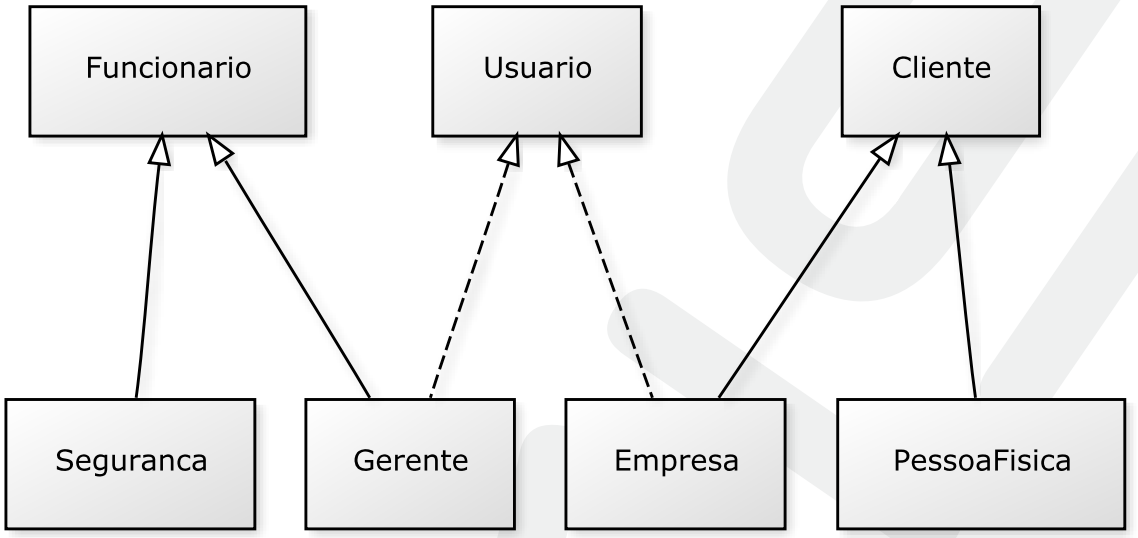
Suponha que os gerentes e as empresas possam acessar o sistema do banco através de um nome de usuário e uma senha. Seria interessante utilizar um único método para implementar a autenticação desses dois tipos de objetos. Mas, qual seria o tipo de parâmetro deste método? Lembrando que ele deve aceitar gerentes e empresas.

```
1 class AutenticadorDeUsuario {
2     public boolean autentica(??? u) {
3         // implementação
4     }
5 }
```

De acordo com as árvores de herança, não há polimorfismo entre objetos da classe GERENTE e da classe EMPRESA. Para obter polimorfismo entre os objetos dessas duas classes somente com herança, deveríamos colocá-las na mesma árvore de herança. Mas, isso não faz

sentido pois uma empresa não é um funcionário e o gerente não é cliente.

Neste caso, a solução é utilizar interfaces para obter polimorfismo entre objetos da classe GERENTE e da classe EMPRESA.



Agora, conseguimos definir o que o método AUTENTICA() deve receber como parâmetro para trabalhar tanto com gerentes quanto com empresas. Ele deve receber um parâmetro do tipo USUARIO.

```
1 class AutenticadorDeUsuario {
2     public boolean autentica(Usuario u) {
3         // implementação
4     }
5 }
```

11.6 Exercícios

1. Crie um projeto no eclipse chamado **Interfaces**.

2. Defina uma interface para padronizar as assinaturas dos métodos das contas do banco.

```
1 interface Conta {
2     void deposita(double valor);
3     void saca(double valor);
4     double getSaldo();
5 }
```

3. Agora, crie algumas classes para modelar tipos diferentes de conta.

```
class ContaCorrente implements Conta {
    private double saldo;
    private double taxaPorOperacao = 0.45;

    public void deposita(double valor) {
        this.saldo += valor - this.taxaPorOperacao;
    }

    public void saca(double valor) {
        this.saldo -= valor + this.taxaPorOperacao;
    }

    public double getSaldo() {
        return this.saldo;
    }
}
```

```
class ContaPoupanca implements Conta {
    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        this.saldo -= valor;
    }

    public double getSaldo() {
        return this.saldo;
    }
}
```

4. Faço um teste simples com as classes criadas anteriormente.

```
class TestaContas {
    public static void main(String[] args) {
        ContaCorrente c1 = new ContaCorrente();
        ContaPoupanca c2 = new ContaPoupanca();

        c1.deposita(500);
        c2.deposita(500);

        c1.saca(100);
        c2.saca(100);

        System.out.println(c1.getSaldo());
        System.out.println(c2.getSaldo());
    }
}
```

5. Altere a assinatura do método DEPOSITA() na classe CONTACORRENTE. Você pode acrescentar um "r" no nome do método. O que acontece? Obs: desfaça a alteração depois deste exercício.

6. Crie um gerador de extratos com um método que pode trabalhar com todos os tipos de conta.

```
1  class GeradorDeExtrato {
2      public void geraExtrato(Conta c) {
3          System.out.println("EXTRATO");
4          System.out.println("SALDO: " + c.getSaldo());
5      }
6  }
```

7. Teste o gerador de extrato.

```
1  class TestaGeradorDeExtrato {
2      public static void main(String[] args) {
3          ContaCorrente c1 = new ContaCorrente();
4          ContaPoupanca c2 = new ContaPoupanca();
5
6          c1.deposita(500);
7          c2.deposita(500);
8
9          GeradorDeExtrato g = new GeradorDeExtrato();
10         g.geraExtrato(c1);
11         g.geraExtrato(c2);
12     }
13 }
```

Capítulo 12

Pacotes

12.1 Organização

O código fonte de uma aplicação real é formado por uma quantidade grande de arquivos. Conforme essa quantidade cresce surge a necessidade de algum tipo de organização para poder encontrar os arquivos mais rapidamente quando for necessário modificá-los.

A ideia para organizar logicamente os arquivos de uma aplicação é bem simples e muito parecida com o que um usuário de computador já está acostumado a fazer. Os arquivos são separados em pastas e subpastas ou diretórios e subdiretórios.

12.2 O comando package

Na terminologia do Java, dizemos que as classes e interfaces são organizadas em pacotes ("pastas"). O primeiro passo, para colocar uma classe ou interface em um determinado pacote, é utilizar o comando PACKAGE no código fonte.

```
1 // Arquivo: Conta.java
2 package sistema;
3
4 class Conta {
5     // corpo da classe
6 }
```

O segundo passo é salvar o arquivo dentro de uma pasta como mesmo nome do pacote definido no código fonte.

`sistema/Conta.java`

A declaração das classes ou interfaces deve aparecer após a declaração de pacote caso contrário ocorrerá um erro de compilação.

12.3 Sub Pacotes

Podemos criar pacotes dentro de pacotes. No código fonte os sub pacotes são definidos com o operador **.**.

```
1 // Arquivo: Conta.java
2 package sistema.contas;
3
4 class Conta {
5     // corpo da classe
6 }
```

Além disso, devemos criar uma estrutura de pastas que reflita os sub pacotes.

`sistema/contas/Conta.java`

12.4 Classes ou Interfaces públicas

12.4.1 Fully Qualified Name

Com a utilização de pacotes é apropriado definir o que é o nome simples e o nome completo (fully qualified name) de uma classe ou interface. O nome simples é o identificador declarado a direita do comando CLASS ou INTERFACE. O nome completo é formado pela concatenação dos nomes dos pacotes com o nome simples através do carácter ".".

Por exemplo, suponha a seguinte código:

```
1 // Arquivo: Conta.java
2 package sistema.contas;
3
4 class Conta {
5     // corpo da classe
6 }
```

O nome simples da classe acima é: CONTA e o nome completo é: SISTEMA.CONTAS.CONTA.

Duas classes de um mesmo pacote podem "conversar" entre si através do nome simples de cada uma delas. O mesmo vale para interfaces. Por exemplo suponha duas classes:

`sistema/contas/Conta.java`

```
1 // Arquivo: Conta.java
2 package sistema.contas;
3
4 class Conta {
5     // corpo da classe
6 }
```

`sistema/contas/ContaPoupanca.java`

```
1 // Arquivo: ContaPoupanca.java
2 package sistema.contas;
3
4 class ContaPoupanca extends Conta {
5     // corpo da classe
6 }
```

A classe CONTAPOUPANCA declara que herda da classe CONTA apenas utilizando o nome simples.

Por outro lado, duas classes de pacotes diferentes precisam utilizar o nome completo de cada uma delas para "conversar" entre si. Além disso, a classe que será utilizada por classes de outro pacote deve ser pública. O mesmo vale para interfaces. Como exemplo suponha duas classes:

`sistema/contas/Conta.java`

```
1 // Arquivo: Conta.java
2 package sistema.contas;
3
4 public class Conta {
5     // corpo da classe
6 }
```

`sistema/clientes/Cliente.java`

```
1 // Arquivo: Cliente.java
2 package sistema.clientes;
3
4 class Cliente {
5     private sistema.contas.Conta conta;
6 }
```

12.5 Import

Para facilitar a escrita do código fonte, podemos utilizar o comando IMPORT para não ter que repetir o nome completo de uma classe ou interface várias vezes dentro do mesmo arquivo.

`sistema/clientes/Cliente.java`

```
1 // Arquivo: Cliente.java
2 package sistema.clientes;
3
4 import sistema.contas.Conta;
5
6 class Cliente {
7     private Conta conta;
8 }
```

Podemos importar várias classes ou interfaces no mesmo arquivo. As declarações de importe devem aparecer após a declaração de pacote e antes das declarações de classes ou interfaces.

12.6 Níveis de visibilidade

No Java, há quatro níveis de visibilidade: privado, padrão, protegido e público. Podemos definir os níveis privado, protegido e público com os modificadores PRIVATE, PROTECTED e PUBLIC respectivamente. Quando nenhum modificador de visibilidade é utilizado o nível padrão é aplicado.

12.6.1 Privado

O nível privado é aplicado com o modificador PRIVATE.

O que pode ser privado? Atributos, construtores, métodos, classes aninhadas ou interfaces aninhadas.

Os itens em nível de visibilidade privado só podem ser acessados por código escrito na mesma classe na qual eles foram declarados.

12.6.2 Padrão

O nível padrão é aplicado quando nenhum modificador é utilizado.

O que pode ser padrão? Atributos, construtores, métodos, classes de todos os tipos e interfaces de todos os tipos.

Os itens em nível de visibilidade padrão só podem ser acessados por código escrito em classes do mesmo pacote da classe na qual eles foram declarados.

12.6.3 Protegido

O nível protegido é aplicado com o modificador PROTECTED.

O que pode ser protegido? Atributos, construtores, métodos, classes aninhadas ou interfaces aninhadas.

Os itens em nível de visibilidade protegido só podem ser acessados por código escrito em classes do mesmo pacote da classe na qual eles foram declarados ou por classes derivadas.

12.6.4 Público

O nível público é aplicado quando o modificador PUBLIC é utilizado.

O que pode ser público? Atributos, construtores, métodos, classes de todos os tipos e interfaces de todos os tipos.

Os itens em nível de visibilidade público podem ser acessados de qualquer lugar do código da aplicação.

12.7 Exercícios

1. Crie um projeto no eclipse chamado **Pacotes**.

2. Crie um pacote chamado **sistema** e outro chamado **testes**.

3. Faça uma classe para modelar as contas no pacote **sistema**.

```
1  // Arquivo: Conta.java
2  package sistema;
3
4  public class Conta {
5      private double saldo;
6
7      public void deposita(double valor) {
8          this.saldo += valor;
9      }
10 }
```

4. Faça uma classe de teste no pacote **testes**.

```
1  // Arquivo: Teste.java
2  package testes;
3
4  import sistema.Conta;
5
6  public class Teste {
7      public static void main(String[] args) {
8          Conta c = new Conta();
9          c.deposita(1000);
10         System.out.println(c.getSaldo());
11     }
12 }
```

5. Retire o modificador PUBLIC da classe CONTA e observe o erro de compilação na classe TESTE. Importante: faça a classe CONTA ser pública novamente.

Capítulo 13

Exceptions

Como erros podem ocorrer durante a execução de uma aplicação, devemos definir como eles serão tratados. Tradicionalmente, códigos de erro são utilizados para lidar com falhas na execução de um programa. Nesta abordagem, os métodos devolveriam números inteiros para indicar o tipo de erro que ocorreu.

```
1 int deposita(double valor) {
2     if(valor < 0) {
3         return 107; // código de erro para valor negativo
4     } else {
5         this.saldo += valor;
6     }
7 }
```

Utilizar códigos de erro exige uma vasta documentação dos métodos para explicar o que cada código significa. Além disso, esta abordagem "gasta" o retorno do método impossibilitando que outros tipos de dados sejam devolvidos. Em outras palavras, ou utilizamos o retorno para devolver códigos de erro ou para devolver algo pertinente a lógica natural do método. Não é possível fazer as duas coisas sem nenhum tipo de gambiarra.

```
1 ??? geraRelatorio() {
2     if(...) {
3         return 200; // código de erro tipo1
4     } else {
5         Relatorio relatorio = ...
6         return relatorio;
7     }
8 }
```

Observe que no código do método GERARELATORIO() seria necessário devolver dois tipos de dados incompatíveis: int e referências de objetos da classe RELATORIO. Porém, não é possível definir dois tipos incopatíveis como retorno de um método.

A linguagem Java tem uma abordagem própria para lidar com erros de execução. Na abordagem do Java não são utilizados códigos de erro ou os retornos dos métodos.

13.1 Tipos de erros de execução

O primeiro passo para entender a abordagem do Java para lidar com os erros de execução é saber classificá-los. A classe THROWABLE modela todos os tipos de erros de execução. Há duas subclasses de THROWABLE: ERROR e EXCEPTION. A subclasse ERROR define erros que não devem ser capturados pelas aplicações pois representam erros graves que não permitem que a execução continue de maneira satisfatória. A subclasse EXCEPTION define erros para os quais as aplicações normalmente têm condições de definir um tratamento.

13.2 Lançando erros

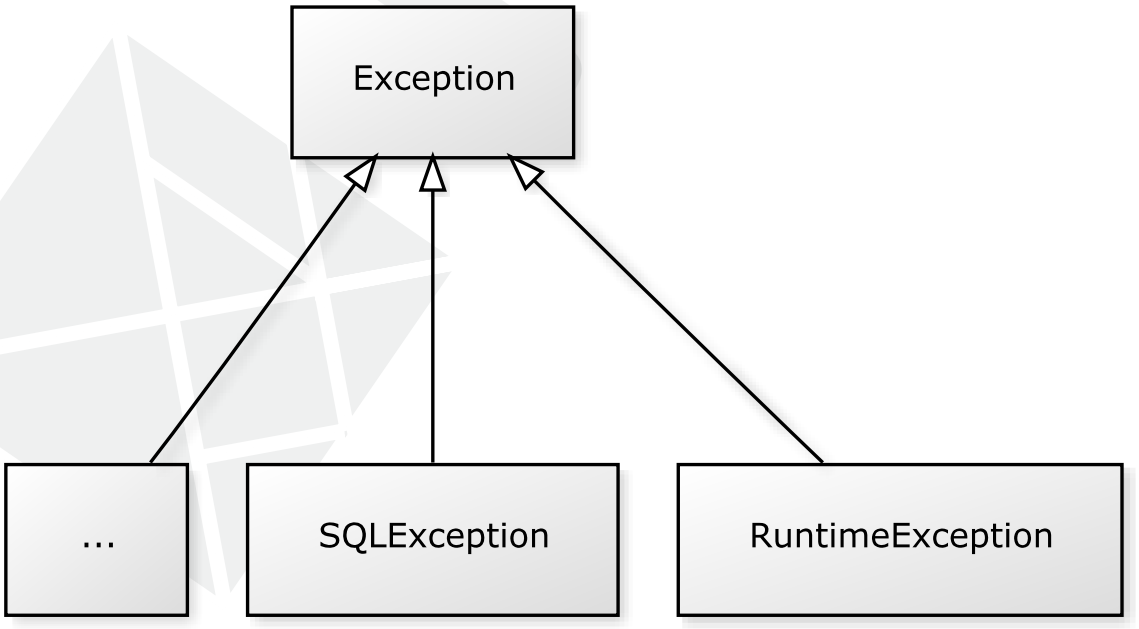
Tecnicamente, podemos criar um objeto de qualquer classe que deriva de THROWABLE para representar um erro que foi identificado. Porém, conceitualmente, devemos utilizar apenas as classes que derivam de EXCEPTION.

Depois de criar uma exception podemos "lançar" a referência dela utilizando o comando THROW. Observe o exemplo utilizando a classe ILLEGALARGUMENTEXCEPTION que deriva indiretamente da classe EXCEPTION.

```
1 if(valor < 0) {
2     IllegalArgumentException erro = new IllegalArgumentException();
3     throw erro;
4 }
```

13.2.1 Checked e Unchecked

As exceptions são classificadas em **checked** e **unchecked**. Para identificar o tipo de uma exception, devemos considerar a árvore de herança da classe EXCEPTION.



As classes que derivam de EXCEPTION mas não derivam de RUNTIMEEXCEPTION definem as chamadas **checked-exceptions** já as classes que derivam de RUNTIMEEXCEPTION definem as chamadas **unchecked-exceptions**.

Para um método lançar uma unchecked-exception, basta utilizar o comando THROW como visto anteriormente. Para um método lançar uma checked-exception, além do comando THROW é necessário utilizar o comando THROWS. Observe o exemplo utilizando a classe ILLEGALARGUMENTEXCEPTION que deriva diretamente da classe RUNTIMEEXCEPTION e a classe SQLEXCEPTION que deriva diretamente da classe EXCEPTION mas não da RUNTIMEEXCEPTION.

```
1  // lançando uma unchecked-exception
2  void deposita(double valor) {
3      if(valor < 0) {
4          IllegalArgumentException erro = new IllegalArgumentException();
5          throw erro;
6      }
7  }
```

```
1  // lançando uma unchecked-exception
2  void deposita(double valor) throws SQLException {
3      if(valor < 0) {
4          SQLException erro = new SQLException();
5          throw erro;
6      }
7  }
```

13.3 Capturando erros

Em um determinado trecho de código, para capturar uma exception devemos utilizar o comando TRY-CACTH.

```
1  class Teste {
2      public static void main(String[] args) {
3          Conta c = new Conta();
4
5          try {
6              c.deposita();
7          } catch (IllegalArgumentException e) {
8              System.out.println("Houve um erro ao depositar");
9          }
10     }
11 }
```

Podemos encadear vários blocos CATCH para capturar exceptions de classes diferentes.

```
class Teste {
    public static void main(String[] args) {
        Conta c = new Conta();

        try {
            c.deposita();
        } catch (IllegalArgumentException e) {
            System.out.println("Houve uma IllegalArgumentException ao depositar");
        } catch (SQLException e) {
            System.out.println("Houve uma SQLException ao depositar");
        }
    }
}
```

13.4 Exercícios

1. Crie um projeto no eclipse chamado **Exceptions**.

2. Crie uma classe para modelar os funcionários do sistema do banco.

```
class Funcionario {
    private double salario;

    public void aumentaSalario(double aumento) {
        if(aumento < 0) {
            IllegalArgumentException erro = new IllegalArgumentException();
            throw erro;
        }
    }
}
```

3. Agora teste a classe FUNCIONARIO.

```
class TestaFuncionario {
    public static void main(String[] args) {
        Funcionairo f = new Funcionario();
        f.aumentaSalario(-1000);
    }
}
```

   Execute e observe o erro no console.

4. Altere o teste para capturar o erro.

```
1  class TestaFuncionario {
2      public static void main(String[] args) {
3          Funcionairo f = new Funcionario();
4
5          try {
6              f.aumentaSalario(-1000);
7          } catch(IllegalArgumentException e) {
8              System.out.println("Houve uma IllegalArgumentException ao aumentar o ↩
                   salário");
9          }
10     }
11 }
```

Capítulo 14

Object

Todas as classes derivam direta ou indiretamente da classe JAVA.LANG.OBJECT. Consequentemente, todo conteúdo definido nessa classe estará presente em todos os objetos.

Além disso, qualquer referência pode ser armazenada em uma variável do tipo OBJECT. A ideia do polimorfismo pode ser aplicada para criar métodos genéricos que trabalham com objetos de qualquer classe.

14.1 Polimorfismo

Aproveitando o polimorfismo gerado pela herança da classe OBJECT, é possível criar uma classe para armazenar objetos de qualquer tipo como se fosse um repositório de objetos.

```
1 class RepositorioDeObjetos {
2     // codigo da classe
3 }
```

Um array de objetos pode ser utilizado como estrutura básica para manter os objetos do repositório.

```
1 Object[] objetos = new Object[100];
```

Alguns métodos podem ser criados para formar a interface do repositório. Por exemplo, métodos para adicionar, retirar e pesquisar elementos.

```
1 public void adiciona(Object o) {
2     // implementacao
3 }
```

```
1 public void remove(Object o) {
2     // implementacao
3 }
```

```
1 public Object pega(int posicao) {
2     // implementacao
3 }
```

Com esses métodos o repositório teria a vantagem de armazenar objetos de qualquer tipo. Porém, na compilação, não haveria garantia sobre os tipos específicos. Em outras palavras, já que objetos de qualquer tipo podem ser armazenados no repositório então objetos de qualquer tipo podem sair dele.

```
1 Repositorio repositorio = new Repositorio();
2 repositorio.adiciona("Rafael");
3 Object o = repositorio.pega(0);
```

Por outro lado, na maioria dos casos, os programadores criam repositórios para armazenar objetos de um determinado tipo. Por exemplo, uma repositório para armazenar somente nomes de pessoas, ou seja, para armazenar objetos do tipo JAVA.LANG.STRING. Nesse caso, em tempo de compilação é possível "forçar" o compilador a tratar os objetos como string.

```
1 Repositorio repositorio = new Repositorio();
2 repositorio.adiciona("Rafael");
3 Object o = repositorio.pega(0);
4 String s = (String)o;
```

14.2 O método toString()

Quando uma variável de um tipo primitivo é passada como parâmetro para o método PRINTLN() o valor primitivo é impresso na tela (saída padrão).

```
1 int a = 10;
2 System.out.println(a);
```

O que será impresso quando uma variável não primitiva é passada como argumento para o método PRINTLN()?

O método PRINTLN() imprime na tela uma string que é obtida através do método TOSTRING(). Todos os objetos possuem esse método pois todas as classes derivam direta ou indiretamente da classe OBJECT que define o TOSTRING().

A implementação padrão do TOSTRING(), na classe OBJECT, devolve o nome da classe mais específica do objeto.

```
1 class Vetor {
2     private double x;
3     private double y;
4 
5     public Vetor(double x, double y) {
6         this.x = x;
7         this.y = y;
8     }
9 }
```

```
1 Vetor vetor = new Vetor(10,5);
2
3 // imprime na tela: Vetor
4 System.out.println(vetor);
```

O método TOSTRING() pode ser reescrito para devolver uma string que faça mais sentido para o tipo de objeto correspondente a classe na qual a reescrita ocorrerá.

interface

```
1 class Vetor {
2     // ATRIBUTOS
3     // CONSTRUTOR
4
5     public String toString() {
6         return "(" + this.x + ", " + this.y + ")";
7     }
8 }
```

```
1 Vetor vetor = new Vetor(10, 5);
2
3 // imprime na tela: (10.0, 5.0)
4 System.out.println(vetor);
```

14.3 O método equals()

Para verificar se os valores armazenados em duas variáveis de algum tipo primitivo são iguais, deve ser utilizado o operador "==" . Esse operador também pode ser aplicado em variáveis de tipos não primitivos.

```
1 Vetor vetor1 = new Vetor(10, 5);
2 Vetor vetor2 = new Vetor(10, 5);
3
4 // imprime na tela: False
5 System.out.println(vetor1 == vetor2);
```

O operador "==", aplicado em referências, verifica se elas apontam para o mesmo objeto na memória. Quando for necessário comparar o conteúdo de dois objetos e não as suas localizações na memória, o método EQUALS() deve ser utilizado. Esse método está presente em todos os objetos pois ele foi definido na classe OBJECT.

A implementação padrão do método EQUALS() na classe OBJECT verifica se os objetos estão localizados no mesmo lugar da memória. Esse método deve ser reescrito caso seja necessário comparar o conteúdo dos objetos.

```
class Vetor {
    // ATRIBUTOS
    // CONSTRUTOR

    public boolean equals(Object o){
        Vetor outro = (Vetor)o;
        return this.x == outro.x && this.y == outro.y;
    }
}
```

O método EQUALS() recebe como parâmetro uma variável do tipo OBJECT. Dessa forma, é necessário aplicar a ideia de casting de referência para poder acessar o conteúdo do objeto definido na classe VETOR.

A implementação do método EQUALS() não está correta pois ela gera um erro de execução quando o parâmetro não puder ser convertido para VETOR. Para garantir que não ocorra um erro de execução o tipo do objeto pode ser testado antes do casting.

```
public boolean equals(Object o){
    if (o instaceof Vetor) {
        Vetor outro = (Vetor)o;
        return this.x == outro.x && this.y == outro.y;
    }
    else {
        return false;
    }
}
```

O uso do método EQUALS() é análogo ao de outro método qualquer.

```
Vetor vetor1 = new Vetor(10, 5);
Vetor vetor2 = new Vetor(10, 5);

// imprime na tela: True
System.out.println(vetor1.equals(vetor2));
```

14.4 Exercícios

1. Crie uma classe *Lista* capaz de armazenar objetos de qualquer tipo.
2. Crie uma classe *Vetor* análoga à vista nesse capítulo.
3. Reescreva o método *ToString* na classe *Vetor*.
4. Reescreva o método *Equals* na classe *Vetor*.

Capítulo 15

Entrada e Saída

No contexto do desenvolvimento de software, quando falamos em entrada e saída, estamos nos referindo a qualquer troca de informação entre uma aplicação e o seu exterior.

A leitura do que o usuário digita no teclado, do conteúdo de um arquivo ou dos dados recebidos pela rede são exemplos de entrada de dados. A impressão de mensagens no console, escrita de texto em um arquivo ou envio de dados pela rede são exemplos de saída de dados.

A plataforma Java oferece diversas classes e interfaces para facilitar o processo de entrada e saída.

15.1 Byte a Byte

Para determinadas situações e problemas, é necessário que uma aplicação faça entrada e saída byte a byte. As classes da plataforma Java responsáveis pela leitura e escrita byte a byte são INPUTSTREAM e OUTPUTSTREAM respectivamente. Essas duas classes estão no pacote **java.io**.

Veja um exemplo de leitura do teclado:

```
1  InputStream entrada = System.in;
2
3  int i;
4  do {
5      i = entrada.read();
6      System.out.println("valor lido: " + i);
7
8  } while (i != -1);
```

O fluxo de entrada associado ao teclado é representado pelo objeto referenciado pelo atributo estático SYSTEM.IN. O método READ() faz a leitura do próximo byte da entrada.

Veja um exemplo de escrita no console:

```
1  OutputStream saida = System.out;
2
3  saida.write(107);
4  saida.write(49);
5  saida.write(57);
6  saida.flush();
```

O fluxo de saída associado ao console é representado pelo objeto referenciado pelo atributo estático SYSTEM.OUT. O método WRITE() armazena um byte(um valor entre 0 e 255) no buffer da saída. O método FLUSH() libera o conteúdo do buffer para a saída.

A classe INPUTSTREAM é genérica e modela um fluxo de entrada sem uma fonte específica definida. Diversas classes herdam direta ou indiretamente da classe INPUTSTREAM para especificar um determinado tipo de fonte de dados.

Eis algumas classes que derivam da classe INPUTSTREAM:

- AudioInputStream
- FileInputStream
- ObjectInputStream

A classe OUTPUTSTREAM é genérica e modela um fluxo de saída sem um destino específico definido. Diversas classes herdam direta ou indiretamente da classe OUTPUTSTREAM para especificar um determinado tipo de destino de dados.

Eis algumas classes que derivam da classe OUTPUTSTREAM:

- ByteArrayOutputStream
- FileOutputStream
- ObjectOutputStream

15.2 Scanner

Nem sempre é necessário fazer entrada byte a byte. Nestes casos, normalmente, é mais simples utilizar a classe SCANNER do pacote **java.util** do Java. Esta classe possui métodos mais sofisticados para obter os dados de uma entrada.

Veja um exemplo de leitura do teclado com a classe SCANNER:

```
1 InputStream entrada = System.in;
2 Scanner scanner = new Scanner(entrada);
3
4 while(scanner.hasNextLine()) {
5     String linha = scanner.nextLine();
6     System.out.println(linha);
7 }
```

Os objetos da classe SCANNER podem ser associados a diversas fontes de dados.

```
1 InputStream teclado = System.in;
2 Scanner scannerTeclado = new Scanner(teclado);
3
4 FileInputStream arquivo = new FileInputStream("arquivo.txt");
5 Scanner scannerArquivo = new Scanner(arquivo);
```

15.3 PrintStream

Nem sempre é necessário fazer saída byte a byte. Nestes casos, normalmente, é mais simples utilizar a classe PRINTSTREAM do pacote **java.io** do Java. Esta classe possui métodos mais sofisticados para enviar dados para uma entrada.

Veja um exemplo de escrita no console com a classe PRINTSTREAM:

```
1 OutputStream console = System.out;
2 PrintStream printStream = new PrintStream(console);
3
4 printStream.println("Curso de Java e Orientação da K19");
```

Os objetos da classe PRINTSTREAM podem ser associados a diversos destinos de dados.

```
1 OutputStream console = System.out;
2 PrintStream printStreamConsole = new PrintStream(console);
3
4 FileOutputStream arquivo = new FileOutputStream("arquivo.txt");
5 PrintStream printStreamArquivo = new PrintStream(arquivo);
```

15.4 Exercícios

1. Crie um projeto no eclipse chamado **EntradaSaida**.

2. Crie um teste para recuperar e imprimir na tela o conteúdo digitado pelo usuário no teclado.

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.util.Scanner;
4
5  public class LeituraDoTeclado {
6      public static void main(String[] args) throws IOException {
7          InputStream teclado = System.in;
8          Scanner scanner = new Scanner(teclado);
9
10         while (scanner.hasNextLine()) {
11             String linha = scanner.nextLine();
12             System.out.println(linha);
13         }
14     }
15 }
```

   OBS: Para finalizar o fluxo de entrado do teclado digite CTRL+D no Linux ou CTRL+Z no Windows.

3. Crie um teste para recuperar e imprimir na tela o conteúdo de um arquivo.

```
1  import java.io.FileInputStream;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.util.Scanner;
5
6  public class LeituraDeArquivo {
7      public static void main(String[] args) throws IOException {
8          InputStream teclado = new FileInputStream("entrada.txt");
9          Scanner scanner = new Scanner(teclado);
10
11         while (scanner.hasNextLine()) {
12             String linha = scanner.nextLine();
13             System.out.println(linha);
14         }
15     }
16 }
```

OBS: O arquivo “entrada.txt” deve ser criado no diretório **raiz** do projeto **EntradaSaida**.

4. Crie um teste para imprimir algumas linhas em um arquivo.

```
1  import java.io.FileOutputStream;
2  import java.io.IOException;
3  import java.io.PrintStream;
4
5  public class EscritaDeArquivo {
6      public static void main(String[] args) throws IOException {
7          FileOutputStream arquivo = new FileOutputStream("saida.txt");
8          PrintStream printStream = new PrintStream(arquivo);
9
10         printStream.println("Primeira linha!!!");
11         printStream.println("Segunda linha!!!");
12         printStream.println("Terceira linha!!!");
13     }
14 }
```

OBS: O projeto **EntradaSaida** deve ser atualizado para que o arquivo “saida.txt” seja mostrado no eclipse.

5. (Opcional) Crie um teste que faça a leitura do conteúdo de um arquivo e grave em outro arquivo.

6. (Opcional) Crie um teste que faça a leitura do teclado e grave em arquivo.

Capítulo 16

Collections

Quando uma aplicação precisa manipular uma quantidade grande de dados, ela deve utilizar alguma estrutura de dados. Podemos dizer que a estrutura de dados mais básica do Java são os arrays.

Muitas vezes, trabalhar diretamente com arrays não é simples dado as diversas limitações que eles possuem. A limitação principal é a capacidade fixa, um array não pode ser redimensionado. Se todas as posições de um array estiverem ocupadas não podemos adicionar mais elementos. Normalmente, criamos um outro array com maior capacidade e transferimos os elementos do array antigo para o novo.

Além disso, adicionar ou remover elementos provavelmente gera a necessidade de deslocar parte do conteúdo do array.

As dificuldades do trabalho com array podem ser superadas com estruturas de dados mais sofisticadas. Na biblioteca do Java, há diversas estruturas de dados que facilitam o trabalho do desenvolvedor.

16.1 Listas

As listas são estruturas de dados de armazenamento sequencial assim como os arrays. Mas, diferentemente dos arrays, as listas não possuem capacidade fixa o que facilita bastante o trabalho.

LIST é a interface Java que define os métodos que uma lista deve implementar. As principais implementações da interface LIST são: ARRAYLIST, LINKEDLIST e VECTOR. Cada implementação possui suas características sendo apropriadas para contextos diferentes.

```
1 ArrayList arrayList = new ArrayList();
```

```
1 LinkedList linkedList = new LinkedList();
```

```
1 Vector vector = new Vector();
```

Podemos aplicar o polimorfismo e referenciar objetos criados a partir das classes: ARRAYLIST, LINKEDLIST e VECTOR como LIST.

```
1 List list = new ArrayList();
```

```
1 List list = new LinkedList();
```

```
1 List list = new Vector();
```

16.1.1 Método: add(Object)

O método ADD(OBJECT) adiciona uma referência no final da lista e aceita referências de qualquer tipo.

```
1 List list = ...
2
3 list.add(258);
4 list.add("Rafael Cosentino");
5 list.add(1575.76);
6 list.add("Marcelo Martins");
```

16.1.2 Método: add(int, Object)

O método ADD(INT, OBJECT) adiciona uma referência em uma determinada posição da lista.

```
1 List list = ...
2
3 list.add(0, "Jonas Hirata");
4 list.add(1, "Rafael Cosentino");
5 list.add(1, "Marcelo Martins");
6 list.add(3, "Thiago Thies");
```

16.1.3 Método: size()

O método SIZE() informa a quantidade de elementos armazenado na lista.

```
List list = ...

list.add("Jonas Hirata");
list.add("Rafael Cosentino");
list.add("Marcelo Martins");
list.add("Thiago Thies");

// quantidade = 4
int quantidade = list.size();
```

16.1.4 Método: clear()

O método CLEAR() remove todos os elementos da lista.

```
List list = ...

list.add("Jonas Hirata");
list.add("Rafael Cosentino");
list.add("Marcelo Martins");
list.add("Thiago Thies");

// quantidade = 4
int quantidade = list.size();

list.clear();

// quantidade = 0
quantidade = list.size();
```

16.1.5 Método: contains(Object)

Para verificar se um elemento está contido em uma lista podemos utilizar o método CONTAINS(OBJECT)

```
List list = ...

list.add("Jonas Hirata");
list.add("Rafael Cosentino");

// x = true
boolean x = list.contains("Jonas Hirata");

// x = false
x = list.contains("Daniel Machado");
```

16.1.6 Método: remove(Object)

Podemos retirar elementos de uma lista através do método REMOVE(OBJECT). Este método remove a primeira ocorrência do elemento passado como parâmetro.

```
List list = ...

list.add("Jonas Hirata");

// x = true
boolean x = list.contains("Jonas Hirata");

list.remove("Jonas Hirata");

// x = false
x = list.contains("Jonas Hirata");
```

16.1.7 Método: remove(int)

Outra maneira para retirar elementos de uma lista através do método REMOVE(INT).

```
List list = ...

list.add("Jonas Hirata");

// x = true
boolean x = list.contains("Jonas Hirata");

list.remove(0);

// x = false
x = list.contains("Jonas Hirata");
```

16.1.8 Método: get(int)

Para recuperar um elemento de uma determinada posição de uma lista podemos utilizar o método GET(INT).

```
List list = ...

list.add("Jonas Hirata");

// nome = "Jonas Hirata"
String nome = list.get(0);
```

16.1.9 Método: indexOf(Object)

Para descobrir o índice da primeira ocorrência de um determinado elemento podemos utilizar o método INDEXOF(OBJECT).

```
List list = ...

list.add("Jonas Hirata");

// indice = 0
int indice = list.indexOf("Jonas Hirata");
```

16.1.10 Benchmarking

As três principais implementações da interface LIST (ARRAYLIST, LINKEDLIST e VECTOR) possuem desempenho diferentes para cada operação. O desenvolvedor deve escolher a implementação de acordo com a sua necessidade.

Operação	ArrayList	LinkedList
Adicionar ou Remover do final da lista	☺	☺
Adicionar ou Remover do começo da lista	☹	☺
Acessar elementos pela posição	☺	☹

Os métodos da classe VECTOR possui desempenho um pouco pior do que os da classe ARRAYLIST. Porém, a classe VECTOR implementa lógica de sincronização de threads.

16.2 Exercícios

1. Crie um projeto no eclipse chamado **Collections**.

2. Vamos calcular o tempo das operações principais das listas.

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3  import java.util.List;
4
5  public class TestaAdicionaNoFinal {
6      public static void main(String[] args) {
7          ArrayList arrayList = new ArrayList();
8          LinkedList linkedList = new LinkedList();
9
10         long tempo = TestaAdicionaNoFinal.AdicionaNoFinal(arrayList);
11         System.out.println("ArrayList: " + tempo + "ms");
12
13         tempo = TestaAdicionaNoFinal.AdicionaNoFinal(linkedList);
14         System.out.println("LinkedList: " + tempo + "ms");
15
16     }
17
18     public static long AdicionaNoFinal(List lista) {
19         long inicio = System.currentTimeMillis();
20
21         int size = 100000;
22
23         for (int i = 0; i < size; i++) {
24             lista.add(i);
25         }
26
27         long fim = System.currentTimeMillis();
28
29         return fim - inicio;
30     }
31 }
```

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3  import java.util.List;
4
5  public class TestaAdicionaNoComeco {
6      public static void main(String[] args) {
7          ArrayList arrayList = new ArrayList();
8          LinkedList linkedList = new LinkedList();
9
10         long tempo = TestaAdicionaNoComeco.AdicionaNoComeco(arrayList);
11         System.out.println("ArrayList: " + tempo + "ms");
12
13         tempo = TestaAdicionaNoComeco.AdicionaNoComeco(linkedList);
14         System.out.println("LinkedList: " + tempo + "ms");
15
16     }
17
18     public static long AdicionaNoComeco(List lista) {
19         long inicio = System.currentTimeMillis();
20
21         int size = 100000;
22
23         for (int i = 0; i < size; i++) {
24             lista.add(0, i);
25         }
26
27         long fim = System.currentTimeMillis();
28
29         return fim - inicio;
30     }
31 }
```

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3  import java.util.List;
4
5  public class TestaGet {
6      public static void main(String[] args) {
7          ArrayList arrayList = new ArrayList();
8          LinkedList linkedList = new LinkedList();
9
10         long tempo = TestaGet.Get(arrayList);
11         System.out.println("ArrayList: " + tempo + "ms");
12
13         tempo = TestaGet.Get(linkedList);
14         System.out.println("LinkedList: " + tempo + "ms");
15
16     }
17
18     public static long Get(List lista) {
19
20         int size = 100000;
21
22         for (int i = 0; i < size; i++) {
23             lista.add(i);
24         }
25
26         long inicio = System.currentTimeMillis();
27
28         for (int i = 0; i < size; i++) {
29             lista.get(i);
30         }
31
32         long fim = System.currentTimeMillis();
33
34         return fim - inicio;
35     }
36 }
```

3. (Opcional) Teste o desempenho para remover elementos do começo ou do fim das listas.

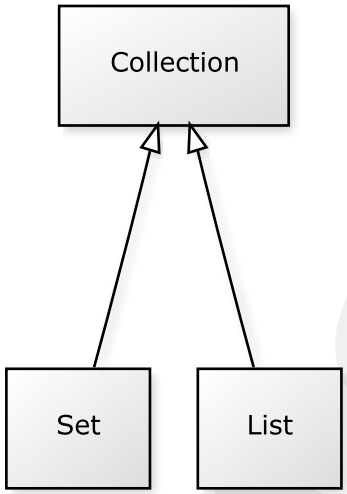
16.3 Conjuntos

Os conjuntos diferem das listas pois não permitem elementos repetidos e não possuem ordem. Como os conjuntos não possuem ordem as operações baseadas em índice que existem nas listas não aparecem nos conjuntos.

SET é a interface Java que define os métodos que um conjunto deve implementar. As principais implementações da interface SET são: HASHSET e TREESET. Cada implementação possui suas características sendo apropriadas para contextos diferentes.

16.4 Coleções

Há semelhanças conceituais entre os conjuntos e as listas por isso existe uma super interface chamada COLLECTION para as interfaces LIST e SET.



Dessa forma, podemos referenciar como COLLECTION qualquer lista ou conjunto.

16.5 Exercícios

4. Vamos comparar o tempo do método CONTAINS() das listas e dos conjuntos.

```
1  import java.util.ArrayList;
2  import java.util.Collection;
3  import java.util.HashSet;
4
5  public class TestaContains {
6      public static void main(String[] args) {
7          ArrayList arrayList = new ArrayList();
8          HashSet hashSet = new HashSet();
9
10         long tempo = TestaContains.Contains(arrayList);
11         System.out.println("ArrayList: " + tempo + "ms");
12
13         tempo = TestaContains.Contains(hashSet);
14         System.out.println("HashSet: " + tempo + "ms");
15
16     }
17
18     public static long Contains(Collection colecao) {
19
20         int size = 100000;
21
22         for (int i = 0; i < size; i++) {
23             colecao.add(i);
24         }
25
26         long inicio = System.currentTimeMillis();
27
28         for (int i = 0; i < size; i++) {
29             colecao.contains(i);
30         }
31
32         long fim = System.currentTimeMillis();
33
34         return fim - inicio;
35     }
36 }
```

16.6 Laço foreach

As listas podem ser iteradas com um laço FOR tradicional.

```
List lista = ...

for(int i = 0; i < lista.size(); i++) {
    Object x = lista.get(i);
}
```

Porém, como os conjuntos não são baseados em índice eles não podem ser iterados com um laço FOR tradicional. Além disso, mesmo para as listas o FOR tradicional nem sempre é eficiente pois o método GET() para determinadas implementações de lista é lento (ex: LINKEDLIST).

A maneira mais eficiente para percorrer uma coleção é utilizar um laço **foreach**.

```
Collection colecao = ...

for(Object x : colecao) {

}
```

16.7 Generics

As coleções armazenam referências de qualquer tipo. Dessa forma, quando recuperamos um elemento de uma coleção temos que trabalhar com referências do tipo OBJECT.

```
Collection colecao = ...

colecao.add("Rafael Cosentino");

for(Object x : colecao) {
    System.out.println(x);
}
```

Porém, normalmente, precisamos tratar os objetos de forma específica pois queremos ter acesso aos métodos específicos desses objetos. Nesses casos, devemos fazer casting nas referências.

```
Collection colecao = ...

colecao.add("Rafael Cosentino");

for(Object x : colecao) {
    String s = (String)x;
    System.out.println(s.toUpperCase());
}
```

O casting de referência é arriscado pois em tempo de compilação não temos garantia que ele está correto. Dessa forma, corremos o risco de obter um erro de execução.

Para ter certeza da tipagem dos objetos em tempo de compilação, devemos aplicar o recurso do **Generics**. Com este recurso podemos determinar o tipo de objeto que queremos armazenar em uma coleção no momento em que ela é criada. A partir daí, o compilador não permitirá que elementos não compatíveis com o tipo escolhido sejam adicionados na coleção. Isso garante o tipo do elemento no momento em que ele é recuperado da coleção e elimina a necessidade de casting.

```
1 Collection<String> colecao = new HashSet<String>();
2
3 colecao.add("Rafael Cosentino");
4
5 for(String x : colecao) {
6     System.out.println(x.toUpperCase());
7 }
```

16.8 Exercícios

5. Vamos testar o desempenho do FOR tradicional e do foreach.

```
1  import java.util.LinkedList;
2  import java.util.List;
3
4  public class TestaContains {
5      public static void main(String[] args) {
6          LinkedList<Integer> linkedList = new LinkedList<Integer>();
7          int size = 100000;
8
9          for (int i = 0; i < size; i++) {
10             linkedList.add(i);
11         }
12
13         long tempo = TestaContains.forTradicional(linkedList);
14         System.out.println("For Tradicional: " + tempo + "ms");
15
16         tempo = TestaContains.foreach(linkedList);
17         System.out.println("Foreach: " + tempo + "ms");
18
19     }
20
21     public static long forTradicional(List<Integer> lista) {
22         long inicio = System.currentTimeMillis();
23
24         for (int i = 0; i < lista.size(); i++) {
25             int x = lista.get(i);
26         }
27
28         long fim = System.currentTimeMillis();
29
30         return fim - inicio;
31     }
32
33     public static long foreach(List<Integer> lista) {
34         long inicio = System.currentTimeMillis();
35
36         for (int x : lista) {
37
38         }
39
40         long fim = System.currentTimeMillis();
41
42         return fim - inicio;
43     }
44 }
```

Capítulo 17

Apêndice -Swing

A plataforma Java oferece recursos sofisticados para construção de interfaces gráficas de usuário **GUI**. Esses recursos fazem parte do framework **Java Foundation Classes (JFC)**. Eis uma visão geral do JFC:

Java Web Start: Permite que aplicações Java sejam facilmente implantadas nas máquinas dos usuários.

Java Plug-In: Permite que **applets** executem dentro dos principais navegadores.

Java 2D: Possibilita a criação de imagens e gráficos 2D.

Java 3D: Possibilita a manipulação de objetos 3D.

Java Sound: Disponibiliza a manipulação de sons para as aplicações Java.

AWT (Abstract Window Toolkit): Conjunto básico de classes e interfaces que definem os componentes de uma janela desktop. AWT é a base para Java Swing API.

Swing: Conjunto sofisticado de classes e interfaces que definem os componentes visuais e serviços necessários para construir uma interface gráfica de usuário.

17.1 Componentes

Os itens que aparecem em uma interface gráfica de usuário (janelas, caixas de texto, botões, listas, caixas de seleção, entre outros) são chamados de componentes. Alguns componentes podem ser colocados dentro de outros componentes, por exemplo, uma caixa de texto dentro de uma janela.

O primeiro passo para construir uma interface gráfica de usuário é conhecer os principais componentes do Java Swing API.

17.1.1 JFrame

A classe JFRAME define janelas com título, borda e alguns itens definidos pelo sistema operacional como botão para minimizar ou maximizar.

```
1 JFrame frame = new JFrame("K19 - Java OO");
2 frame.setSize(300, 200);
3 frame.setVisible(true);
```

É possível associar uma das ações abaixo ao botão de fechar janela.

DO_NOTHING_ON_CLOSE: Não faz nada.

HIDE_ON_CLOSE: Esconde a janela (Padrão no JFRAME).

DISPOSE_ON_CLOSE: Fecha a janela (Mais utilizado em janelas internas).

EXIT_ON_CLOSE: Fecha a aplicação (System.exit(0)).

```
1 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Por padrão, o visual das janelas utiliza o estilo definido pelo sistema operacional. Mas, podemos alterar esse comportamento padrão.

17.1.2 JPanel

A classe JPANEL define um componente que basicamente é utilizado para agrupar nas janelas outros componentes como caixas de texto, botões, listas, entre outros.

Normalmente, criamos um objeto da classe JPANEL e associamos a um objeto da classe JFRAME para agrupar todo o conteúdo da janela.

```
1 JFrame frame = new JFrame("K19 - Java OO");
2
3 JPanel panel = new JPanel();
4
5 frame.setContentPane(panel);
```

17.1.3 JTextField e JLabel

A classe JTEXTFIELD define os campos de texto que podem ser preenchidos pelo usuário. A classe JLABEL define rótulos que podem ser utilizados por exemplo em caixas de texto.

```
1  JFrame frame = new JFrame("K19 - Java OO");
2  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
3
4  JPanel panel = new JPanel();
5
6  JLabel rotulo = new JLabel();
7  rotulo.setText("Nome: ");
8  panel.add(rotulo);
9
10 JTextField textField = new JTextField(40);
11 panel.add(textField);
12
13 frame.setContentPane(panel);
14
15 frame.pack();
16 frame.setVisible(true);
```

17.1.4 JTextArea

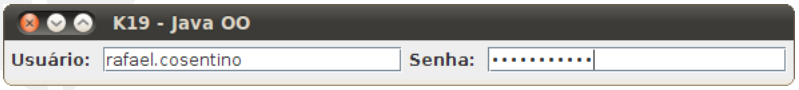
Para textos maiores podemos aplicar o componente definido pela classe JTEXTAREA.

```
1  JTextArea textArea = new JTextArea(10, 20);
```

17.1.5 JPasswordField

Em formulários que necessitam de caixa de texto para digitar senhas, podemos aplicar o componente definido pela classe JPASSWORDFIELD. O conteúdo digitado na caixa de texto gerado pelo componente da classe JPASSWORDFIELD não é apresentado ao usuário.

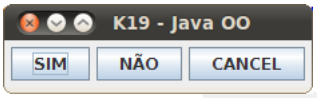
```
1  JPasswordField passwordField = new JPasswordField(20);
```



17.1.6 JButton

Os botões que permitem que os usuários indiquem quais ações ele deseja que a aplicação execute podem ser criados através do componente definido pela classe JBUTTON.

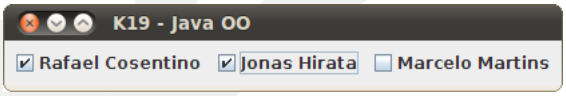
```
1 JButton button1 = new JButton("SIM");
2
3 JButton button2 = new JButton("NÃO");
4
5 JButton button3 = new JButton("CANCEL");
```



17.1.7 JCheckBox

Podemos criar formulários com checkbox's utilizando o componente da classe JCHECKBOX.

```
1 JCheckBox checkBox1 = new JCheckBox("Rafael Cosentino");
2
3 JCheckBox checkBox2 = new JCheckBox("Jonas Hirata");
4
5 JCheckBox checkBox3 = new JCheckBox("Marcelo Martins");
```



17.1.8 JComboBox

Podemos criar formulários com combobox's utilizando o componente da classe JCOMBOBOX.

```
1 String[] items = new String[3];
2 items[0] = "Rafael Cosentino";
3 items[1] = "Jonas Hirata";
4 items[2] = "Marcelo Martins";
5
6 JComboBox comboBox = new JComboBox(items);
```

17.2 Layout Manager

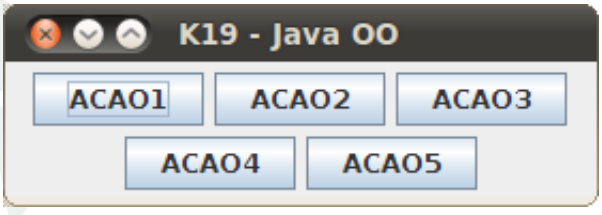
Muitas pessoas consideram que uma das tarefas mais complicadas quando se utiliza a Java Swing API é o posicionamento e o tamanho dos componentes. Posicionamento e tamanho dos componentes Java Swing são controlados por **Layout Manager's**.

Um Layout Manager é um objeto Java associado a um componente Java Swing que na maioria dos casos é um componente de background como uma janela ou um painel. Um Layout Manager controla os componentes que estão dentro do componente ao qual ele está associado.

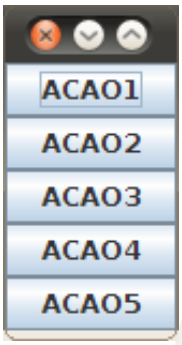
Os quatro principais Layout Manager's do Java Swing são:

BorderLayout: Esse Layout Manager divide a área de um componente de background em cinco regiões (norte, sul, leste, oeste e centro). Somente um componente pode ser adicionado em cada região. Eventualmente, o BorderLayout altera o tamanho preferencial dos componentes para torná-los compatíveis com o tamanho das regiões. O BorderLayout é o Layout Manager padrão de um JFrame.

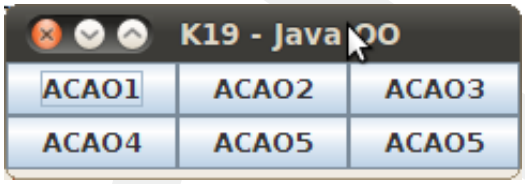
FlowLayout: Esse Layout Manager arranja os componentes da esquerda para direita e quando o tamanho horizontal não é suficiente ele "pula" para a próxima "linha". O FlowLayout não altera o tamanho preferencial dos componentes. O FlowLayout é o Layout Manager padrão de um JPanel.



BoxLayout: Esse Layout Manager arranja os componentes de cima para baixo "quebrando linha" a cada componente adicionado. O BoxLayout não altera o tamanho preferencial dos componentes.



GridLayout: Esse Layout Manager divide a área de um componente de background em células semelhantemente a uma tabela. As células possuem o mesmo tamanho.



17.3 Events, Listeners e Sources

A principal função de uma interface gráfica de usuário é permitir interação entre usuários e aplicação. Os usuários interagem com uma aplicação clicando em botões, preenchendo caixas de texto, movimentando o mouse, entre outros. Essas ações dos usuários disparam **eventos** que são processados pela aplicação através de **listeners** (callbacks).

Para criar um listener, devemos implementar a interface correspondente ao tipo de evento que queremos tratar. Eis algumas das interfaces que devemos implementar quando queremos criar um listener.

ActionListener: Essa interface deve ser implementada quando desejamos tratar eventos como por exemplo cliques em botões, seleção de items de um menu ou teclar enter dentro de uma caixa de texto.

MouseListener: Essa interface deve ser implementada quando desejamos tratar eventos como clique dos botões do mouse.

KeyListener: Essa interface deve ser implementada quando desejamos tratar eventos de pressionar ou soltar teclas do teclado.

17.3.1 Exemplo

Vamos criar um listener para executar quando o usuário clicar em um botão. O primeiro passo é definir uma classe que implemente ACTIONLISTENER.

```
1 class MeuListener implements ActionListener {
2     public void actionPerformed(ActionEvent e) {
3         JButton button = (JButton) e.getSource();
4         button.setText("clicado");
5     }
6 }
```

O método ACTIONPERFORMED() deverá ser executado quando algum botão for clicado pelo usuário. Perceba que este método recebe um referência de um objeto da classe ACTIONEVENT que representa o evento que ocorreu. Através do objeto que representa o evento do clique do usuário em algum botão podemos recuperar a fonte do evento que no caso é o próprio botão com o método GETSOURCE() e alterar alguma característica da fonte.

O segundo passo é associar esse listener aos botões desejados.

```
1 JButton button1 = new JButton("ACAO1");
2 JButton button2 = new JButton("ACAO2");
3
4 MeuListener listener = new MeuListener();
5
6 button1.addActionListener(listener);
7 button2.addActionListener(listener);
```

17.4 Exercícios

1. Crie um projeto no eclipse chamado **Swing**.

2. Crie uma tela de login com caixas de texto e rótulos para o nome de usuário e senha e um botão para logar.

```
1  public class Teste {
2      public static void main(String[] args) {
3          JFrame frame = new JFrame("K19 - Login");
4          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5
6          JPanel panel = new JPanel();
7
8          JLabel label1 = new JLabel("Usuário: ");
9
10         JTextField textField = new JTextField(20);
11
12         JLabel label2 = new JLabel("Senha: ");
13
14         JPasswordField passwordField = new JPasswordField(20);
15
16         JButton button = new JButton("Logar");
17
18         panel.add(label1);
19         panel.add(textField);
20         panel.add(label2);
21         panel.add(passwordField);
22         panel.add(button);
23
24         frame.setContentPane(panel);
25
26         frame.pack();
27         frame.setVisible(true);
28     }
29 }
```

3. Redimensione a janela e observe o que ocorre com os elementos e pense o que determina o comportamento observado.

4. Altere o Layout Manager do painel utilizado na tela de login para GRIDLAYOUT adicionando a linha a seguir logo após a criação do JPANEL.

```
1 panel.setLayout(new GridLayout(3, 2));
```

   Execute novamente o teste e observe o resultado. Depois tente redimensionar a tela para observar o comportamento.

5. (Opcional) Observando a tela obtida no exercício anterior, verificamos que o botão é colocado na primeira coluna do grid gerado pelo GRIDLAYOUT. Tente fazer o botão aparecer na segunda coluna.

Capítulo 18

Apêndice -Threads

Se pensarmos nos programas que utilizamos comumente no dia a dia, conseguiremos chegar a seguinte conclusão: um programa executa um conjunto de tarefas relativamente independentes entre si. Por exemplo, um navegador pode baixar vários arquivos diferentes além de permitir a navegação. Um software de visualização de vídeos além de reproduzir imagens também reproduzir sons.

Se pensarmos em sistemas corporativos, também chegamos na mesma conclusão: um sistema corporativo executa um conjunto de tarefas relativamente independentes entre si. Por exemplo, dois ou mais usuários acessando o mesmo sistema para fazer coisas diferentes.

Já que um programa ou um sistema corporativo executa tarefas relativamente independentes entre si podemos pensar em executá-las simultaneamente. A primeira grande limitação para executar tarefas simultaneamente é a quantidade de unidades de processamento (cpu's) disponíveis.

Em geral, a regra para saber quantas tarefas podemos executar simultaneamente é bem simples: se temos N unidades de processamento podemos executar no máximo N tarefas. Uma exceção a essa regra ocorre quando a tecnologia hyperthreding é aplicada. Essa tecnologia permite o aproveitamento do tempo ocioso de uma cpu.

Geralmente, a quantidade de tarefas que desejamos executar é maior do que a quantidades de cpu's. Supondo que as tarefas sejam executadas sem interrupção do começo até o fim então com alta probabilidade teríamos constantemente um cenário com todas as cpu's ocupadas com tarefas grandes e demoradas e diversas tarefas menores que poderiam ser executadas rapidamente esperando em uma fila. Esse cenário não é adequado para sistema com alta interatividade com usuários pois diminui a sua responsividade (o efeito de uma ação do usuário demora).

Para aumentar a responsividade das aplicações, o sistema operacional faz um revezamento das tarefas que precisam executar. Isso evita que tarefas demoradas travem a utilização das cpu's tornando a interatividade mais satisfatória.

O trabalho do desenvolvedor é definir quais são as tarefas que uma aplicação deve realizar e determinar quando elas devem executar.

18.1 Definindo Tarefas - (Runnables)

Para definir as tarefas que uma aplicação Java deve executar, devemos criar classes que implementam a interface RUNNABLE. Essa interface possui apenas um método (RUN()). O método RUN() é conceitualmente análogo ao método MAIN() pois o primeiro funciona como

"ponto de partida" de uma tarefa de uma aplicação o segundo funciona como "ponto de partida" de uma aplicação.

Veja alguns exemplos de tarefas definidas em Java implementando a interface RUNNABLE:

```
1 class TarefaImprimeOi implements Runnable {
2     public void run() {
3         for(int i = 0; i < 100; i++) {
4             System.out.println("OI");
5         }
6     }
7 }
```

```
1 class TarefaSomaAte100 implements Runnable {
2     public void run() {
3         int soma = 0;
4         for(int i = 1; i <= 100; i++) {
5             soma += i;
6         }
7         System.out.println(soma);
8     }
9 }
```

18.2 Executando Tarefas

As tarefas são executadas "dentro" de objetos da classe THREAD. Então, para cada tarefa que desejamos executar, devemos criar um objeto da classe THREAD e associá-lo a tarefa.

```
1 TarefaImprimeOi tarefa1 = new TarefaImprimeOi();
2 TarefaImprimeOi tarefa2 = new TarefaImprimeOi();
3 TarefaSomaAte100 tarefa3 = new TarefaSomaAte100();
4
5 Thread thread1 = new Thread(tarefa1);
6 Thread thread2 = new Thread(tarefa2);
7 Thread thread3 = new Thread(tarefa3);
```

Depois de associar uma tarefa (objeto de uma classe que implementa RUNNABLE) a um objeto da classe THREAD, devemos "disparar" a execução da thread através do método START().

```
1 TarefaImprimeOi tarefa = new TarefaImprimeOi();
2 Thread thread = new Thread(tarefa);
3 thread.start();
```

Podemos "disparar" diversas threads e elas poderão ser executadas simultaneamente de acordo com o revezamento que a máquina virtual e o sistema operacional aplicarem.

18.3 Exercícios

1. Crie um projeto no eclipse chamado **Threads**.

2. Defina uma tarefa para imprimir mensagens na tela.

```
1  class TarefaImprimeMensagens implements Runnable {
2
3      private String msg;
4
5      public TarefaImprimeMensagens(String msg) {
6          this.msg = msg;
7      }
8
9      public void run() {
10         for(int i = 0; i < 100000; i++) {
11             System.out.println(i + " : " + this.msg);
12         }
13     }
14 }
```

3. Crie tarefas e associe-as com threads para executá-las.

```
1  class Teste {
2      public static void main(String[] args) {
3          TarefaImprimeMensagens tarefa1 = new TarefaImprimeMensagens("K19");
4          TarefaImprimeMensagens tarefa2 = new TarefaImprimeMensagens("Java");
5          TarefaImprimeMensagens tarefa3 = new TarefaImprimeMensagens("Web");
6
7          Thread thread1 = new Thread(tarefa1);
8          Thread thread2 = new Thread(tarefa2);
9          Thread thread3 = new Thread(tarefa3);
10
11         thread1.start();
12         thread2.start();
13         thread3.start();
14     }
15 }
```

Execute o teste!

18.4 Controlando a Execução das Tarefas

Controlar a execução das tarefas de uma aplicação pode ser bem complicado. Esse controle envolve, por exemplo, decidir quando uma tarefa pode executar, quando não pode, a ordem na qual duas ou mais tarefas devem ser executadas, etc.

A própria classe THREAD oferece alguns métodos para controlar a execução das tarefas de uma aplicação. Veremos o funcionamento alguns desses métodos.

18.4.1 sleep()

Durante a execução de uma thread, se o método SLEEP() for chamado a thread ficará sem executar pelo menos durante a quantidade de tempo passada como parâmetro para este método.

```
1  // Faz a thread corrente dormir por 3 segundos
2  Thread.sleep(3000);
```

InterruptedException

Uma thread que está "dormindo" pode ser interrompida por outra thread. Quando isso ocorrer, a thread que está "dormindo" recebe uma INTERRUPTEDEXCEPTION.

```
try {
    Thread.sleep(3000);
} catch (InterruptedException e) {

}
```

18.4.2 join()

Uma thread pode "pedir" para esperar o término de outra thread para continuar a execução através do método JOIN(). Esse método também pode lançar uma INTERRUPTEDEXCEPTION.

```
TarefaImprimeMensagens tarefa = new TarefaImprimeMensagens("K19");
Thread thread = new Thread(tarefa);
thread.start();

try {
    thread.join();
} catch (InterruptedException e) {

}
```

18.5 Exercícios

4. Altere a classe TAREFAIMPRIMEMENSAGENS do projeto **Threads**, adicionando uma chamada ao método SLEEP().

```
class TarefaImprimeMensagens implements Runnable {

    private String msg;

    public TarefaImprimeMensagens(String msg) {
        this.msg = msg;
    }

    public void run() {
        for(int i = 0; i < 100000; i++) {
            System.out.println(i + " : " + this.msg);

            if(i % 1000 == 0) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {

                }
            }
        }
    }
}
```

Execute o teste novamente!

5. Crie um nova classe para testar o método JOIN()

```
1  class Teste2 {
2      public static void main(String[] args) throws InterruptedException {
3          TarefaImprimeMensagens tarefa = new TarefaImprimeMensagens("K19");
4
5          Thread thread = new Thread(tarefa);
6
7          thread.start();
8
9          thread.join();
10
11         System.out.println("FIM");
12     }
13 }
```

Execute o segundo teste e observe a mensagem “FIM” no console. Depois, comente a linha do método JOIN() e verifique o que mudou.

Capítulo 19

Apêndice - Socket

Os computadores ganham muito mais importância quando conectados entre si para trocar informações. A troca de dados entre computadores de uma mesma rede é realizada através de **sockets**. Um socket permite que um computador receba ou envia dados para outros computadores da mesma rede.

19.1 Socket

A classe SOCKET define o funcionamento dos sockets em Java.

```
1 Socket socket = new Socket("184.72.247.119", 1000);
```

Um dos construtores da classe SOCKET recebe o **ip** e a **porta** da máquina que queremos nos conectar. Após a conexão através do socket ser estabelecida, podemos criar um objeto da classe PRINTSTREAM e outro da classe SCANNER associados ao socket para facilitar o envio e o recebimento dados respectivamente.

```
1 Socket socket = new Socket("184.72.247.119", 1000);
2
3 PrintStream saida = new PrintStream(socket.getOutputStream());
4
5 Scanner entrada = new Scanner(socket.getInputStream());
```

O funcionamento da classe PRINTSTREAM e SCANNER foi visto no capítulo 15.

19.2 ServerSocket

Um server socket é um tipo especial de socket. Ele deve ser utilizado quando desejamos que uma aplicação seja capaz de aguardar que outras aplicações possivelmente em outras máquinas se conectem a ela.

A classe SERVERSOCKET define o funcionamento de um server socket.s

```
1 ServerSocket severSocket = new ServerSocket(1000);
2
3 Socket socket = serverSocket.accept();
```

Um dos construtores da classe SERVERSOCKET recebe a porta que será utilizada pelas aplicações que querem estabelecer uma conexão com a aplicação do server socket.

O método ACCEPT() espera alguma aplicação se conectar na porta do server socket. Quando isso acontecer, o método ACCEPT() cria um novo socket em outra porta associado à aplicação que se conectou para realizar a troca de dados e liberar a porta do server socket para outras aplicações que desejem se conectar.

Se uma aplicação deseja permitir que diversas aplicação se conectem a ela então é necessário chamar várias vezes o método ACCEPT(). Este método pode ser colocado em um laço.

```
1 ServerSocket severSocket = new ServerSocket(1000);
2
3 while(true) {
4     Socket socket = serverSocket.accept();
5
6 }
```

Cada iteração do laço acima estabelece uma conexão nova com uma aplicação cliente.

19.3 Exercícios

1. Crie um projeto no eclipse chamado **Sockets**.

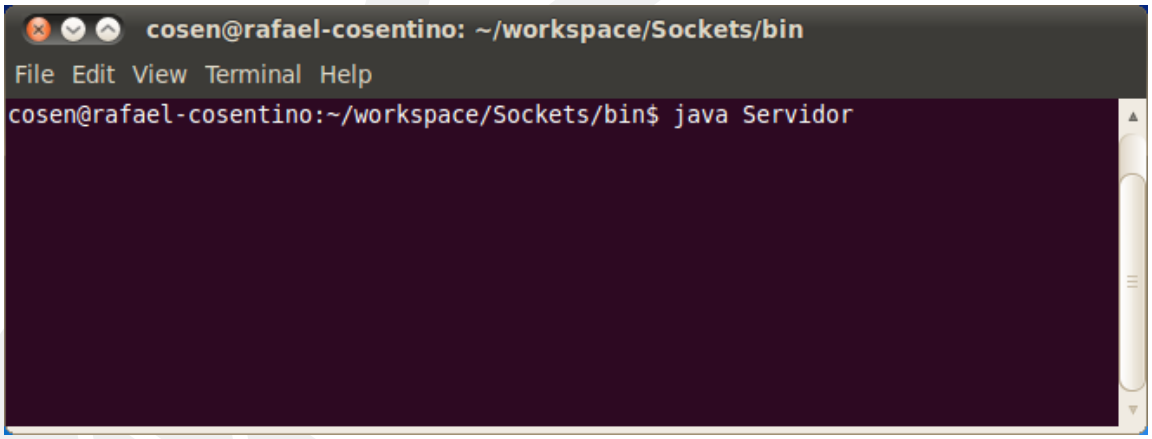
2. Crie o código de uma aplicação servidora.

```
1  public class Servidor {
2      public static void main(String[] args) throws IOException {
3          ServerSocket serverSocket = new ServerSocket(10000);
4
5          Socket socket = serverSocket.accept();
6
7          PrintStream saida = new PrintStream(socket.getOutputStream());
8
9          saida.println("Você se conectou ao servidor da K19!");
10     }
11 }
```

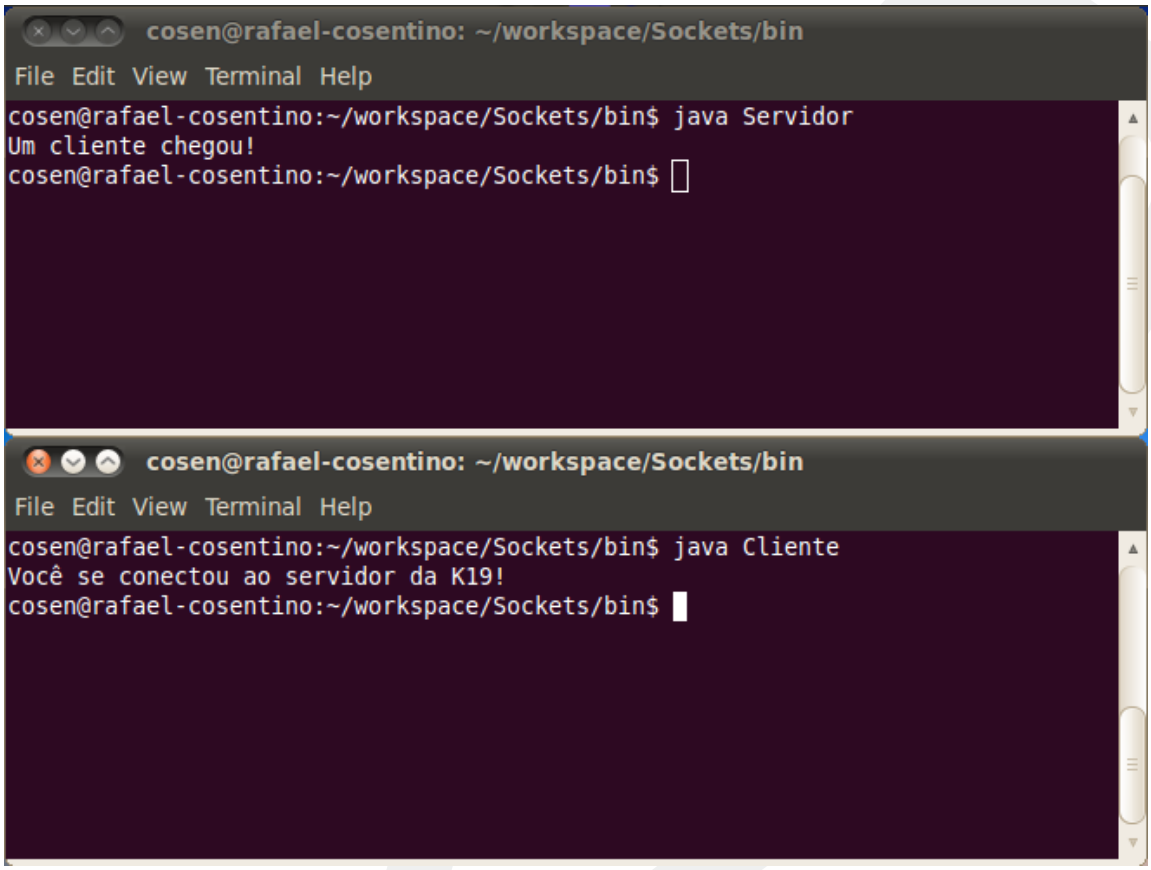
3. Crie o código de uma aplicação cliente.

```
1  public class Cliente {
2      public static void main(String[] args) throws Exception {
3          Socket socket = new Socket("127.0.0.1", 10000);
4
5          Scanner entrada = new Scanner(socket.getInputStream());
6
7          String linha = entrada.nextLine();
8
9          System.out.println(linha);
10     }
11 }
```

4. Abra um terminal, entre na pasta **bin** do projeto **Sockets** e execute a classe SERVIDOR.



5. Abra outro terminal, entre na pasta **bin** do projeto **Sockets** e execute a classe CLIENTE.


cosen@rafael-cosentino: ~/workspace/Sockets/bin
File Edit View Terminal Help
cosen@rafael-cosentino:~/workspace/Sockets/bin$ java Servidor
Um cliente chegou!
cosen@rafael-cosentino:~/workspace/Sockets/bin$
cosen@rafael-cosentino: ~/workspace/Sockets/bin
File Edit View Terminal Help
cosen@rafael-cosentino:~/workspace/Sockets/bin$ java Cliente
Você se conectou ao servidor da K19!
cosen@rafael-cosentino:~/workspace/Sockets/bin$

Capítulo 20

Apêndice - Chat K19

20.1 Arquitetura do Sistema

O sistema de chat da K19 é dividido em aplicação servidora e aplicação cliente. A aplicação servidora não possuirá interface gráfica e sua principal tarefa é distribuir as mensagens enviadas pelos usuários. A aplicação cliente possuirá interface gráfica que permita que um usuário envie e receba mensagens.

Criaremos neste capítulo um esqueleto de cada uma das principais classes do sistema de chat da K19.

20.2 Aplicação servidora

20.2.1 Registrador

Na aplicação servidora, um objeto registrador deve esperar novos usuários do chat da K19 e realizar todo processo de registro de novos usuários quando alguém chegar.

```
1 public class Registrador {
2     public void aguardaUsuario() {
3
4     }
5 }
```

20.2.2 Receptor

Para cada usuário cadastrado no chat da K19 deve ser criado um objeto da classe RECEPTOR. A tarefa de um objeto da classe RECEPTOR é aguardar as mensagens enviadas pelo usuário correspondente.

```
1 public class Receptor {
2     public void aguardaMensagens() {
3
4     }
5 }
```

20.2.3 Emissor

Para cada usuário cadastrado no chat da K19 deve ser criado um objeto da classe EMISSOR. A tarefa de um objeto da classe EMISSOR é enviar as mensagens do chat para o usuário correspondente.

```
1 public class Emissor {
2     public void envia(String mensagem) {
3
4     }
5 }
```

20.2.4 Distribuidor

Na aplicação servidora, deve existir um objeto da classe DISTRIBUIDOR que tem como tarefa receber as mensagens dos receptores e repassá-las para os emissores.

```
1 public class Distribuidor {
2     public void distribuiMensagem(String mensagem) {
3
4     }
5 }
```

20.3 Aplicação cliente

20.3.1 EmissorDeMensagem

Na aplicação cliente, deve existir um objeto da classe EMISSORDEMENSAGEM que envia as mensagens digitadas pelo usuário para a aplicação servidora.

```
1 public class EmissorDeMensagem {
2     public void enviaMensagem(String mensagem) {
3
4     }
5 }
```

20.3.2 ReceptorDeMensagem

Na aplicação cliente, deve existir um objeto da classe RECEPTORDEMENSAGEM que aguarda as mensagens enviadas pela aplicação servidora e as apresenta par o usuário.

```
1 public class ReceptorDeMensagem {
2     public void aguardaMensagem() {
3
4     }
5 }
```

20.4 Exercícios

1. Crie um projeto no eclipse chamado **K19-chat-server**.

2. Crie um projeto no eclipse chamado **K19-chat-client**.

3. No projeto **K19-chat-server** crie uma classe para definir os emissores.

```
1  import java.io.PrintStream;
2
3  public class Emissor {
4
5      private PrintStream saida;
6
7      public Emissor(PrintStream saida) {
8          this.saida = saida;
9      }
10
11     public void envia(String mensagem) {
12         this.saida.println(mensagem);
13     }
14 }
```

   Cada emissor possui uma saída de dados relacionada a um cliente conectado ao chat. Para criação de um emissor, a saída deve ser passada como parâmetro através do construtor.

   Quando alguma mensagem de algum cliente conectado ao chat chegar no servidor, o distribuidor chamará o método ENVIA() passando a mensagem para o emissor enviá-la ao cliente correspondente.

4. No projeto **K19-chat-server** crie uma classe para definir o distribuidor de mensagens.

```
1  import java.util.Collection;
2
3  public class Distribuidor {
4      private Collection<Emissor> emissores;
5
6      public void adicionaEmissor(Emissor emissor) {
7          this.emissores.add(emissor);
8      }
9
10     public void distribuiMensagem(String mensagem) {
11         for (Emissor emissor : this.emissores) {
12             emissor.envia(mensagem);
13         }
14     }
15 }
```

   O distribuidor possui uma coleção de emissores, um emissor para cada cliente conectado.

   Quando um novo cliente se conecta ao chat, o método ADICIONAEMISSOR() permite que um novo emissor seja adicionada na coleção do distribuidor.

   Quando algum cliente envia uma mensagem, o método DISTRIBUIMENSAGEM() permite que a mesma seja enviada para todos os clientes conectados.

5. No projeto **K19-chat-server** crie uma classe para definir os receptores.

```
1  import java.util.Scanner;
2
3  public class Receptor implements Runnable {
4      private Scanner entrada;
5      private Distribuidor distribuidor;
```

```
6
7       public Receptor(Scanner entrada, Distribuidor distribuidor) {
8           this.entrada = entrada;
9           this.distribuidor = distribuidor;
10      }
11
12      public void run() {
13          while (this.entrada.hasNextLine()) {
14              String mensagem = this.entrada.nextLine();
15              this.distribuidor.distribuiMensagem(mensagem);
16          }
17      }
18  }
```

Cada receptor possui uma entrada de dados relacionada a um cliente conectado ao chat e o distribuidor. Para criação de um receptor, devem ser passados a entrada relacionada a um cliente e o distribuidor através do construtor.

Como o servidor de chat precisa receber simultaneamente as mensagens de todos os clientes, cada receptor será executado em uma thread por isso a classe RECEPTOR implementa a interface RUNNABLE.

No método RUN(), o receptor entra em um laço esperando que uma mensagem seja enviada pelo seu cliente para repassá-la ao distribuidor.

6. No projeto **K19-chat-server** crie uma classe para definir o registrador.

```
1   import java.io.IOException;
2   import java.io.PrintStream;
3   import java.net.ServerSocket;
4   import java.net.Socket;
5   import java.util.Scanner;
6
7   public class Registrador implements Runnable {
8
9       private Distribuidor distribuidor;
10      private ServerSocket serverSocket;
11
12      public Registrador(Distribuidor distribuidor, ServerSocket serverSocket) {
13          this.distribuidor = distribuidor;
14          this.serverSocket = serverSocket;
15
16      }
17
18      public void run() {
19          while (true) {
20              try {
21                  Socket socket = this.serverSocket.accept();
22
23                  Scanner entrada = new Scanner(socket.getInputStream());
24                  PrintStream saida = new PrintStream(socket.getOutputStream());
25
26                  Receptor receptor = new Receptor(entrada, this.distribuidor);
27                  Thread pilha = new Thread(receptor);
28                  pilha.start();
29
30                  Emissor emissor = new Emissor(saida);
31
32                  this.distribuidor.adicionaEmissor(emissor);
33
34              } catch (IOException e) {
35                  System.out.println("ERRO");
36              }
37          }
38      }
39  }
```

7. No projeto **K19-chat-server** crie uma classe para inicializar o servidor.

```
import java.io.IOException;
import java.net.ServerSocket;

public class Server {
    public static void main(String[] args) throws IOException {
        Distribuidor distribuidor = new Distribuidor();

        ServerSocket serverSocket = new ServerSocket(10000);

        Registrador registrador = new Registrador(distribuidor, serverSocket);
        Thread pilha = new Thread(registrador);
        pilha.start();
    }
}
```

8. No projeto **K19-chat-client** crie uma classe para definir o emissor de mensagens.

```
import java.io.PrintStream;

public class EmissorDeMensagem {
    private PrintStream saida;

    public EmissorDeMensagem(PrintStream saida) {
        this.saida = saida;
    }

    public void envia(String mensagem) {
        this.saida.println(mensagem);
    }
}
```

9. No projeto **K19-chat-client** crie uma classe para definir a tela em Java Swing do chat.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class TelaK19Chat {

    private final JFrame frame;
    private final JPanel panel;
    private final JScrollPane scrollPane;
    private final JTextArea textArea1;
    private final JLabel label1;
    private final JTextField textField;
    private final JButton button;

    private final EmissorDeMensagem emissorDeMensagem;

    public TelaK19Chat(EmissorDeMensagem emissor) {
        this.emissorDeMensagem = emissor;

        this.frame = new JFrame("K19 - Chat");
        this.panel = new JPanel();
        this.textArea1 = new JTextArea(10, 60);
        this.textArea1.setEditable(false);
        this.scrollPane = new JScrollPane(this.textArea1);
        this.label1 = new JLabel("Digite uma mensagem...");
        this.textField = new JTextField(60);
```

```
34          this.button = new JButton("Enviar");
35
36          this.frame.setContentPane(this.panel);
37          this.panel.add(this.scrollPane);
38          this.panel.add(this.label1);
39          this.panel.add(this.textField);
40          this.panel.add(button);
41
42          class EnviaMensagemListener implements ActionListener {
43
44              public void actionPerformed(ActionEvent e) {
45                  emissorDeMensagem.envia(textField.getText());
46                  textField.setText("");
47              }
48          }
49
50          this.button.addActionListener(new EnviaMensagemListener());
51
52          this.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
53          this.frame.setSize(700, 300);
54          this.frame.setVisible(true);
55
56      }
57
58      public void adicionaMensagem(String mensagem) {
59          this.textArea1.append(mensagem + "\n");
60
61      }
62  }
```

10. No projeto **K19-chat-client** crie uma classe para definir o receptor de mensagens.

```
1   import java.util.Scanner;
2
3   public class ReceptorDeMensagem implements Runnable {
4       private Scanner entrada;
5
6       private TelaK19Chat telaK19Chat;
7
8       public ReceptorDeMensagem(Scanner entrada, TelaK19Chat telaK19Chat) {
9           this.entrada = entrada;
10          this.telaK19Chat = telaK19Chat;
11      }
12
13      public void run() {
14          while (this.entrada.hasNextLine()) {
15              String mensagem = this.entrada.nextLine();
16              this.telaK19Chat.adicionaMensagem(mensagem);
17          }
18      }
19  }
```

11. No projeto **K19-chat-client** crie uma classe para inicializar o cliente.

```
1   import java.io.PrintStream;
2   import java.net.Socket;
3   import java.util.Scanner;
4
5   public class Client {
6       public static void main(String[] args) throws Exception {
7
8           Socket socket = new Socket("IP DO SERVIDOR", 10000);
9
10          PrintStream saida = new PrintStream(socket.getOutputStream());
11
12          Scanner entrada = new Scanner(socket.getInputStream());
13
```

```
14          EmissorDeMensagem emissor = new EmissorDeMensagem(saida);
15
16          TelaK19Chat telaK19Chat = new TelaK19Chat(emissor);
17
18          ReceptorDeMensagem receptor = new ReceptorDeMensagem(entrada,
19                  telaK19Chat);
20          Thread pilha = new Thread(receptor);
21          pilha.start();
22      }
23  }
```