



K9

TREINAMENTOS

# Orientação a Objetos em Java



# Orientação a Objetos em Java

**19 de janeiro de 2016**

As apostilas atualizadas estão disponíveis em [www.k19.com.br](http://www.k19.com.br)

Esta apostila contém:

- 168 exercícios de fixação.
- 41 exercícios complementares.
- 0 desafios.
- 0 questões de prova.

<b>Sumário</b>	<b>i</b>
<b>Sobre a K19</b>	<b>1</b>
<b>Seguro Treinamento</b>	<b>2</b>
<b>Termo de Uso</b>	<b>3</b>
<b>Cursos</b>	<b>4</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Objetivo . . . . .	1
1.2 Orientação a Objetos . . . . .	1
1.3 Plataforma Java . . . . .	2
1.4 Plataforma Java VS Orientação a Objetos . . . . .	2
1.5 Eclipse . . . . .	2
<b>2 Lógica</b>	<b>5</b>
2.1 O que é um Programa? . . . . .	5
2.2 Linguagem de Máquina . . . . .	5
2.3 Linguagem de Programação . . . . .	6
2.4 Compilador . . . . .	6
2.5 Máquinas Virtuais . . . . .	6
2.6 Exemplo de programa Java . . . . .	9
2.7 Método Main - Ponto de Entrada . . . . .	10

---

2.8 Exercícios de Fixação . . . . .	11
2.9 Variáveis . . . . .	16
2.10 Operadores . . . . .	18
2.11 IF-ELSE . . . . .	21
2.12 WHILE . . . . .	21
2.13 FOR . . . . .	22
2.14 Exercícios de Fixação . . . . .	22
2.15 Exercícios Complementares . . . . .	26
<b>3 Orientação a Objetos</b> . . . . .	<b>29</b>
3.1 Domínio e Aplicação . . . . .	29
3.2 Objetos, Atributos e Métodos . . . . .	30
3.3 Classes . . . . .	32
3.4 Referências . . . . .	35
3.5 Manipulando Atributos . . . . .	36
3.6 Valores Padrão . . . . .	36
3.7 Exercícios de Fixação . . . . .	37
3.8 Exercícios Complementares . . . . .	42
3.9 Relacionamentos: Associação, Agregação e Composição . . . . .	42
3.10 Exercícios de Fixação . . . . .	44
3.11 Exercícios Complementares . . . . .	46
3.12 Métodos . . . . .	46
3.13 Exercícios de Fixação . . . . .	48
3.14 Exercícios Complementares . . . . .	49
3.15 Sobrecarga (Overloading) . . . . .	49
3.16 Exercícios de Fixação . . . . .	50
3.17 Construtores . . . . .	51
3.18 Exercícios de Fixação . . . . .	54
3.19 Referências como parâmetro . . . . .	59
3.20 Exercícios de Fixação . . . . .	59
<b>4 Arrays</b> . . . . .	<b>61</b>
4.1 Criando um array . . . . .	61
4.2 Modificando o conteúdo de um array . . . . .	62
4.3 Acessando o conteúdo de um array . . . . .	62
4.4 Percorrendo um Array . . . . .	63
4.5 foreach . . . . .	63
4.6 Operações . . . . .	64
4.7 Exercícios de Fixação . . . . .	64
4.8 Exercícios Complementares . . . . .	68
<b>5 Atributos e Métodos de Classe</b> . . . . .	<b>69</b>
5.1 Atributos Estáticos . . . . .	69
5.2 Métodos Estáticos . . . . .	70
5.3 Exercícios de Fixação . . . . .	71
5.4 Exercícios Complementares . . . . .	75
<b>6 Encapsulamento</b> . . . . .	<b>77</b>
6.1 Atributos Privados . . . . .	77
6.2 Métodos Privados . . . . .	77

---

6.3	Métodos Públicos . . . . .	78
6.4	Implementação e Interface de Uso . . . . .	79
6.5	Por quê encapsular? . . . . .	79
6.6	Celular - Escondendo a complexidade . . . . .	79
6.7	Carro - Evitando efeitos colaterais . . . . .	80
6.8	Máquinas de Porcarias - Aumentando o controle . . . . .	81
6.9	Acessando ou modificando atributos . . . . .	81
6.10	Getters e Setters . . . . .	82
6.11	Exercícios de Fixação . . . . .	83
6.12	Exercícios Complementares . . . . .	87
<b>7</b>	<b>Herança</b>	<b>89</b>
7.1	Reutilização de Código . . . . .	89
7.2	Uma classe para todos os serviços . . . . .	89
7.3	Uma classe para cada serviço . . . . .	90
7.4	Uma classe genérica e várias específicas . . . . .	91
7.5	Preço Fixo . . . . .	93
7.6	Reescrita de Método . . . . .	93
7.7	Fixo + Específico . . . . .	94
7.8	Construtores e Herança . . . . .	95
7.9	Exercícios de Fixação . . . . .	96
7.10	Exercícios Complementares . . . . .	100
<b>8</b>	<b>Polimorfismo</b>	<b>101</b>
8.1	Controle de Ponto . . . . .	101
8.2	Modelagem dos funcionários . . . . .	102
8.3	É UM (extends) . . . . .	102
8.4	Melhorando o controle de ponto . . . . .	103
8.5	Exercícios de Fixação . . . . .	104
8.6	Exercícios Complementares . . . . .	106
<b>9</b>	<b>Object</b>	<b>109</b>
9.1	Polimorfismo . . . . .	109
9.2	O método <code>toString()</code> . . . . .	110
9.3	O método <code>equals()</code> . . . . .	113
9.4	Exercícios de Fixação . . . . .	115
<b>10</b>	<b>Classes Abstratas</b>	<b>121</b>
10.1	Classes Abstratas . . . . .	121
10.2	Métodos Abstratos . . . . .	122
10.3	Exercícios de Fixação . . . . .	123
10.4	Exercícios Complementares . . . . .	126
<b>11</b>	<b>Interfaces</b>	<b>129</b>
11.1	Padronização . . . . .	129
11.2	Contratos . . . . .	129
11.3	Exemplo . . . . .	130
11.4	Polimorfismo . . . . .	131
11.5	Interface e Herança . . . . .	131
11.6	Exercícios de Fixação . . . . .	133

<b>12 Pacotes</b>	<b>137</b>
12.1 Organização . . . . .	137
12.2 O comando package . . . . .	137
12.3 sub-pacotes . . . . .	137
12.4 Unqualified Name vs Fully Qualified Name . . . . .	138
12.5 Classes ou Interfaces públicas . . . . .	138
12.6 Import . . . . .	139
12.7 Conflito de nomes . . . . .	140
12.8 Exercícios de Fixação . . . . .	140
<b>13 Exceptions</b>	<b>143</b>
13.1 Errors vs Exceptions . . . . .	143
13.2 Checked e Unchecked . . . . .	144
13.3 Lançando uma unchecked exception . . . . .	144
13.4 Lançando uma checked exception . . . . .	145
13.5 Capturando exceptions . . . . .	145
13.6 Exercícios de Fixação . . . . .	146
<b>14 String</b>	<b>149</b>
14.1 Pool de Strings . . . . .	149
14.2 Imutabilidade . . . . .	150
14.3 Métodos principais . . . . .	150
14.4 Exercícios de Fixação . . . . .	152
<b>15 Entrada e Saída</b>	<b>155</b>
15.1 Byte a Byte . . . . .	155
15.2 Scanner . . . . .	156
15.3 PrintStream . . . . .	156
15.4 Exercícios de Fixação . . . . .	157
15.5 Exercícios Complementares . . . . .	159
<b>16 Collections</b>	<b>161</b>
16.1 Listas . . . . .	161
16.2 Exercícios de Fixação . . . . .	164
16.3 Conjuntos . . . . .	167
16.4 Coleções . . . . .	167
16.5 Exercícios de Fixação . . . . .	168
16.6 Laço foreach . . . . .	169
16.7 Generics . . . . .	169
16.8 Exercícios de Fixação . . . . .	170
<b>A Documentação</b>	<b>173</b>
A.1 A ferramenta javadoc . . . . .	174
A.2 Exercícios de Fixação . . . . .	176
<b>B Swing</b>	<b>181</b>
B.1 Componentes . . . . .	181
B.2 Layout Manager . . . . .	184
B.3 Events, Listeners e Sources . . . . .	186
B.4 Exercícios de Fixação . . . . .	187

<b>C Empacotamento</b>	<b>189</b>
C.1 Empacotando uma biblioteca . . . . .	189
C.2 Empacotando uma applicação . . . . .	189
C.3 Exercícios de Fixação . . . . .	189
<b>D Threads</b>	<b>193</b>
D.1 Definindo Tarefas - (Runnables) . . . . .	193
D.2 Executando Tarefas . . . . .	194
D.3 Exercícios de Fixação . . . . .	194
D.4 Controlando a Execução das Tarefas . . . . .	196
D.5 Exercícios de Fixação . . . . .	197
<b>E Visibilidade</b>	<b>199</b>
<b>F Socket</b>	<b>201</b>
F.1 Socket . . . . .	201
F.2 ServerSocket . . . . .	201
F.3 Exercícios de Fixação . . . . .	202
<b>G Chat K19</b>	<b>205</b>
G.1 Arquitetura do Sistema . . . . .	205
G.2 Aplicação servidora . . . . .	205
G.3 Aplicação cliente . . . . .	206
G.4 Exercícios de Fixação . . . . .	206
<b>H Quizzes</b>	<b>215</b>
<b>I Respostas</b>	<b>217</b>





## Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.



## Seguro Treinamento

**Na K19 o aluno faz o curso quantas vezes quiser!**

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



# Termo de Uso

## Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.



## Conheça os nossos cursos

-  K01 - Lógica de Programação
-  K02 - Desenvolvimento Web com HTML, CSS e JavaScript
-  K03 - SQL e Modelo Relacional
-  K11 - Orientação a Objetos em Java
-  K12 - Desenvolvimento Web com JSF2 e JPA2
-  K21 - Persistência com JPA2 e Hibernate
-  K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI
-  K23 - Integração de Sistemas com Webservices, JMS e EJB
-  K41 - Desenvolvimento Mobile com Android
-  K51 - Design Patterns em Java
-  K52 - Desenvolvimento Web com Struts
-  K31 - C# e Orientação a Objetos
-  K32 - Desenvolvimento Web com ASP.NET MVC

**[www.k19.com.br/cursos](http://www.k19.com.br/cursos)**

# INTRODUÇÃO



## Objetivo

O objetivo fundamental dos treinamentos da K19 é transmitir os conhecimentos necessários para que os seus alunos possam atuar no mercado de trabalho na área de desenvolvimento de software.

As plataformas **Java** e **.NET** são as mais utilizadas no desenvolvimento de software. Para utilizar os recursos oferecidos por essas plataformas de forma eficiente, é necessário possuir conhecimento sólido em **orientação a objetos**.



## Orientação a Objetos

Um **modelo de programação** ou **paradigma de programação** é um conjunto de princípios, ideias, conceitos e abstrações utilizado para o desenvolvimento de uma aplicação.

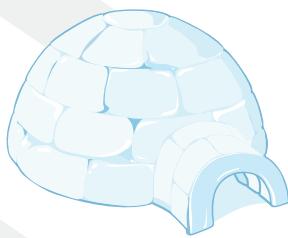


### Analogia

Para entender melhor o que são os modelos de programação, podemos compará-los com padrões arquiteturais utilizados por diferentes povos para construção de casas. As características ambientais definem quais técnicas devem ser adotadas para a construção das moradias. Analogamente, devemos escolher o modelo de programação mais adequado às necessidades da aplicação que queremos desenvolver.



CABANA DE ÍNDIO



IGLU



CASA OCIDENTAL

Figura 1.1: Moradias

O modelo de programação mais adotado no desenvolvimento de sistemas corporativos é o modelo **orientado a objetos**. Esse modelo é utilizado com o intuito de obter alguns benefícios específicos. Normalmente, o principal benefício desejado é facilitar a **manutenção** das aplicações.

Em geral, os conceitos do modelo de programação orientado a objetos diminuem a complexidade do desenvolvimento de sistemas que possuem as seguintes características:

- Sistemas com grande quantidade de funcionalidades desenvolvidos por uma equipe.
- Sistemas que serão utilizados por um longo período de tempo e sofrerão alterações constantes.



## Plataforma Java

A plataforma Java será objeto de estudo desse treinamento. Mas, devemos salientar que os conceitos de orientação a objetos que serão vistos poderão ser aplicados também na plataforma .NET.

No primeiro momento, os dois elementos mais importantes da plataforma Java são:

- A linguagem de programação Java.
- O ambiente de execução Java.

A linguagem de programação Java permite que os conceitos de orientação a objetos sejam utilizados no desenvolvimento de uma aplicação.

O ambiente de execução Java permite que uma aplicação Java seja executada em sistemas operacionais diferentes.



Figura 1.2: Plataforma Java



## Plataforma Java VS Orientação a Objetos

Do ponto de vista do aprendizado, é interessante tentar definir o que é mais importante: a plataforma Java ou a orientação a objetos. Consideramos que a orientação a objetos é mais importante pois ela é aplicada em muitas outras plataformas.



## Eclipse

Na prática, alguma ferramenta de desenvolvimento é adotada para aumentar a produtividade. Essas ferramentas são chamadas **IDE** (Integrated Development Environment - Ambiente de Desen-

volvimento 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 é o **Eclipse**. Essa ferramenta é bem abrangente e oferece recursos sofisticados para o desenvolvimento de uma aplicação Java. Além disso, ela é gratuita.

Através do site <http://www.eclipse.org/>, as diversas distribuições do Eclipse podem ser obtidas.





# O que é um Programa?

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

Quando as apurações dos votos das eleições no Brasil eram realizadas manualmente, o tempo para obter os resultados era alto e havia alta probabilidade de 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 drasticamente.

Basicamente, os computadores são capazes de executar instruções matemáticas mais rapidamente do que o homem. Essa simples capacidade permite que eles resolvam problemas complexos de maneira mais eficiente. Porém, eles não possuem a inteligência necessária para definir quais instruções devem ser executadas para resolver uma determinada tarefa. Por outro lado, os seres humanos possuem essa inteligência. Dessa forma, uma pessoa precisa definir um **roteiro** com a sequência de comandos necessários para realizar uma determinada tarefa e depois passar para um computador executar esse roteiro. Formalmente, esses roteiros são chamados de **programas**.

Os programas 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.



## Linguagem de Máquina

Os computadores só sabem ler instruções escritas em **linguagem de máquina**. Uma instrução escrita em linguagem de máquina é uma sequência formada por “0s” e “1s” que representa a ação que um computador deve executar.

*Figura 2.1: Código de Máquina.*

Teoricamente, as pessoas poderiam escrever os programas diretamente em linguagem de máquina. Na prática, ninguém faz isso pois é uma tarefa muito complicada e demorada.

Um arquivo contendo as instruções de um programa em Linguagem de Máquina é chamado de **executável**.



## Linguagem de Programação

Como vimos anteriormente, escrever um programa em linguagem de máquina é totalmente inviável para uma pessoa. Para resolver esse problema, surgiram as *linguagens de programação*, que tentam se aproximar das linguagens humanas. Confira um trecho de um código escrito com a linguagem de programação Java:

```

1 class OláMundo {
2     public static void main(String[] args) {
3         System.out.println("Olá Mundo");
4     }
5 }
```

Código Java 2.1: OláMundo.java

Por enquanto você pode não entender muito do que está escrito, porém fica bem claro que um programa escrito dessa forma fica bem mais fácil de ser lido.

Um arquivo contendo as instruções de um programa em linguagem de programação é chamado de **arquivo fonte**.



## Compilador

Por um lado, os computadores processam apenas instruções em linguagem de máquina. Por outro lado, as pessoas definem as instruções em linguagem de programação. Dessa forma, é necessário traduzir o código escrito em linguagem de programação por uma pessoa para um código em linguagem de máquina para que um computador possa processar. Essa tradução é realizada por programas especiais chamados **compiladores**.

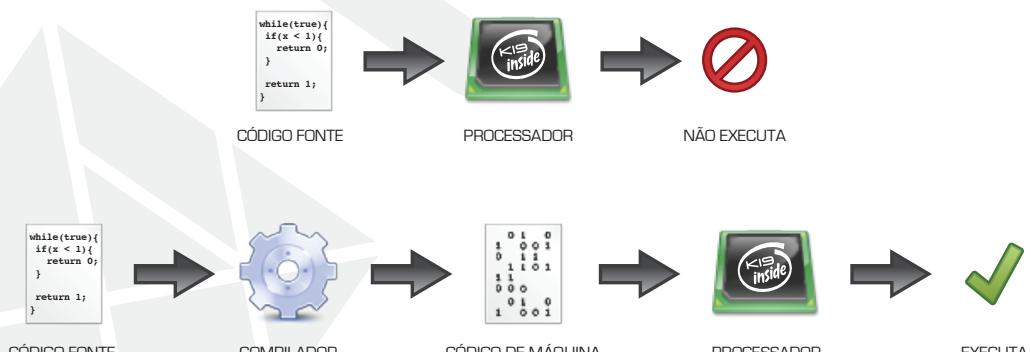


Figura 2.2: Processo de compilação e execução de um programa.



## Máquinas Virtuais

Assim como as pessoas podem se comunicar através de línguas diferentes, os computadores podem se comunicar através de linguagens de máquina diferentes. A linguagem de máquina de um computador é definida pela **arquitetura do processador** desse computador. Há diversas arquiteturas diferentes (Intel, ARM, PowerPC, etc) e cada uma delas define uma linguagem de máquina diferente. Em outras palavras, um programa pode não executar em computadores com processadores de arquiteturas diferentes.

Os computadores são controlados por um **sistema operacional** que oferece diversas bibliotecas necessárias para o desenvolvimento das aplicações que podem ser executadas através dele. Sistemas operacionais diferentes (Windows, Linux, Mac OS X, etc) possuem bibliotecas diferentes. Em outras palavras, um programa pode não executar em computadores com sistemas operacionais diferentes.

Portanto, para determinar se um código em linguagem de máquina pode ou não ser executada por um computador, devemos considerar a arquitetura do processador e o sistema operacional desse computador.

Algumas bibliotecas específicas de sistema operacional são chamadas diretamente pelas instruções em linguagem de programação. Dessa forma, geralmente, o código fonte está “amarrado” a uma plataforma (sistema operacional + arquitetura de processador).

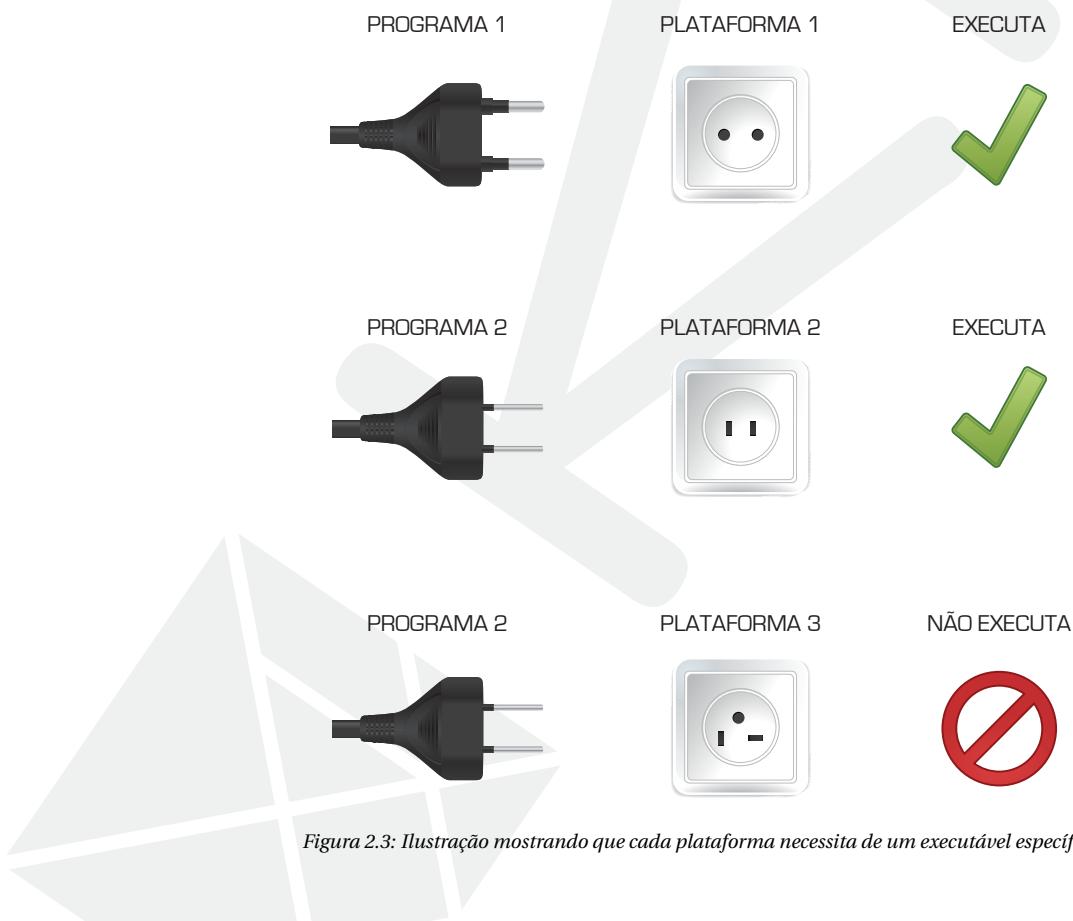


Figura 2.3: Ilustração mostrando que cada plataforma necessita de um executável específico.

Uma empresa que deseja ter a sua aplicação disponível para diversos sistemas operacionais (Windows, Linux, Mac OS X, etc), e diversas arquiteturas de processador (Intel, ARM, PowerPC, etc), terá que desenvolver versões diferentes do código fonte para cada plataforma (sistema operacional + arquitetura de processador). Isso pode causar um impacto financeiro nessa empresa que inviabiliza o negócio.

Para tentar resolver o problema do desenvolvimento de aplicações multiplataforma, surgiu o conceito de *máquina virtual*.

Uma máquina virtual funciona como uma camada a mais entre o código compilado e a plataforma. Quando compilamos um código fonte, estamos criando um executável que a máquina virtual saberá interpretar e ela é quem deverá traduzir as instruções do seu programa para a plataforma.

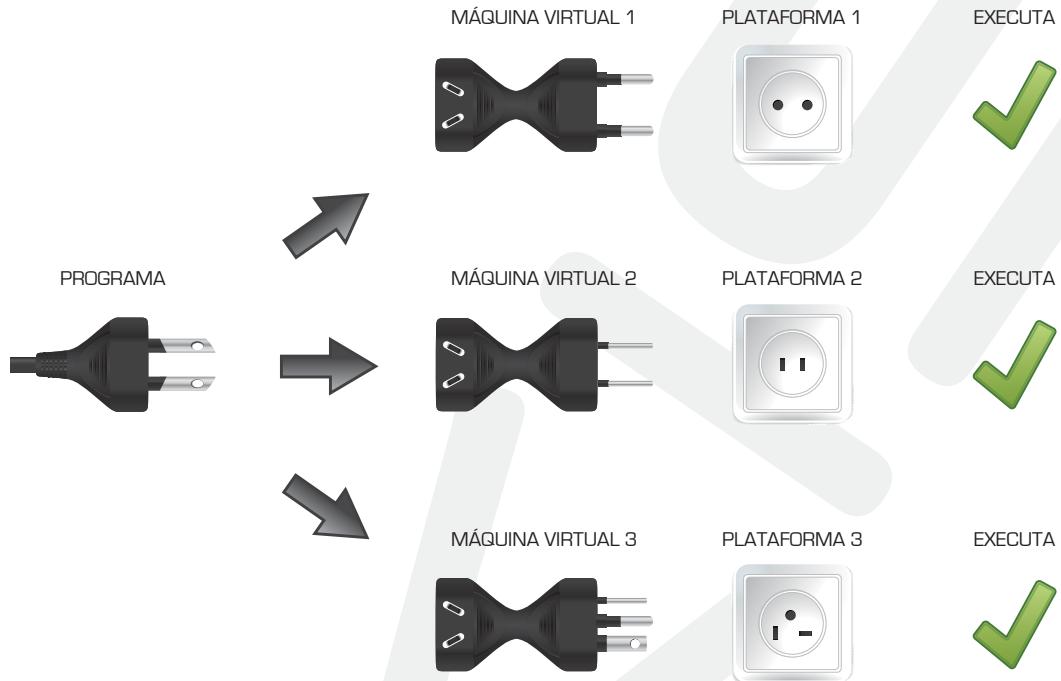


Figura 2.4: Ilustração do funcionamento da máquina virtual.

Tudo parece estar perfeito agora. Porém, olhando atentamente a figura acima, percebemos que existe a necessidade de uma máquina virtual para cada plataforma. Alguém poderia dizer que, de fato, o problema não foi resolvido, apenas mudou de lugar.

A diferença é que implementar a máquina virtual não é tarefa do programador que desenvolve as aplicações que serão executadas nela. A implementação da máquina virtual é responsabilidade de terceiros, que geralmente são empresas bem conceituadas ou projetos de código aberto que envolvem programadores do mundo inteiro. Como maiores exemplos podemos citar a Oracle JVM (Java Virtual Machine) e OpenJDK JVM.

Uma desvantagem em utilizar uma máquina virtual para executar um programa é 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 com-

putador. A partir daí, esses trechos podem ser executados diretamente no processador 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.



### Mais Sobre

Geralmente, as máquinas virtuais utilizam uma estratégia de compilação chamada **Just-in-time compilation (JIT)**. Nessa abordagem, o código de máquina pode ser gerado diversas vezes durante o processamento de um programa com o intuito de melhorar a utilização dos recursos disponíveis em um determinado instante da execução.



## Exemplo de programa Java

Vamos criar um simples programa para entendermos como funciona o processo de compilação e execução. Utilizaremos a linguagem Java, que é amplamente adotada nas empresas. Observe o código do exemplo de um programa escrito em Java que imprime uma mensagem na tela:

```

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

Código Java 2.2: OlaMundo.java

O código fonte Java deve ser colocado em arquivos com a extensão **.java**. Agora, não é necessário entender todo o código do exemplo. Basta saber que toda aplicação Java precisa ter um método especial chamado **main** para executar.

O próximo passo é compilar o código fonte, para gerar um executável que possa ser processado pela máquina virtual do Java. O compilador padrão da plataforma Java (**javac**) pode ser utilizado para compilar esse arquivo. O compilador pode ser executado pelo **terminal**.

```

K19$ ls
OlaMundo.java
K19$ javac OlaMundo.java
K19$ ls
OlaMundo.class OlaMundo.java
```

Terminal 2.1: Compilando

O código gerado pelo compilador Java é armazenado em arquivos com a extensão **.class**. No exemplo, o programa gerado pelo compilador é colocado em um arquivo chamado *OlaMundo.class* e ele pode ser executado através de um terminal.

```

K19$ ls
OlaMundo.class OlaMundo.java
K19$ java OlaMundo
Olá Mundo
```

Terminal 2.2: Executando



### Importante

Antes de compilar e executar um programa escrito em Java, é necessário que você tenha instalado e configurado em seu computador o JDK (Java Development Kit). Consulte o artigo da K19, <http://www.k19.com.br/artigos/installando-o-jdk-java-development-kit/>.



### Mais Sobre

Quando uma aplicação ou biblioteca Java é composta por diversos arquivos **.class**, podemos “empacotá-los” em um único arquivo com a extensão **.jar** com o intuito de facilitar a distribuição da aplicação ou da biblioteca.



## 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 variações da assinatura 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)
```

*Código Java 2.3: Variações da Assinatura do Método Main*

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.

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

*Código Java 2.4: Imprimindo os parâmetros da linha de comando*

Os parâmetros devem ser passados imediatamente após o nome do programa. A execução do programa é mostrada na figura abaixo.

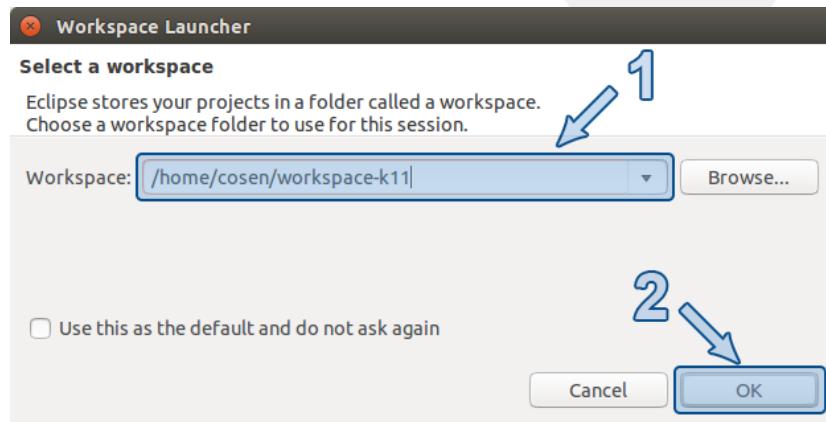
```
K19$ ls
Programa.class Programa.java
K19$ java Programa K19 Java Rafael Cosentino
K19
Java
Rafael
Cosentino
```

*Terminal 2.3: Imprimindo os parâmetros da linha de comando*



## Exercícios de Fixação

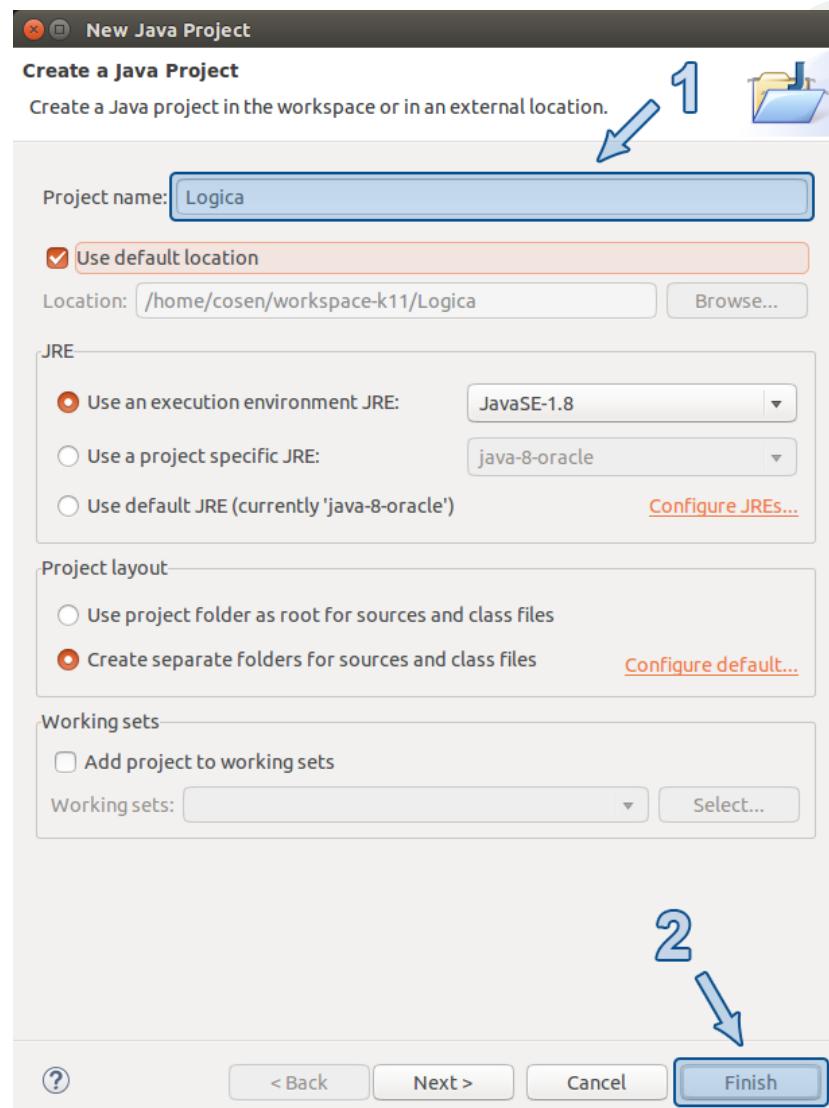
- 1 Abra o Eclipse e defina como workspace uma pasta chamada **workspace-k11** na sua área de trabalho.



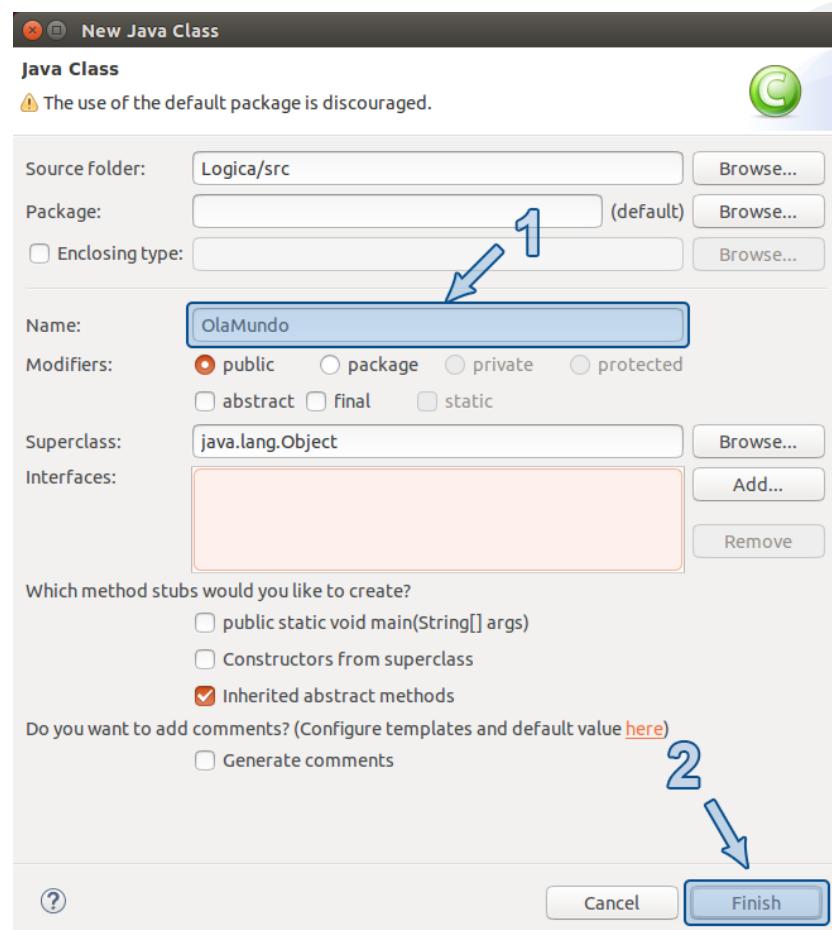
### Importante

Uma workspace do Eclipse é uma pasta que contém algumas configurações. Por padrão, os projetos também são armazenados nessa pasta. Podemos ter várias workspaces para organizar projetos e configurações.

- 2 O próximo passo é criar um novo projeto. Feche a Welcome Page. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga as imagens abaixo.



- 3 Agora, crie uma classe. Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e siga a imagem abaixo.



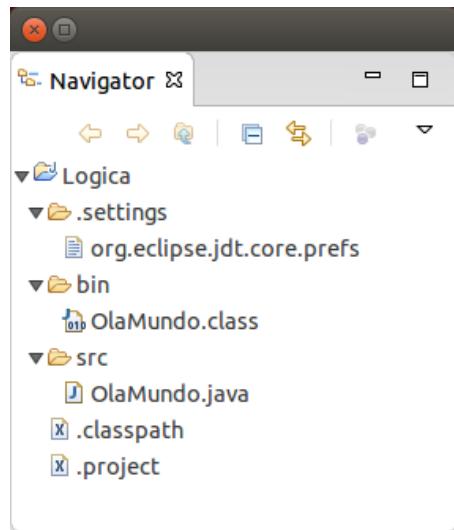
Veja o código gerado pelo Eclipse.

```

1 |
2 public class OlaMundo {
3 |
4 }
5 |

```

- 4 Em seguida, abra a visão Navigator. Digite “CTRL + 3” e pesquise por “Navigator”. Selecione a opção correspondente e veja a imagem abaixo.



Observe que o Eclipse criou duas pastas no projeto Logica. Uma chamada **src** e outra chamada **bin**. O código fonte do projeto é armazenado na pasta **src** e o código compilado na pasta **bin**. Por isso, o arquivo **OlaMundo.java** foi colocado na pasta **src** e o arquivo **OlaMundo.class** na pasta **bin**.

O Eclipse compila automaticamente o código fonte e salva o código compilado na pasta **bin**.

A pasta **.settings** e os arquivos **.classpath** e **.project** são itens de configuração. Não modificaremos esses itens.

- 5 Para prosseguir, acrescente o método **main** na classe **OlaMundo**. No corpo dessa classe, digite “main” seguido de “CTRL + ESPAÇO”. No menu de sugestões, selecione o modelo do método **main**.



```
1
2 public class OlaMundo {
3     public static void main(String[] args) {
4     }
5 }
6
7
```

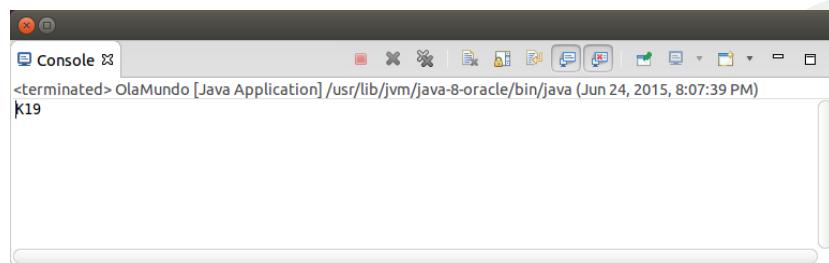
- 6 Utilize o método **println** para exibir uma mensagem no **Console**. No corpo do método `main`, digite “`sysout`” seguido de “`CTRL + ESPAÇO`” e defina a mensagem que deve ser exibida dentro de aspas dupla.

```
1
2 public class OlaMundo {
3     public static void main(String[] args) {
4         sysout
5     }
6 }
7
```

```
1
2 public class OlaMundo {
3     public static void main(String[] args) {
4         System.out.println();
5     }
6 }
7
```

```
1
2 public class OlaMundo {
3     public static void main(String[] args) {
4         System.out.println("K19");
5     }
6 }
7
```

- 7 Salve o arquivo `OlaMundo.java`. Em seguida, execute a classe `OlaMundo`. Selecione o arquivo `OlaMundo.java` e digite “`CTRL + F11`”. Observe a mensagem exibida na visão **Console**.



## Variáveis

Basicamente, o que um programa faz é manipular dados. Em geral, esses dados são armazenados em **variáveis** localizadas na memória RAM do computador. Uma variável pode guardar dados de vários tipos: números, textos, booleanos (verdadeiro ou falso), referências de objetos. Além disso, toda variável possui um nome que é utilizado quando a informação dentro da variável precisa ser manipulada pelo programa.

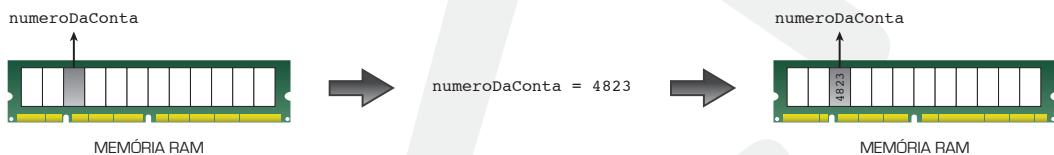


Figura 2.5: Processo de atribuição do valor numérico 4823 à variável numeroDaConta.

### 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) dentro de um escopo e um tipo de valor. As variáveis são acessadas pelos nomes e armazenam valores compatíveis com o seu tipo.

```

1 // Uma variável do tipo int chamada numeroDaConta.
2 int numeroDaConta;
3
4 // Uma variável do tipo double chamada precoDoProduto.
5 double precoDoProduto;

```

Código Java 2.5: Declaração de Variáveis



### Mais Sobre

Uma linguagem de programação é dita **estaticamente tipada** quando ela exige que os tipos das variáveis sejam definidos antes da compilação. A linguagem Java é uma linguagem estaticamente tipada.

Uma linguagem de programação é dita **fortemente tipada** quando ela exige que os valores armazenados em uma variável sejam compatíveis com o tipo da variável. A linguagem Java é uma linguagem fortemente tipada.



### Mais Sobre

Em geral, as linguagens de programação possuem convenções para definir os nomes das variáveis. Essas convenções ajudam o desenvolvimento de um código mais legível.

Na convenção de nomes da linguagem Java, os nomes das variáveis devem seguir o padrão **camel case** com a primeira letra minúscula (**lower camel case**). Veja alguns exemplos:

- nomeDoCliente
- numeroDeAprovados

A convenção de nomes da linguagem Java pode ser consultada na seguinte url: <http://www.oracle.com/technetwork/java/codeconv-138413.html>

A declaração de uma variável pode ser realizada em qualquer linha de um bloco. Não é necessário declarar todas as variáveis no começo do bloco como acontece em algumas linguagens de programação.

```

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

*Código Java 2.6: Declarando em qualquer linha de um bloco.*

Não podemos declarar duas variáveis com o mesmo nome em um único bloco ou escopo pois ocorrerá um erro de compilação.

```

1 // Declaração
2 int numero = 10;
3
4 // Erro de Compilação
5 int numero = 10;
```

*Código Java 2.7: Duas variáveis com o mesmo nome no mesmo bloco.*

## Inicialização

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 em uma variável.

```

1 // Declarações
2 int numero;
3 double preco;
4
5 // Inicialização
```

```

6 numero = 10;
7 // Uso Correto
8 System.out.println(numero);
10
11 // Erro de compilação
12 System.out.println(preco);

```

Código Java 2.8: Inicialização

## Tipos Primitivos

A linguagem Java define um conjunto de tipos básicos de dados que são chamados **tipos primitivos**. A tabela abaixo mostra os oito tipos primitivos da linguagem Java e os valores compatíveis.

Tipo	Descrição	Tamanho (“peso”)
byte	Valor inteiro entre -128 e 127 (inclusivo)	1 byte
short	Valor inteiro entre -32.768 e 32.767 (inclusivo)	2 bytes
int	Valor inteiro entre -2.147.483.648 e 2.147.483.647 (inclusivo)	4 bytes
long	Valor inteiro entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807 (inclusivo)	8 bytes
float	Valor com ponto flutuante entre $1,40129846432481707 \times 10^{-45}$ e $3,40282346638528860 \times 10^{38}$ (positivo ou negativo)	4 bytes
double	Valor com ponto flutuante entre $4,94065645841246544 \times 10^{-324}$ e $1,79769313486231570 \times 10^{308}$ (positivo ou negativo)	8 bytes
boolean	true ou false	1 bit
char	Um único caractere Unicode de 16 bits. Valor inteiro e positivo entre 0 (ou ‘\u0000’) e 65.535 (ou ‘\uffff’)	2 bytes

Tabela 2.1: Tipos primitivos de dados em Java.



### Importante

Nenhum tipo primitivo da linguagem Java permite o armazenamento de texto. O tipo primitivo **char** armazena apenas um caractere. Quando é necessário armazenar um texto, devemos utilizar o tipo **String**. Contudo, é importante salientar que o tipo String **não** é um tipo primitivo.



## Operadores

Para manipular os valores das variáveis de um programa, devemos utilizar os operadores oferecidos pela linguagem de programação adotada. A linguagem Java possui diversos operadores e os principais são categorizados da seguinte forma:

- Aritmético (+, -, \*, /, %)
- Atribuição (=, +=, -=, \*=, /=, %=)

- Relacional (`==`, `!=`, `<`, `<=`, `>`, `>=`)
- Lógico (`&&`, `||`)

## Aritmético

Os operadores aritméticos funcionam de forma muito semelhante aos operadores na matemática. Os operadores aritméticos são:

- Soma `+`
- Subtração `-`
- Multiplicação `*`
- Divisão `/`
- Módulo `%`

```

1 int umMaisUm = 1 + 1;           // umMaisUm = 2
2 int tresVezesDois = 3 * 2;     // tresVezesDois = 6
3 int quatroDivididoPor2 = 4 / 2; // quatroDivididoPor2 = 2
4 int seisModuloCinco = 6 % 5;   // seisModuloCinco = 1
5 int x = 7;
6 x = x + 1 * 2;                // x = 9
7 x = x - 3;                   // x = 6
8 x = x / (6 - 2 + (3*5)/(16-1)); // x = 2

```

Código Java 2.9: Exemplo de uso dos operadores aritméticos.



### Importante

O módulo de um número  $x$ , na matemática, é o valor numérico de  $x$  desconsiderando o seu sinal (valor absoluto). Na matemática expressamos o módulo da seguinte forma:  $| -2 | = 2$ .

Em linguagens de programação, o módulo de um número é o resto da divisão desse número por outro. No exemplo acima, o resto da divisão de 6 por 5 é igual a 1. Além disso, lemos a expressão `6%5` da seguinte forma: seis módulo cinco.



### Importante

As operações aritméticas em Java obedecem as mesmas regras da matemática com relação à precedência dos operadores e parênteses. Portanto, as operações são resolvidas a partir dos parênteses mais internos até os mais externos, primeiro resolvemos as multiplicações, divisões e os módulos. Em seguida, resolvemos as adições e subtrações.

## Atribuição

Nas seções anteriores, já vimos um dos operadores de atribuição, o operador `=` (igual). Os operadores de atribuição são:

- Simples `=`
- Incremental `+=`
- Decremental `-=`

- Multiplicativa \*=
- Divisória /=
- Modular %=

```

1 int valor = 1;      // valor = 1
2 valor += 2;        // valor = 3
3 valor -= 1;        // valor = 2
4 valor *= 6;        // valor = 12
5 valor /= 3;        // valor = 4
6 valor %= 3;        // valor = 1

```

*Código Java 2.10: Exemplo de uso dos operadores de atribuição.*

As instruções acima poderiam ser escritas de outra forma:

```

1 int valor = 1;      // valor = 1
2 valor = valor + 2; // valor = 3
3 valor = valor - 1; // valor = 2
4 valor = valor * 6; // valor = 12
5 valor = valor / 3; // valor = 4
6 valor = valor % 3; // valor = 1

```

*Código Java 2.11: O mesmo exemplo anterior, usando os operadores aritméticos.*

Como podemos observar, os operadores de atribuição, com exceção do simples (=), reduzem a quantidade de código escrito. Podemos dizer que esses operadores funcionam como “atalhos” para as operações que utilizam os operadores aritméticos.

## Relacional

Muitas vezes precisamos determinar a relação entre uma variável ou valor e outra outra variável ou valor. Nessas situações, utilizamos os operadores relacionais. As operações realizadas com os operadores relacionais devolvem valores do tipo primitivo boolean. Os operadores relacionais são:

- Igualdade ==
- Diferença !=
- Menor <
- Menor ou igual <=
- Maior >
- Maior ou igual >=

```

1 int valor = 2;
2 boolean t = false;
3 t = (valor == 2); // t = true
4 t = (valor != 2); // t = false
5 t = (valor < 2); // t = false
6 t = (valor <= 2); // t = true
7 t = (valor > 1); // t = true
8 t = (valor >= 1); // t = true

```

*Código Java 2.12: Exemplo de uso dos operadores relacionais em Java.*

## Lógico

A linguagem Java permite verificar duas ou mais condições através de operadores lógicos. Os operadores lógicos devolvem valores do tipo primitivo boolean. Os operadores lógicos são:

- “E” lógico &&
- “OU” lógico ||

```

1 int valor = 30;
2 boolean teste = false;
3 teste = valor < 40 && valor > 20;      // teste = true
4 teste = valor < 40 && valor > 30;      // teste = false
5 teste = valor > 30 || valor > 20;      // teste = true
6 teste = valor > 30 || valor < 20;      // teste = false
7 teste = valor < 50 && valor == 30;    // teste = true

```

*Código Java 2.13: Exemplo de uso dos operadores lógicos em Java.*



## IF-ELSE

O comportamento de uma aplicação pode ser influenciado por valores definidos pelos usuários. Por exemplo, considere um sistema de cadastro de produtos. Se um usuário tenta adicionar um produto com preço negativo, a aplicação não deve cadastrar esse produto. Caso contrário, se o preço não for negativo, o cadastro pode ser realizado normalmente.

Outro exemplo, quando o pagamento de um boleto é realizado em uma agência bancária, o sistema do banco deve verificar a data de vencimento do boleto para aplicar ou não uma multa por atraso.

Para verificar uma determinada condição e decidir qual bloco de instruções deve ser executado, devemos aplicar o comando **if**.

```

1 if (preco < 0) {
2     System.out.println("O preço do produto não pode ser negativo");
3 } else {
4     System.out.println("Produto cadastrado com sucesso");
5 }

```

*Código Java 2.14: Comando if*

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 não compila. O comando **else** e o seu bloco são opcionais.



## 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 10 linhas iguais a essa no código fonte:

```
1 System.out.println("Bom Dia");
```

Código Java 2.15: "Bom Dia"

Se ao invés de 10 vezes fosse necessário imprimir 100 vezes, já seriam 100 linhas iguais no código fonte. É muito trabalhoso utilizar essa abordagem para solucionar esse problema.

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 }
```

Código Java 2.16: Comando while

A variável **contador** indica o número de vezes que 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.



## FOR

O comando **for** é análogo ao **while**. A diferença entre esses dois comandos é que o **for** recebe três argumentos.

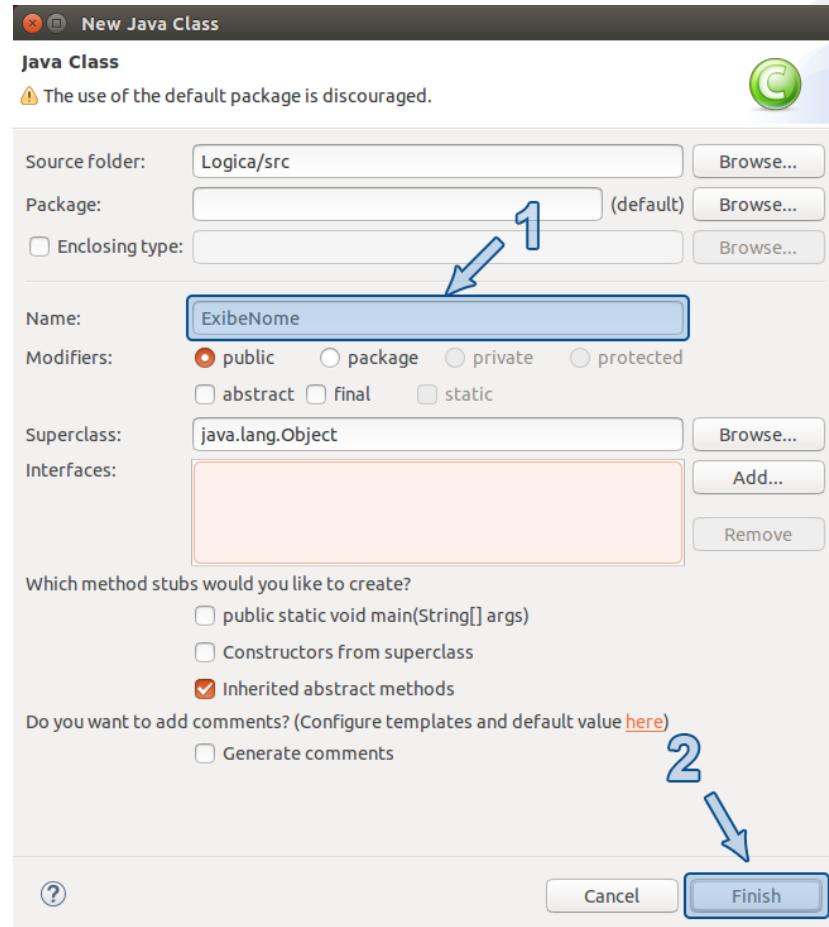
```
1 for(int contador = 0; contador < 100; contador++) {
2     System.out.println("Bom Dia");
3 }
```

Código Java 2.17: Comando for



## Exercícios de Fixação

- 8 Crie um programa que exiba o seu nome no Console 100 vezes. Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e siga a imagem abaixo.



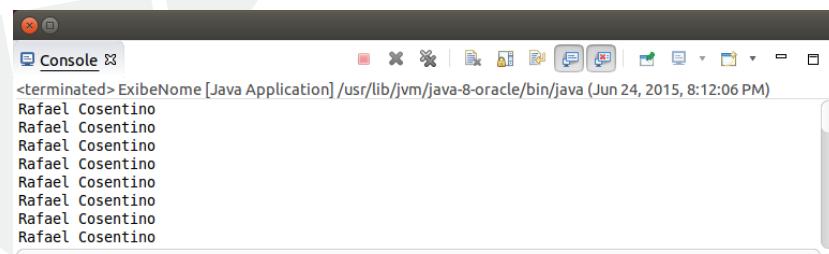
- 9 Altere o código da classe ExibeNome.

```

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

Código Java 2.18: ExibeNome.java

- 10 Salve o arquivo ExibeNome.java. Em seguida, execute a classe ExibeNome. Selecione o arquivo ExibeNome.java e digite "CTRL + F11". Observe o conteúdo exibido na visão Console.



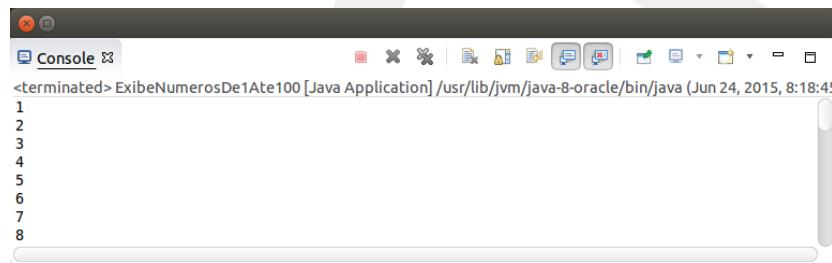
- 11 Crie um programa que exiba os números de 1 a 100 no Console. Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e digite “ExibeNumerosDe1Ate100” no campo que define o nome da nova classe.

- 12 Altere o código da classe ExibeNumerosDe1Ate100.

```
1 public class ExibeNumerosDe1Ate100 {  
2     public static void main(String[] args) {  
3         for(int contador = 1; contador <= 100; contador++) {  
4             System.out.println(contador);  
5         }  
6     }  
7 }
```

Código Java 2.19: ExibeNumerosDe1Ate100.java

- 13 Salve o arquivo ExibeNumerosDe1Ate100.java. Em seguida, execute a classe ExibeNumerosDe1Ate100. Selecione o arquivo ExibeNumerosDe1Ate100.java e digite “CTRL + F11”. Observe o conteúdo exibido na visão Console.



- 14 Faça um programa que percorra todos os números de 1 até 100. Para os números ímpares, deve ser exibido um “\*” e para os números pares “\*\*”. Veja o exemplo abaixo:

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

Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e digite “ExibeAsteriscos” no campo que define o nome da nova classe.

- 15 Altere o código da classe ExibeAsteriscos.

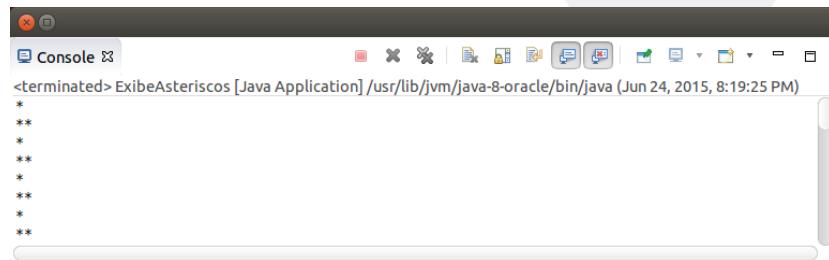
```
1 public class ExibeAsteriscos {  
2     public static void main(String[] args) {  
3         for(int contador = 1; contador <= 100; contador++) {  
4             int resto = contador % 2;  
5             if(resto == 1) {  
6                 System.out.println("**");  
7             } else {  
8                 System.out.println("*");  
9             }  
10        }  
11    }  
12 }
```

```

7     } else {
8         System.out.println("**");
9     }
10    }
11 }
12 }
```

Código Java 2.20: ExibeAsteriscos.java

- 16** Salve o arquivo ExibeAsteriscos.java. Em seguida, execute a classe ExibeAsteriscos. Selecione o arquivo ExibeAsteriscos.java e digite “CTRL + F11”. Observe o conteúdo exibido na visão Console.



- 17** Faça um programa que percorra todos os números de 1 até 100. Para os números múltiplos de 4, exiba a palavra “PIN”, e para os outros, exiba o próprio número. Veja o exemplo abaixo:

```

1
2
3
PIN
5
6
7
PIN
```

Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e digite “PIN” no campo que define o nome da nova classe.

- 18** Altere o código da classe PIN.

```

1 public class PIN {
2     public static void main(String[] args) {
3         for(int contador = 1; contador <= 100; contador++) {
4             int resto = contador % 4;
5             if(resto == 0) {
6                 System.out.println("PI");
7             } else {
8                 System.out.println(contador);
9             }
10        }
11    }
12 }
```

*Código Java 2.21: PIN.java*

- 19 Salve o arquivo PIN.java. Em seguida, execute a classe PIN. Selecione o arquivo PIN.java e digite “CTRL + F11”. Observe o conteúdo exibido na visão Console.

The screenshot shows a Java application window titled "Console". The title bar includes standard icons for close, minimize, maximize, and file operations. Below the title bar, the text "<terminated> PIN [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Jun 24, 2015, 8:19:54 PM)" is displayed. The main area of the window shows the following text:  
1  
2  
3  
PI  
5  
6  
7



## Exercícios Complementares

- 1 Crie um programa que exiba no Console um triângulo de “\*”. Adicione uma classe chamada **Triangulo**. Veja o exemplo abaixo:

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

- 2 Crie um programa que exiba no Console vários triângulos de “\*”. Adicione uma classe chamada **Triangulos**. Veja o exemplo abaixo:

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

- 3 Os números de Fibonacci são uma sequência de números definida recursivamente. O primeiro elemento da sequência 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 exibir os 30 primeiros números da sequência de Fibonacci. Adicione uma classe chamada **Fibonacci**.





# ORIENTAÇÃO A OBJETOS



## Domínio e Aplicação

Um **domínio** é composto pelas entidades, informações e processos relacionados a um determinado contexto. Uma **aplicação** pode ser desenvolvida para automatizar ou tornar factível as tarefas de um domínio. Portanto, uma aplicação é basicamente o “reflexo” de um domínio.

Para exemplificar, suponha que estamos interessados em desenvolver uma aplicação para facilitar as tarefas do cotidiano de um banco. Podemos identificar clientes, funcionários, agências e contas como entidades desse domínio. Assim como podemos identificar as informações e os processos relacionados a essas entidades.



Figura 3.1: Domínio bancário



### Mais Sobre

A identificação dos elementos de um domínio é uma tarefa difícil, pois depende fortemente do conhecimento das entidades, informações e processos que o compõem. Em geral, as pessoas que possuem esse conhecimento ou parte dele estão em contato constante com o domínio e não possuem conhecimentos técnicos para desenvolver uma aplicação.

Desenvolvedores de software buscam constantemente mecanismos para tornar mais eficiente o entendimento dos domínios para os quais eles devem desenvolver aplicações.



## Objetos, Atributos e Métodos

As entidades identificadas no domínio devem ser representadas de alguma forma dentro da aplicação correspondente. Nas aplicações orientadas a objetos, as entidades são representadas por **objetos**.

- Uma aplicação orientada a objetos é composta por objetos.
- Em geral, um objeto representa uma entidade do domínio.

Para exemplificar, suponha que no domínio de um determinado banco exista um cliente chamado João. Dentro de uma aplicação orientada a objetos correspondente a esse domínio, deve existir um objeto para representar esse cliente.

Suponha que algumas informações do cliente João como nome, data de nascimento e sexo são importantes para o banco. Já que esses dados são relevantes para o domínio, o objeto que representa esse cliente deve possuir essas informações. Esses dados são armazenados nos **atributos** do objeto que representa o João.

- Um atributo é uma variável que pertence a um objeto.
- Os dados de um objeto são armazenados nos seus atributos.

O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos. Essas operações são definidas nos **métodos** do objeto.

Os métodos também são utilizados para possibilitar interações entre os objetos de uma aplicação. 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.

- As tarefas que um objeto pode realizar são definidas pelos seus métodos.
- Um objeto é composto por atributos e métodos.

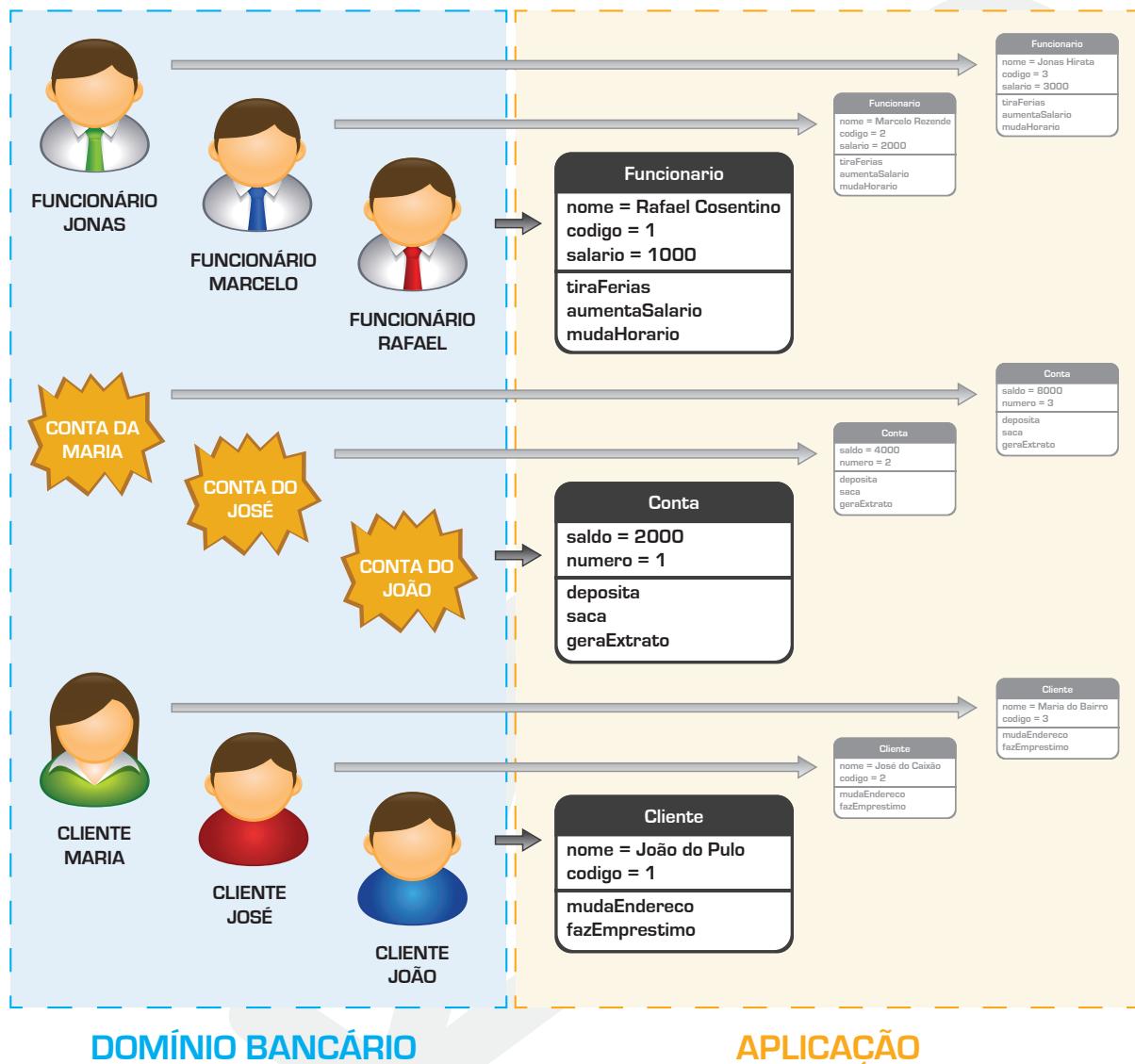


Figura 3.2: Mapeamento Domínio-Aplicação



### Mais Sobre

Em geral, não é adequado utilizar o objeto que representa um determinado cliente para representar outro cliente do banco, pois os dados dos clientes podem ser diferentes. Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.



### Mais Sobre

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.



## Classes

Antes de um objeto ser criado, devemos definir quais serão os seus atributos e métodos. Essa definição é realizada através de uma **classe** elaborada por um programador. A partir de uma classe, podemos construir objetos na memória do computador que executa a nossa aplicação.

Podemos representar uma classe através de diagramas **UML**. O diagrama UML de uma classe é composto pelo nome da mesma e pelos atributos e métodos que ela define. Todos os objetos criados a partir da classe Conta terão os atributos e métodos mostrados no diagrama UML. Os valores dos atributos de dois objetos criados a partir da classe Conta podem ser diferentes.

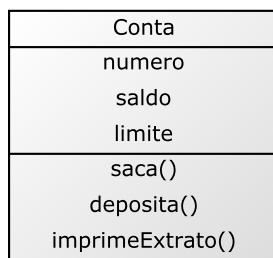


Figura 3.3: Diagrama UML da classe Conta.



### Analogia

Um objeto é como se fosse uma casa ou um prédio. Para ser construído, precisa de um espaço físico. No caso dos objetos, esse espaço físico é algum trecho vago da memória do computador que executa a aplicação. No caso das casas e dos prédios, o espaço físico é algum terreno vazio.

Um prédio é construído a partir de uma planta criada por um engenheiro ou arquiteto. Para criar um objeto, é necessário algo semelhante a uma planta para que sejam “desenhados” os atributos e métodos que o objeto deve ter. Em orientação a objetos, a “planta” de um objeto é o que chamamos de classe.

Uma classe funciona como uma “receita” para criar objetos. Inclusive, vários objetos podem ser criados a partir de uma única classe. Assim como várias casas ou prédios poderiam ser construídos a partir de uma única planta; ou vários bolos poderiam ser preparados a partir de uma única receita; ou vários carros poderiam ser construídos a partir de um único projeto.

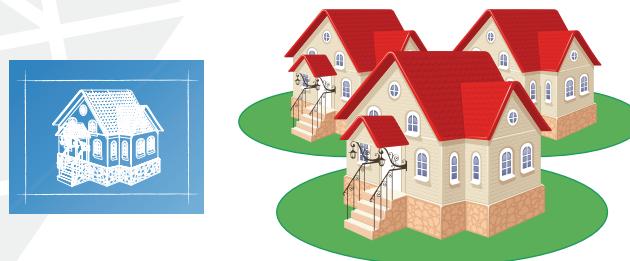


Figura 3.4: Diversas casas construídas a partir da mesma planta



Figura 3.5: Diversos batos preparados a partir da mesma receita



Figura 3.6: Diversos carros construídos a partir do mesmo projeto

Basicamente, as diferenças entre dois objetos criados a partir da classe Conta são os valores dos seus atributos. Assim como duas casas construídas a partir da mesma planta podem possuir características diferentes. Por exemplo, a cor das paredes.



Figura 3.7: Diversas casas com características diferentes

## Classes em 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     public double saldo;
3     public double limite;
4     public int numero;
5 }
```

Código Java 3.1: Conta.java

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

esticamente tipada, os tipos dos atributos são definidos no código. Os atributos 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. O modificador public é adicionado em cada atributo para que eles possam ser acessados a partir de qualquer ponto do código. Discutiremos sobre esse e outros modificadores de visibilidade em capítulos posteriores.



### Importante

Por convenção, os nomes das classes na linguagem Java devem seguir o padrão “Pascal Case”.

## 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 e 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         // criando um objeto
4         new Conta();
5     }
6 }
```

*Código Java 3.2: TestaConta.java*

A linha com o comando new poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe Conta. A classe TestaConta serve apenas para colocarmos o método main, que é o ponto de partida da aplicação.

```

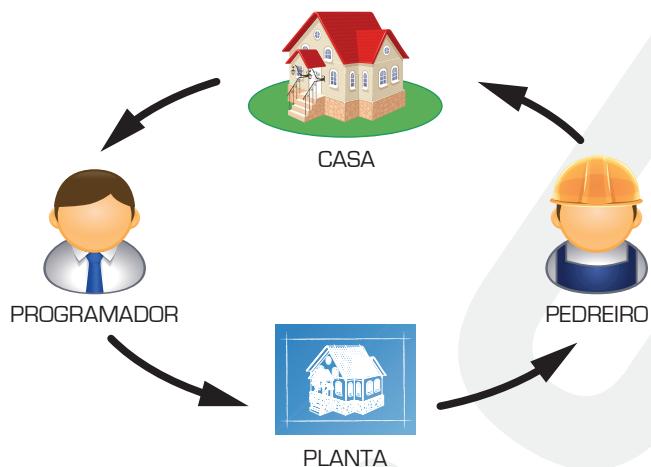
1 class TestaConta {
2     public static void main(String[] args) {
3         // criando três objetos
4         new Conta();
5         new Conta();
6         new Conta();
7     }
8 }
```

*Código Java 3.3: TestaConta.java*



### Analogia

Chamar o comando new passando uma classe Java é como se estivéssemos contratando uma construtora passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta. Assim como a máquina virtual se encarrega de construir o objeto na memória do computador.



*Figura 3.8: Construindo casas*



## Referências

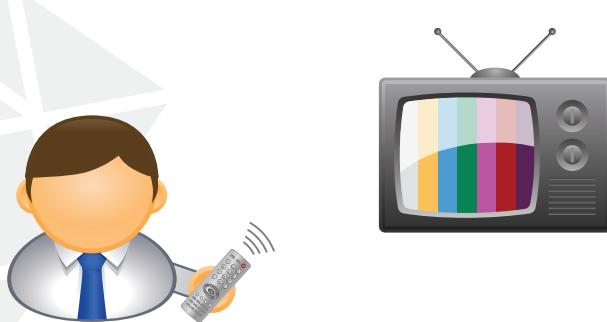
Todo objeto possui uma referência. A referência de um objeto é a única maneira de acessar os seus atributos e métodos. Dessa forma, devemos guardar as referências dos objetos que desejamos utilizar.



### Analogia

A princípio, podemos comparar a referência de um objeto com o endereço de memória desse objeto. De fato, essa comparação simplifica o aprendizado. Contudo, o conceito de referência é mais amplo. Uma referência é o elemento que permite que um determinado objeto seja acessado.

Uma referência está para um objeto assim como um controle remoto está para um aparelho de TV. Através do controle remoto de uma TV você pode aumentar o volume ou trocar de canal. Analogicamente, podemos controlar um objeto através da referência do mesmo.



*Figura 3.9: Controle remoto*

## Referências em Java

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

Para guardar as referências devolvidas pelo comando `new`, devemos utilizar variáveis não primitivas.

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

*Código Java 3.4: Criando um objeto e guardando a referência.*

No código Java acima, a variável **referencia** receberá a referência do objeto criado pelo comando `new`. Essa variável é do tipo `Conta`. Isso significa que ela só pode armazenar referências de objetos do tipo `Conta`.



## Manipulando Atributos

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

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

*Código Java 3.5: Alterando e acessando os atributos de um objeto.*

No código acima, o atributo `saldo` 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 Padrão

Poderíamos instanciar um objeto e utilizar seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores padrão. Os atributos de tipos numéricos são inicializados com `0`, os atributos do tipo `boolean` são inicializados com `false` e os demais atributos com `null` (referência vazia).

```
1 class Conta {
2     double limite;
3 }
```

*Código Java 3.6: Conta.java*

```

1 class TestaConta {
2     public static void main(String[] args) {
3         Conta conta = new Conta();
4
5         // imprime 0
6         System.out.println(conta.limite);
7     }
8 }
```

*Código Java 3.7: TestaConta.java*

A inicialização dos atributos com os valores padrão ocorre na instanciação, ou seja, quando o comando `new` é utilizado. Dessa forma, todo objeto “nasce” com os valores padrão. Em alguns casos, é necessário trocar esses valores. Para trocar o valor padrão de um atributo, devemos inicializá-lo na declaração. Por exemplo, suponha que o limite padrão das contas de um banco seja R\$ 500. Nesse caso, seria interessante definir esse valor como padrão para o atributo `limite`.

```

1 class Conta {
2     public double limite = 500;
3 }
```

*Código Java 3.8: Conta.java*

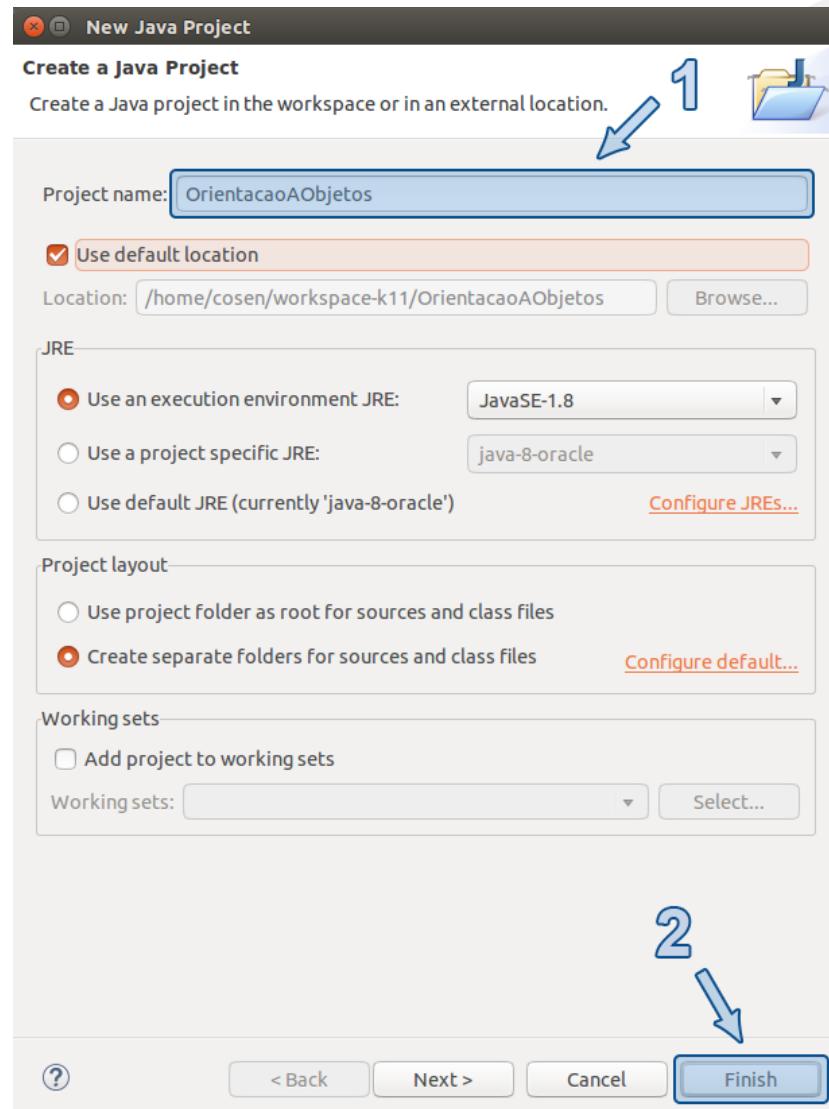
```

1 class TestaConta {
2     public static void main(String[] args) {
3         Conta conta = new Conta();
4
5         // imprime 500
6         System.out.println(conta.limite);
7     }
8 }
```

*Código Java 3.9: TestaConta.java*

## Exercícios de Fixação

- Para não confundir, feche o projeto **Logica** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2** Implemente uma classe para definir os objetos que representarão os clientes de um banco. Essa classe deve possuir dois atributos: um para armazenar os nomes e outro para armazenar os códigos dos clientes. Digite “CTRL + 3” e pesquise por “Create a Java class”. Selecione a opção correspondente e digite “Cliente” no campo que define o nome da nova classe. Adicione os atributos solicitados nessa classe.

```

1 public class Cliente {
2     public String nome;
3     public int codigo;
4 }
```

Código Java 3.10: Cliente.java

- 3** Faça um teste criando dois objetos da classe Cliente. Altere e exiba no Console os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaCliente** com o código abaixo.



### Lembre-se

Para não digitar todo o código, utilize os templates do método `main` e do método `println`. Para utilizar o template do método `main`, digite “`main`” e em seguida “`CTRL + ESPAÇO`”. Analogamente, digite “`sysout`” e em seguida “`CTRL + ESPAÇO`” para utilizar o template do método `println`.

```

1 public class TestaCliente {
2     public static void main(String[] args) {
3         Cliente c1 = new Cliente();
4         c1.nome = "Rafael Cosentino";
5         c1.codigo = 1;
6
7         Cliente c2 = new Cliente();
8         c2.nome = "Jonas Hirata";
9         c2.codigo = 2;
10
11        System.out.println("Dados do primeiro cliente");
12        System.out.println("Nome: " + c1.nome);
13        System.out.println("Código: " + c1.codigo);
14
15        System.out.println("-----");
16
17        System.out.println("Dados do segundo cliente");
18        System.out.println("Nome: " + c2.nome);
19        System.out.println("Código: " + c2.codigo);
20    }
21 }
```

Código Java 3.11: `TestaCliente.java`

Execute a classe `TestaCliente`. Selecione o arquivo `TestaCliente.java` e digite “`CTRL + F11`”.

- 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 chamada **CartaoDeCredito** para modelar os objetos que representarão os cartões de crédito.

```

1 public class CartaoDeCredito {
2     public int numero;
3     public String dataDeValidade;
4 }
```

Código Java 3.12: `CartaoDeCredito.java`

- 5** Faça um teste criando dois objetos da classe `CartaoDeCredito`. Altere e exiba no Console os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaCartaoDeCredito** com o código abaixo.

```

1 public class TestaCartaoDeCredito {
2     public static void main(String[] args) {
3         CartaoDeCredito cdc1 = new CartaoDeCredito();
4         cdc1.numero = 111111;
5         cdc1.dataDeValidade = "01/01/2013";
6
7         CartaoDeCredito cdc2 = new CartaoDeCredito();
8         cdc2.numero = 222222;
```

```

9  cdc2.dataDeValidade = "01/01/2014";
10 System.out.println("Dados do primeiro cartão");
11 System.out.println("Número: " + cdc1.numero);
12 System.out.println("Data de validade: " + cdc1.dataDeValidade);
13
14 System.out.println("-----");
15 System.out.println("Dados do segundo cartão");
16 System.out.println("Número: " + cdc2.numero);
17 System.out.println("Data de validade: " + cdc2.dataDeValidade);
18 }
19 }
20 }
21 }
```

Código Java 3.13: TestaCartaoDeCredito.java

Execute a classe TestaCartaoDeCredito.

- 6** As agências do banco possuem número. Crie uma classe chamada **Agencia** para definir os objetos que representarão as agências do banco.

```

1 public class Agencia {
2     public int numero;
3 }
```

Código Java 3.14: Agencia.java

- 7** Faça um teste criando dois objetos da classe Agencia. Altere e exiba no Console os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaAgencia** com o código abaixo.

```

1 public class TestaAgencia {
2     public static void main(String[] args) {
3         Agencia a1 = new Agencia();
4         a1.numero = 1234;
5
6         Agencia a2 = new Agencia();
7         a2.numero = 5678;
8
9         System.out.println("Dados da primeira agência");
10        System.out.println("Número: " + a1.numero);
11
12        System.out.println("-----");
13
14        System.out.println("Dados da segunda agência");
15        System.out.println("Número: " + a2.numero);
16    }
17 }
```

Código Java 3.15: TestaAgencia.java

Execute a classe TestaAgencia.

- 8** As contas do banco possuem número, saldo e limite. Crie uma classe chamada **Conta** para definir os objetos que representarão as contas do banco.

```

1 public class Conta {
2     public int numero;
3     public double saldo;
```

```

4  public double limite;
5 }
```

*Código Java 3.16: Conta.java*

- 9 Faça um teste criando dois objetos da classe Conta. Altere e exiba no Console os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaConta** com o código abaixo.

```

1 public class TestaConta {
2     public static void main(String[] args) {
3         Conta c1 = new Conta();
4         c1.numero = 1234;
5         c1.saldo = 1000;
6         c1.limite = 500;
7
8         Conta c2 = new Conta();
9         c2.numero = 5678;
10        c2.saldo = 2000;
11        c2.limite = 250;
12
13        System.out.println("Dados da primeira conta");
14        System.out.println("Número: " + c1.numero);
15        System.out.println("Saldo: " + c1.saldo);
16        System.out.println("Limite: " + c1.limite);
17
18        System.out.println("-----");
19
20        System.out.println("Dados da segunda conta");
21        System.out.println("Número: " + c2.numero);
22        System.out.println("Saldo: " + c2.saldo);
23        System.out.println("Limite: " + c2.limite);
24    }
25 }
```

*Código Java 3.17: TestaConta.java*

Execute a classe TestaConta.

- 10 Faça um teste que exiba no Console os valores armazenados nos atributos de um objeto da classe Conta logo após a sua criação. Crie uma nova classe chamada **TestaValoresPadrao** com o código abaixo.

```

1 public class TestaValoresPadrao {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         System.out.println("Valores Padrão");
6         System.out.println("Número: " + c.numero);
7         System.out.println("Saldo: " + c.saldo);
8         System.out.println("Limite: " + c.limite);
9     }
10 }
```

*Código Java 3.18: TestaValoresPadrao.java*

Execute a classe TestaValoresPadrao.

- 11 Altere a classe Conta para que todos os objetos criados a partir dessa classe possuam R\$ 100 de

limite inicial.

```
1 public class Conta {  
2     public int numero;  
3     public double saldo;  
4     public double limite = 100;  
5 }
```

Código Java 3.19: Conta.java

Execute a classe TestaValoresPadrao.



## Exercícios Complementares

- 1 Crie uma classe chamada **Aluno** para definir os objetos que representarão os alunos de uma escola. Nessa classe, declare três atributos: o primeiro para o nome, o segundo para o RG e o terceiro para a data de nascimento dos alunos.
- 2 Faça uma classe chamada **TestaAluno**. Crie dois objetos da classe Aluno. Altere os valores dos atributos desses objetos e exiba no Console os valores armazenados nesses atributos.
- 3 Em uma escola, além dos alunos temos os funcionários, que também precisam ser representados em nossa aplicação. Então crie uma classe chamada **Funcionario** que contenha dois atributos: o primeiro para o nome e o segundo para o salário dos funcionários.
- 4 Faça uma classe chamada **TestaFuncionario**. Crie dois objetos da classe Funcionario. Altere os valores dos atributos desses objetos e exiba no Console os valores armazenados nesses atributos.
- 5 Em uma escola, os alunos precisam ser divididos por turmas, que devem ser representadas dentro da aplicação. Crie uma classe chamada **Turma** que contenha quatro atributos: o primeiro para o período, o segundo para definir a série, o terceiro para sigla e o quarto para o tipo de ensino.
- 6 Faça uma classe chamada **TestaTurma**. Crie dois objetos da classe Turma. Altere os valores dos atributos desses objetos e exiba no Console os valores armazenados nesses atributos.



## Relacionamentos: Associação, Agregação e Composição

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.

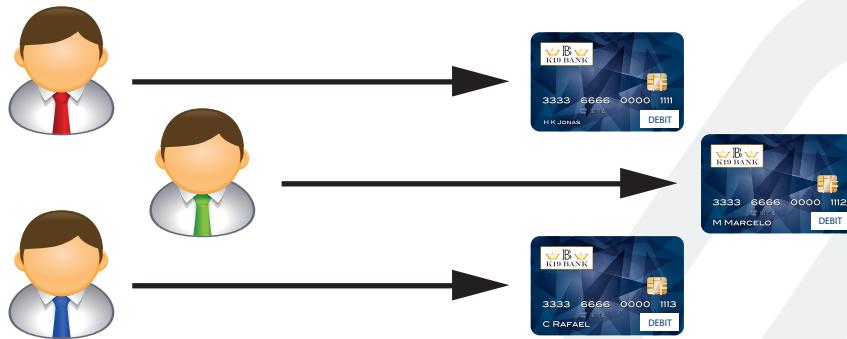


Figura 3.10: Clientes e cartões

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

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

Código Java 3.26: Cliente.java

```
1 class CartaoDeCredito {
2     public int numero;
3     public String dataDeValidade;
4     public Cliente cliente;
5 }
```

Código Java 3.27: CartaoDeCredito.java

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

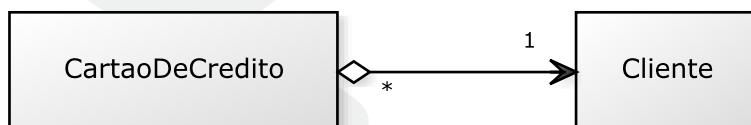


Figura 3.11: Agregação entre clientes e cartões de crédito.

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 número 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 caractere "\*" 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, indiretamente, 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 // Ligando os objetos
5 cdc.cliente = c;
6
7 // Acessando o nome do cliente
8 cdc.cliente.nome = "Rafael Cosentino";
9

```

Código Java 3.28: Concretizando uma agregação

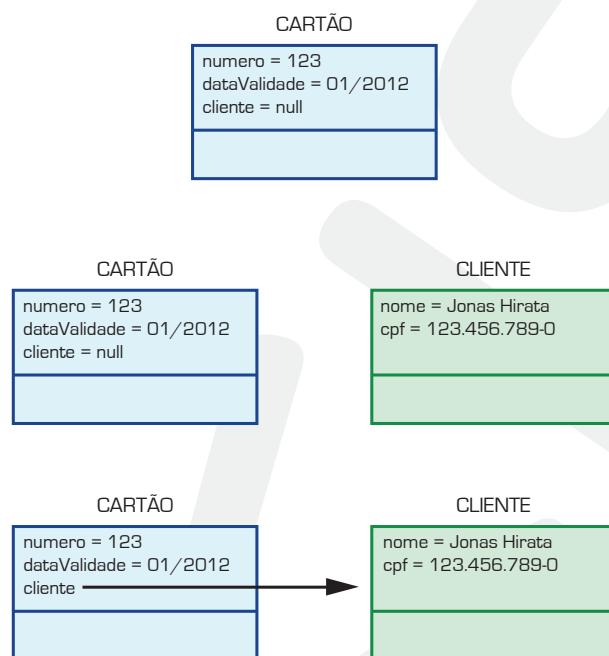


Figura 3.12: Conectando um cliente e um cartão



## Exercícios de Fixação

- 12** Defina um vínculo entre os objetos que representam os clientes e os objetos que representam os cartões de crédito. Para isso, você deve **alterar** a classe CartaoDeCredito.

```

1 public class CartaoDeCredito {
2     public int numero;
3     public String dataDeValidade;
4     public Cliente cliente;
5 }

```

Código Java 3.29: CartaoDeCredito.java

- 13** Teste o relacionamento entre clientes e cartões de crédito. Crie uma nova classe chamada **TestaClienteECartao** com o código abaixo.

```

1 public class TestaClienteECartao {
2     public static void main(String[] args) {
3         Cliente c = new Cliente();

```

```

4   CartaoDeCredito cdc = new CartaoDeCredito();
5
6   c.nome = "Rafael Cosentino";
7   c.codigo = 123;
8
9   cdc.numero = 111111;
10  cdc.dataDeValidade = "12/12/18";
11
12  System.out.println("Dados do cliente");
13  System.out.println("Nome: " + c.nome);
14  System.out.println("Código: " + c.codigo);
15
16  System.out.println("-----");
17
18  System.out.println("Dados do cartão");
19  System.out.println("Número: " + cdc.numero);
20  System.out.println("Data de validade: " + cdc.dataDeValidade);
21
22  System.out.println("-----");
23
24  cdc.cliente = c;
25
26  System.out.println("Dados do cliente obtidos através do cartão");
27  System.out.println(cdc.cliente.nome);
28  System.out.println(cdc.cliente.codigo);
29 }
30 }
```

Código Java 3.30: TestaClienteECartao.java

Execute a classe TestaClienteECartao.

- 14** Defina um vínculo entre os objetos que representam as agências e os objetos que representam os contas. Para isso, você deve **alterar** a classe Conta.

```

1 public class Conta {
2   public int numero;
3   public double saldo;
4   public double limite = 100;
5   public Agencia agencia;
6 }
```

Código Java 3.31: Conta.java

- 15** Teste o relacionamento entre contas e agências. Crie uma nova classe chamada **TestaContaEA-****gencia** com o código abaixo.

```

1 public class TestaContaEAgendencia {
2   public static void main(String[] args) {
3     Agencia a = new Agencia();
4     Conta c = new Conta();
5
6     a.numero = 178;
7
8     c.numero = 123;
9     c.saldo = 1000.0;
10    c.limite = 500;
11
12    System.out.println("Dados da agência");
13    System.out.println("Número: " + a.numero);
14
15    System.out.println("-----");
```

```

17 System.out.println("Dados da conta");
18 System.out.println("Número: " + c.numero);
19 System.out.println("Saldo: " + c.saldo);
20 System.out.println("Limite: " + c.limite);
21
22 System.out.println("-----");
23
24 c.agencia = a;
25
26 System.out.println("Dados do agência obtidos através da conta");
27 System.out.println(c.agencia.numero);
28 }
29 }
```

*Código Java 3.32: TestaContaEAgencia.java*

Execute a classe TestaContaEAgencia.



## Exercícios Complementares

- 7 Defina um vínculo entre os alunos e as turmas, criando na classe Aluno um atributo do tipo Turma.
- 8 Teste o relacionamento entre os alunos e as turmas, criando um objeto de cada classe e alterando os valores dos atributos desses objetos. Exiba na Console os valores que estão nos atributos da turma através do aluno. Crie uma nova classe chamada **TesteAlunoTurma** para implementar esse teste.



## Métodos

No banco, é possível realizar diversas operações em uma conta: depósito, saque, transferência, consultas e etc. Essas operações podem modificar ou apenas acessar os valores dos atributos dos objetos que representam as contas.

Essas operações são realizadas em **métodos** definidos na própria classe Conta. Por exemplo, para realizar a operação de depósito, podemos acrescentar o seguinte método na classe Conta.

```

1 void deposita(double valor) {
2     // implementação
3 }
```

*Código Java 3.35: Definindo um método*

Podemos dividir um método em quatro partes:

**Nome:** É utilizado para chamar o método. Na linguagem Java, é uma boa prática definir os nomes dos métodos utilizando a convenção “Camel Case” com a primeira letra minúscula.

**Lista de Parâmetros:** Define os valores que o método deve receber. Métodos que não devem receber nenhum valor possuem a lista de parâmetros vazia.

**Corpo:** Define o que acontecerá quando o método for chamado.

**Retorno:** A resposta que será devolvida ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado com a palavra reservada **void**.

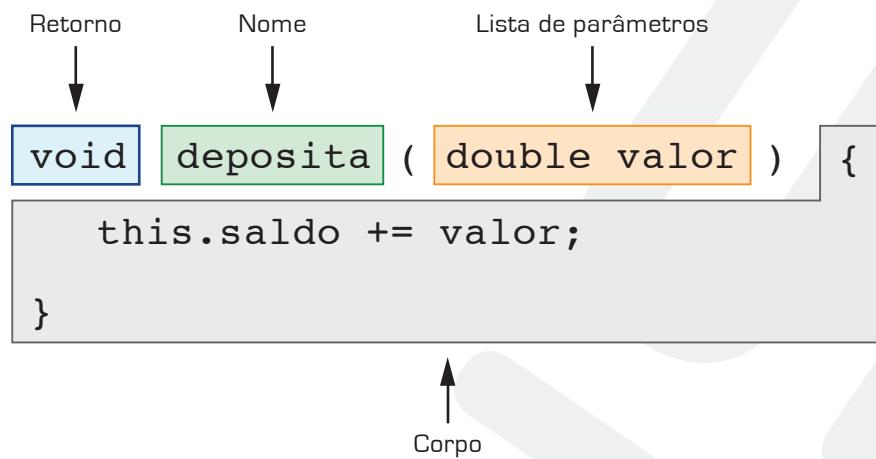


Figura 3.13: Estrutura de um método

Para realizar um depósito, devemos chamar o método `deposita()` através da referência do objeto que representa a conta que terá o dinheiro creditado.

```

1 // Referência de um objeto
2 Conta c = new Conta();
3
4 // Chamando o método deposita()
5 c.deposita(1000);
  
```

Código Java 3.36: Chamando o método `deposita()`

Normalmente, os métodos acessam ou alteram os valores armazenados nos atributos dos objetos. Por exemplo, na execução do método `deposita()`, é necessário alterar o valor do atributo `saldo` do objeto que foi escolhido para realizar a operação.

Dentro de um método, para acessar os atributos do objeto que está processando o método, devemos utilizar a palavra reservada **this**.

```

1 void deposita(double valor) {
2     this.saldo += valor;
3 }
  
```

Código Java 3.37: Utilizando o `this` para acessar e/ou modificar um atributo

O método `deposita()` não possui nenhum retorno lógico. Por isso, foi marcado com `void`. Mas, para outros métodos, pode ser necessário definir um tipo de retorno específico.

Considere, por exemplo, um método para realizar a operação que consulta o saldo disponível das contas. Suponha também que o saldo disponível é igual a soma do saldo e do limite. Então, esse método deve somar os atributos `saldo` e `limite` e devolver o resultado. Por outro lado, esse método não deve receber nenhum valor, pois todas as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas.

```

1 double consultaSaldoDisponivel() {
2     return this.saldo + this.limite;
3 }
```

Código Java 3.38: Método com retorno double

Ao chamar o método `consultaSaldoDisponivel()` a resposta pode ser armazenada em uma variável do tipo `double`.

```

1 Conta c = new Conta();
2 c.deposita(1000);
3
4 // Armazenando a resposta de um método em uma variável
5 double saldoDisponivel = c.consultaSaldoDisponivel();
6
7 System.out.println("Saldo Disponível: " + this.saldoDisponivel);
```

Código Java 3.39: Armazenando a resposta de um método



## Exercícios de Fixação

- 16** **Acrescente** alguns métodos na classe `Conta` para realizar as operações de deposito, saque, impressão de extrato e consulta do saldo disponível.

```

1 public class Conta {
2     public int numero;
3     public double saldo;
4     public double limite = 100;
5     public Agencia agencia;
6
7     // ADICIONE OS MÉTODOS ABAIXO
8     public void deposita(double valor) {
9         this.saldo += valor;
10    }
11
12    public void saca(double valor) {
13        this.saldo -= valor;
14    }
15
16    public void imprimeExtrato() {
17        System.out.println("SALDO: " + this.saldo);
18    }
19
20    public double consultaSaldoDisponivel() {
21        return this.saldo + this.limite;
22    }
23 }
```

Código Java 3.40: Conta.java

- 17** Teste os métodos da classe `Conta`. Crie uma nova classe chamada **TestaMetodosConta** com o código abaixo.

```

1 public class TestaMetodosConta {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         System.out.println("Chamando o método deposita passando o valor 1000");
```

```

6   c.deposita(1000);
7   c.imprimeExtrato();
8
9   System.out.println("-----");
10
11  System.out.println("Chamando o método saca passando o valor 100");
12  c.saca(100);
13  c.imprimeExtrato();
14
15  System.out.println("-----");
16
17  double saldoDisponivel = c.consultaSaldoDisponivel();
18  System.out.println("SALDO DISPONÍVEL: " + saldoDisponivel);
19
20 }
}

```

Código Java 3.41: TestaMetodosConta.java

Execute a classe TestaMetodosConta.



## Exercícios Complementares

- 9 Adicione na classe **Funcionario** dois métodos: um para aumentar o salário e outro para consultar os dados dos funcionários.
- 10 Teste os métodos da classe Funcionario. Crie uma nova classe chamada **TestaMetodosFuncionario** para implementar esse teste.



## Sobrecarga (Overloading)

Os clientes dos bancos costumam consultar periodicamente informações relativas às suas contas. Geralmente, essas informações são obtidas através de extratos. No sistema do banco, os extratos podem ser gerados por métodos da classe Conta.

```

1 class Conta {
2     public double saldo;
3     public double limite;
4
5     public void imprimeExtrato(int dias){
6         // extrato
7     }
8 }

```

Código Java 3.44: Conta.java

O método `imprimeExtrato()` recebe a quantidade de dias que deve ser considerada para gerar o extrato da conta. Por exemplo, se esse método receber o valor 30 então ele deve gerar um extrato com as movimentações dos últimos 30 dias.

Em geral, extratos dos últimos 15 dias atendem as necessidades dos clientes. Dessa forma, poderíamos acrescentar um método na classe Conta para gerar extratos com essa quantidade fixa de dias.

```

1 class Conta {
2     public double saldo;
3     public double limite;
4
5     public void imprimeExtrato(){
6         // extrato dos últimos 15 dias
7     }
8
9     public void imprimeExtrato(int dias){
10        // extrato
11    }
12 }
```

Código Java 3.45: Conta.java

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão definida pelo banco para gerar os extratos (15 dias).

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.

```

1 class Conta {
2
3     public void imprimeExtrato(){
4         this.imprimeExtrato(15);
5     }
6
7     public void imprimeExtrato(int dias){
8         // extrato
9     }
10 }
```

Código Java 3.46: Conta.java



## Exercícios de Fixação

- 18** Crie uma classe chamada **Gerente** para definir os objetos que representarão os gerentes do banco. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

```

1 public class Gerente {
2     public String nome;
3     public double salario;
4
5     public void aumentaSalario() {
6         this.aumentaSalario(0.1);
7     }
8 }
```

```

9  public void aumentaSalario(double taxa) {
10    this.salario += this.salario * taxa;
11  }
12 }
```

*Código Java 3.47: Gerente.java*

- 19 Teste os métodos de aumento salarial definidos na classe Gerente. Crie uma nova classe chamada **TestaGerente** com o código abaixo.

```

1 public class TestaGerente {
2   public static void main(String[] args){
3     Gerente g = new Gerente();
4     g.salario = 1000;
5
6     System.out.println("Salário: " + g.salario);
7
8     System.out.println("Aumentando o salário em 10% ");
9     g.aumentaSalario();
10
11    System.out.println("Salário: " + g.salario);
12
13    System.out.println("Aumentando o salário em 30% ");
14    g.aumentaSalario(0.3);
15
16    System.out.println("Salário: " + g.salario);
17  }
18 }
```

*Código Java 3.48: TestaGerente.java*

Execute a classe TestaGerente.



## Construtores

No domínio de um banco, todo cartão de crédito deve possuir um número. Toda agência deve possuir um número. Toda conta deve estar 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;
```

*Código Java 3.49: Definindo um número para um cartão de crédito*

```

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

*Código Java 3.50: Definindo um número para uma agência*

```

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

*Código Java 3.51: Definindo uma agência para uma conta*

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 não correr esse risco, podemos utilizar **construtores**. Um construtor permite que um determinado trecho de código seja executado toda vez que um objeto é criado, ou seja, toda vez que o operador new é chamado. Assim como os métodos, os construtores podem receber parâmetros. Contudo, diferentemente dos métodos, os construtores não devolvem resposta.

Em Java, um construtor deve ter o mesmo nome da classe na qual ele foi definido.

```

1 class CartaoDeCredito {
2     public int numero;
3
4     public CartaoDeCredito(int numero) {
5         this.numero = numero;
6     }
7 }
```

Código Java 3.52: *CartaoDeCredito.java*

```

1 class Agencia {
2     public int numero;
3
4     public Agencia(int numero) {
5         this.numero = numero;
6     }
7 }
```

Código Java 3.53: *Agencia.java*

```

1 class Conta {
2     public Agencia agencia;
3
4     public Conta(Agencia agencia) {
5         this.agencia = agencia;
6     }
7 }
```

Código Java 3.54: *Conta.java*

Na criação de um objeto com o comando new, os argumentos passados devem ser compatíveis com a lista de parâmetros 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.

```

1 // Passando corretamente os parâmetros para os construtores
2 CartaoDeCredito cdc = new CartaoDeCredito(1111);
3
4 Agencia a = new Agencia(1234);
5
6 Conta c = new Conta(a);
```

Código Java 3.55: Construtores

```

1 // ERRO DE COMPILAÇÃO
2 CartaoDeCredito cdc = new CartaoDeCredito();
3
4 // ERRO DE COMPILAÇÃO
5 Agencia a = new Agencia();
```

```

6 // ERRO DE COMPILAÇÃO
7 Conta c = new Conta();
8

```

*Código Java 3.56: Construtores*

## Construtor Padrão

Toda vez que um objeto é criado, um construtor da classe correspondente deve ser chamado. Mesmo quando nenhum construtor for definido explicitamente, há um construtor padrão que será inserido pelo próprio compilador. O construtor padrão não recebe parâmetros e será inserido sempre que o desenvolvedor não definir pelo menos um construtor explicitamente.

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 class Conta {
2
3 }

```

*Código Java 3.57: Conta.java*

```

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

```

*Código Java 3.58: Utilizando o construtor padrão*

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

```

1 class Agencia {
2     public int numero;
3
4     public Agencia(int numero) {
5         this.numero = numero;
6     }
7 }

```

*Código Java 3.59: Agencia.java*

```

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

```

*Código Java 3.60: Chamando um construtor sem argumentos*

## Sobrecarga de Construtores

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

```

1 class Pessoa {
2     public String rg;
3     public int cpf;
4
5     public Pessoa(String rg){
6         this.rg = rg;
7     }
8 }

```

```

7 }
8
9 public Pessoa(int cpf){
10     this.cpf = cpf;
11 }
12 }
```

Código Java 3.61: Pessoa.java

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

```

1 // Chamando o primeiro construtor
2 Pessoa p1 = new Pessoa("123456X");
3
4 // Chamando o segundo construtor
5 Pessoa p2 = new Pessoa(123456789);
```

Código Java 3.62: Utilizando dois construtores diferentes

## Construtores chamando Construtores

Assim como podemos encadear métodos, também podemos encadear construtores.

```

1 class Conta {
2     public int numero;
3     public double limite;
4
5     public Conta(int numero) {
6         this.numero = numero;
7     }
8
9     public Conta(int numero, double limite) {
10        this(numero);
11        this.limite = limite;
12    }
13 }
```

Código Java 3.63: Conta.java



## Exercícios de Fixação

- 20** Acrescente um construtor na classe Agencia para receber um número como parâmetro.

```

1 public class Agencia {
2     public int numero;
3
4     public Agencia(int numero) {
5         this.numero = numero;
6     }
7 }
```

Código Java 3.64: Agencia.java

- 21** Verifique as classes TestaAgencia e TestaContaEAgencia. Observe os erros de compilação.

- 22** Altere o código das classes TestaAgencia e TestaContaEAgencia para que os erros de compilação sejam resolvidos.

```

1 public class TestaAgencia {
2     public static void main(String[] args) {
3         Agencia a1 = new Agencia(1234);
4
5         Agencia a2 = new Agencia(5678);
6
7         System.out.println("Dados da primeira agência");
8         System.out.println("Número: " + a1.numero);
9
10        System.out.println("-----");
11
12        System.out.println("Dados da segunda agência");
13        System.out.println("Número: " + a2.numero);
14    }
15 }
```

Código Java 3.65: TestaAgencia.java

```

1 public class TestaContaEAgencia {
2     public static void main(String[] args) {
3         Agencia a = new Agencia(1234);
4         Conta c = new Conta();
5
6         c.numero = 123;
7         c.saldo = 1000.0;
8         c.limite = 500;
9
10        System.out.println("Dados da agência");
11        System.out.println("Número: " + a.numero);
12
13        System.out.println("-----");
14
15        System.out.println("Dados da conta");
16        System.out.println("Número: " + c.numero);
17        System.out.println("Saldo: " + c.saldo);
18        System.out.println("Limite: " + c.limite);
19
20        System.out.println("-----");
21
22        c.agencia = a;
23
24        System.out.println("Dados do agência obtidos através da conta");
25        System.out.println(c.agencia.numero);
26    }
27 }
```

Código Java 3.66: TestaContaEAgencia.java

Execute novamente as classes TestaAgencia e TestaContaEAgencia.

- 23** Acrescente um construtor na classe CartaoDeCredito para receber um número como parâmetro.

```

1 public class CartaoDeCredito {
2     public int numero;
3     public String dataDeValidade;
4
5     public Cliente cliente;
6
7     public CartaoDeCredito(int numero) {
8         this.numero = numero;
9     }
10 }
```

```

9 }
10 }
```

*Código Java 3.67: CartaoDeCredito.java*

**24** Verifique as classes TestaCartaoDeCredito e TestaClienteECartao. Observe os erros de compilação.

**25** Altere o código das classes TestaCartaoDeCredito e TestaClienteECartao para que os erros de compilação sejam resolvidos.

```

1 public class TestaCartaoDeCredito {
2     public static void main(String[] args) {
3         CartaoDeCredito cdc1 = new CartaoDeCredito(111111);
4         cdc1.dataDeValidade = "01/01/2013";
5
6         CartaoDeCredito cdc2 = new CartaoDeCredito(222222);
7         cdc2.dataDeValidade = "01/01/2014";
8
9         System.out.println("Dados do primeiro cartão");
10        System.out.println("Número: " + cdc1.numero);
11        System.out.println("Data de validade: " + cdc1.dataDeValidade);
12
13        System.out.println("-----");
14
15        System.out.println("Dados do segundo cartão");
16        System.out.println("Número: " + cdc2.numero);
17        System.out.println("Data de validade: " + cdc2.dataDeValidade);
18    }
19 }
```

*Código Java 3.68: TestaCartaoDeCredito.java*

```

1 public class TestaClienteECartao {
2     public static void main(String[] args) {
3         Cliente c = new Cliente();
4         CartaoDeCredito cdc = new CartaoDeCredito(111111);
5
6         c.nome = "Rafael Cosentino";
7         c.codigo = 123;
8
9         cdc.dataDeValidade = "12/12/18";
10
11        System.out.println("Dados do cliente");
12        System.out.println("Nome: " + c.nome);
13        System.out.println("Código: " + c.codigo);
14
15        System.out.println("-----");
16
17        System.out.println("Dados do cartão");
18        System.out.println("Número: " + cdc.numero);
19        System.out.println("Data de validade: " + cdc.dataDeValidade);
20
21        System.out.println("-----");
22
23        cdc.cliente = c;
24
25        System.out.println("Dados do cliente obtidos através do cartão");
26        System.out.println(cdc.cliente.nome);
27        System.out.println(cdc.cliente.codigo);
28    }
29 }
```

*Código Java 3.69: TestaClienteECartao.java*

Execute novamente as classes TestaCartaoDeCredito e TestaClienteECartao.

- 26** Acrescente um construtor na classe Conta para receber uma referência de um objeto do tipo Agencia como parâmetro.

```

1 class Conta {
2     public int numero;
3     public double saldo;
4     public double limite = 100;
5     public Agencia agencia;
6
7     public Conta(Agencia agencia) {
8         this.agencia = agencia;
9     }
10
11    public void deposita(double valor) {
12        this.saldo += valor;
13    }
14
15    public void saca(double valor) {
16        this.saldo -= valor;
17    }
18
19    public void imprimeExtrato() {
20        System.out.println("SALDO: " + this.saldo);
21    }
22
23    public double consultaSaldoDisponivel() {
24        return this.saldo + this.limite;
25    }
26 }
```

*Código Java 3.70: Conta.java*

- 27** Verifique as classes TestaConta, TestaContaEAgencia, TestaMetodosConta e TestaValoresPadrao. Observe os erros de compilação.

- 28** Altere o código das classes TestaConta, TestaContaEAgencia, TestaMetodosConta e TestaValoresPadrao para que os erros de compilação sejam resolvidos.

```

1 public class TestaConta {
2     public static void main(String[] args) {
3         Agencia a = new Agencia(123);
4
5         Conta c1 = new Conta(a);
6         c1.numero = 1234;
7         c1.saldo = 1000;
8         c1.limite = 500;
9
10        Conta c2 = new Conta(a);
11        c2.numero = 5678;
12        c2.saldo = 2000;
13        c2.limite = 250;
14
15        System.out.println("Dados da primeira conta");
16        System.out.println("Número: " + c1.numero);
17        System.out.println("Saldo: " + c1.saldo);
```

```

18     System.out.println("Limite: " + c1.limite);
19
20     System.out.println("-----");
21
22     System.out.println("Dados da segunda conta");
23     System.out.println("Número: " + c2.numero);
24     System.out.println("Saldo: " + c2.saldo);
25     System.out.println("Limite: " + c2.limite);
26 }
27 }
```

Código Java 3.71: TestaConta.java

```

1 public class TestaContaEAgencia {
2     public static void main(String[] args) {
3         Agencia a = new Agencia(1234);
4         Conta c = new Conta(a);
5
6         c.numero = 123;
7         c.saldo = 1000.0;
8         c.limite = 500;
9
10        System.out.println("Dados da agência");
11        System.out.println("Número: " + a.numero);
12
13        System.out.println("-----");
14
15        System.out.println("Dados da conta");
16        System.out.println("Número: " + c.numero);
17        System.out.println("Saldo: " + c.saldo);
18        System.out.println("Limite: " + c.limite);
19
20        System.out.println("-----");
21
22        System.out.println("Dados do agência obtidos através da conta");
23        System.out.println(c.agencia.numero);
24    }
25 }
```

Código Java 3.72: TestaContaEAgencia.java

```

1 public class TestaMetodosConta {
2     public static void main(String[] args) {
3         Agencia a = new Agencia(123);
4
5         Conta c = new Conta(a);
6
7         System.out.println("Chamando o método deposita passando o valor 1000");
8         c.deposita(1000);
9         c.imprimeExtrato();
10
11        System.out.println("-----");
12
13        System.out.println("Chamando o método saca passando o valor 100");
14        c.saca(100);
15        c.imprimeExtrato();
16
17        System.out.println("-----");
18
19        double saldoDisponivel = c.consultaSaldoDisponivel();
20        System.out.println("SALDO DISPONÍVEL: " + saldoDisponivel);
21    }
22 }
```

Código Java 3.73: TestaMetodosConta.java

```
1 public class TestaValoresPadrao {
```

```

2 public static void main(String[] args) {
3     Agencia a = new Agencia(123);
4
5     Conta c = new Conta(a);
6
7     System.out.println("Valores Padrão");
8     System.out.println("Número: " + c.numero);
9     System.out.println("Saldo: " + c.saldo);
10    System.out.println("Limite: " + c.limite);
11 }
12 }
```

*Código Java 3.74: TestaValoresPadrao.java*

Exexeute novamente as classes TestaConta, TestaContaEAgencia, TestaMetodosConta e TestaValoresPadrao.



## Referências como parâmetro

Da mesma forma que podemos passar valores primitivos como parâmetro para um método ou construtor, também podemos passar valores não primitivos (referências).

Considere 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 referência da conta que receberá o dinheiro.

```

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

*Código Java 3.75: Método transfere()*

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);
```

*Código Java 3.76: Chamando o método transfere()*

Quando a variável `destino` é passada como parâmetro, somente a referência armazenada nessa variável é enviada para o método `transfere()` e não o objeto em si. Em outras palavras, somente o “endereço” para a conta que receberá o valor da transferência é enviado para o método `transfere()`.



## Exercícios de Fixação

- 29** Acrescente um método na classe Conta para implementar a lógica de transferência de valores entre contas.

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

Código Java 3.77: Método transfere()

- 30 Faça um teste para verificar o funcionamento do método transfere. Crie uma nova classe chamada **TestaMetodoTransfere** com o código abaixo.

```
1 public class TestaMetodoTransfere {  
2     public static void main(String[] args) {  
3         Agencia a = new Agencia(1234);  
4  
5         Conta origem = new Conta(a);  
6         origem.saldo = 1000;  
7         System.out.println("Saldo da primeira conta: " + origem.saldo);  
8  
9         Conta destino = new Conta(a);  
10        destino.saldo = 1000;  
11        System.out.println("Saldo da segunda conta: " + destino.saldo);  
12  
13        System.out.println("-----");  
14  
15        System.out.println("Realizando a transferência");  
16        origem.transfere(destino, 500);  
17  
18        System.out.println("-----");  
19  
20        System.out.println("Saldo da primeira conta: " + origem.saldo);  
21        System.out.println("Saldo da segunda conta: " + destino.saldo);  
22    }  
23 }
```

Código Java 3.78: TestaMetodoTransfere.java

Execute a classe TestaMetodoTransfere.

# ARRAYS

Suponha que o sistema do banco tenha que gerar listas com os números das contas de uma agência. Poderíamos declarar uma variável para cada número.

```
1 int numero1;
2 int numero2;
3 int numero3;
4 ...
```

Código Java 4.1: Uma variável para cada número de conta

Contudo, não seria uma abordagem prática, pois uma agência pode ter uma quantidade muito grande de contas. Além disso, novas contas podem ser abertas todos os dias. Isso implicaria em alterações constantes no código fonte.

Quando desejamos armazenar uma grande quantidade de valores de um determinado tipo, podemos utilizar **arrays**. Um array é um objeto que pode armazenar muitos valores de um determinado tipo.

Podemos imaginar um array como sendo um armário com um determinado número de gavetas. E cada gaveta possui um rótulo com um número de identificação.

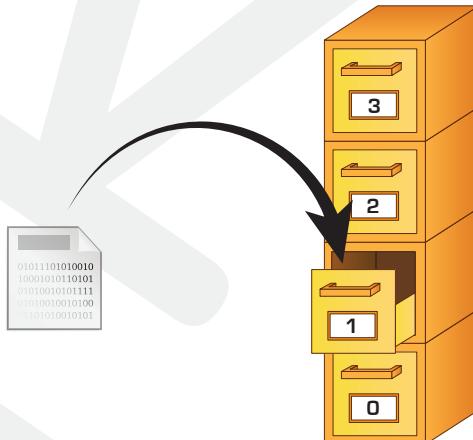


Figura 4.1: Analogia de array.



## Criando um array

Em Java, os arrays são criados através do comando `new`.

```
1 int[] numeros = new int[100];
```

Código Java 4.2: Criando um array com capacidade para 100 valores do tipo int

A variável `numeros` armazena a referência de um array criado na memória do computador através do comando `new`. 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 `int`.

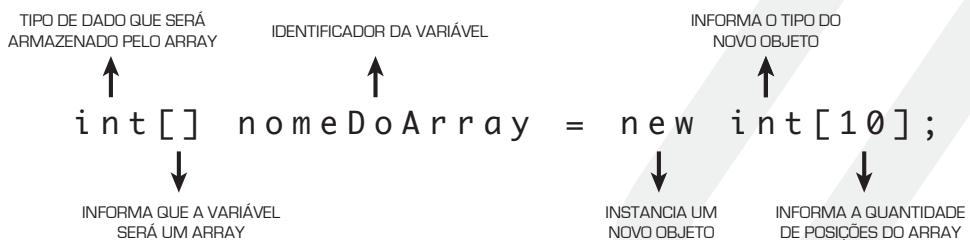


Figura 4.2: Criando um array.



## Modificando o conteúdo de um array

Para modificar o conteúdo de um array, devemos escolher uma ou mais posições que devem ser alteradas e utilizar a sintaxe abaixo:

```
1 int[] numeros = new int[100];
2 numeros[0] = 136;
3 numeros[99] = 17;
```

Código Java 4.3: Modificando o conteúdo das posições 0 e 99



### Importante

Quando um array é criado com o comando `new`, todas as posições são inicializadas com os valores padrão (números são inicializados com 0, booleanos com `false` e referências com `null`).

Também podemos definir os valores de cada posição de um array no momento da sua criação utilizando as sintaxes abaixo:

```
1 int[] numeros = new int[]{100, 87};
```

Código Java 4.4: Inicializando o conteúdo de um array

```
1 int[] numeros = {100, 87};
```

Código Java 4.5: Inicializando o conteúdo de um array



## Acessando o conteúdo de um array

Para acessar o conteúdo de um array, devemos escolher uma ou mais posições e utilizar a sintaxe abaixo:

```
1 int[] numeros = {100, 87};
2 System.out.println(numeros[0]);
```

```
3 System.out.println(numero[1]);
```

*Código Java 4.6: Acessando o conteúdo das posições 0 e 1*



### Importante

Acessar posições fora do intervalo de índices de um array gera erro de execução. Mais especificamente, em Java, ocorrerá a exception **ArrayIndexOutOfBoundsException**.



## Percorrendo um Array

Quando trabalhamos com um array, uma das tarefas mais comuns é acessarmos todas ou algumas de suas posições sistematicamente. Geralmente, fazemos isso para resgatar todos ou alguns dos valores armazenados e realizar algum processamento sobre tais informações.

Para percorrermos um array, utilizaremos a instrução de repetição `for`. Podemos utilizar a instrução `while` também. Porém, logo perceberemos que a sintaxe da instrução `for`, em geral, é mais apropriada quando estamos trabalhando com arrays.

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

*Código Java 4.7: Percorrendo um array*

Para percorrer um array, é necessário saber a quantidade de posições do mesmo. Essa quantidade é definida quando o array é criado através do comando `new`. Nem sempre essa informação está explícita no código. Por exemplo, considere um método que imprima na saída padrão os valores armazenados em um array. Provavelmente, esse método receberá como parâmetro um array e a quantidade de posições desse array não estará explícita no código fonte.

```
1 void imprimeArray(int[] numeros) {
2     // implementação
3 }
```

*Código Java 4.8: Método que deve imprimir o conteúdo de um array de int*

Podemos recuperar a quantidade de posições de um array acessando o seu atributo `length`.

```
1 void imprimeArray(int[] numeros) {
2     for(int i = 0; i < numeros.length; i++) {
3         System.out.println(numeros[i]);
4     }
5 }
```

*Código Java 4.9: Método que deve imprimir o conteúdo de um array de int*



## foreach

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

```
1 void imprimeArray(int[] numeros) {  
2     for(int numero : numeros) {  
3         System.out.println(numero);  
4     }  
5 }
```

*Código Java 4.10: Percorrendo um array com foreach*



## Operações

Nas bibliotecas da plataforma Java, existem métodos que realizam algumas tarefas úteis relacionadas a arrays. Veremos esses métodos a seguir.

### Ordenando um Array

Considere um array de `String` criado para armazenar nomes de pessoas. Podemos ordenar esses nomes 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     System.out.println(nome);  
6 }
```

*Código Java 4.11: Ordenando um array*

Analogamente, também podemos ordenar números.

### Duplicando 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[] nomesDuplicados = Arrays.copyOf(nomes, 10);
```

*Código Java 4.12: Duplicando*

### Preenchendo um Array

Podemos preencher todas as posições de um array com um valor específico utilizando o método `Arrays.fill()`.

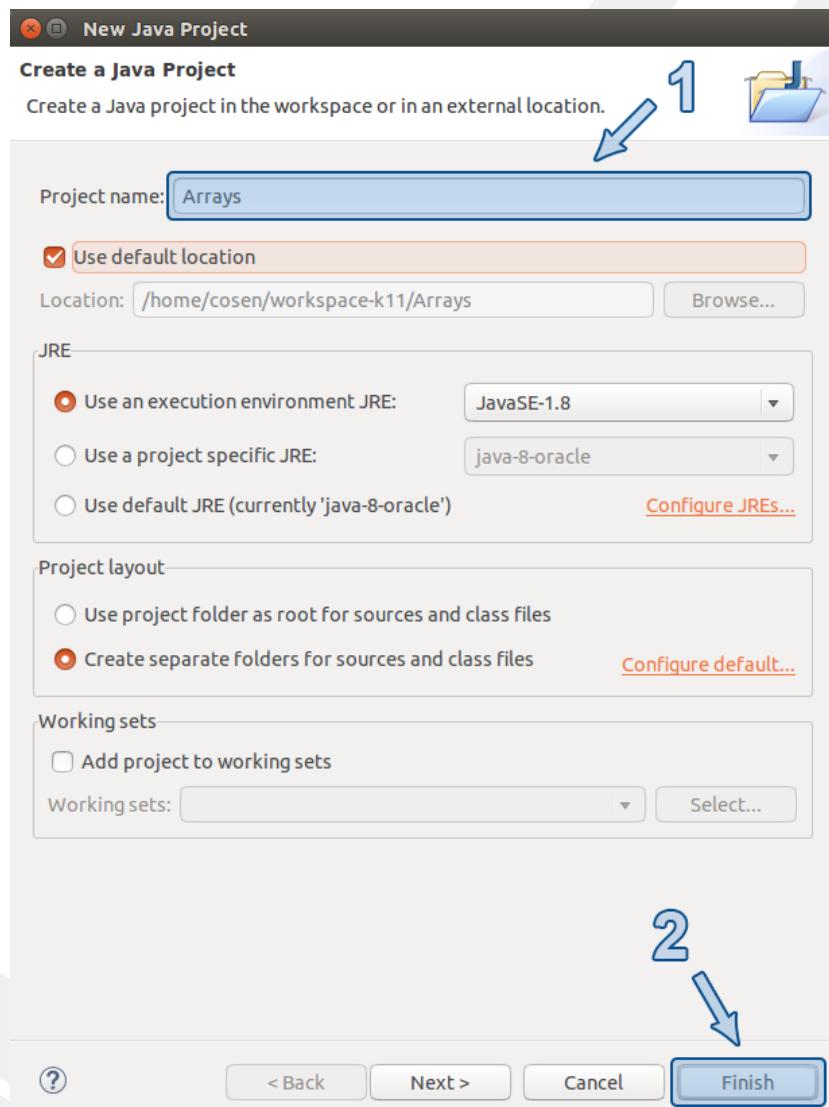
```
1 int[] numeros = new int[10];  
2 java.util.Arrays.fill(numeros, 5);
```

*Código Java 4.13: Preenchendo um array com um valor específico*



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **OrientacaoAObjetos** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie um programa que exiba no Console os argumentos passados na linha de comando para o método main. Faça uma classe chamada **ExibeArgumentos** com o seguinte conteúdo.

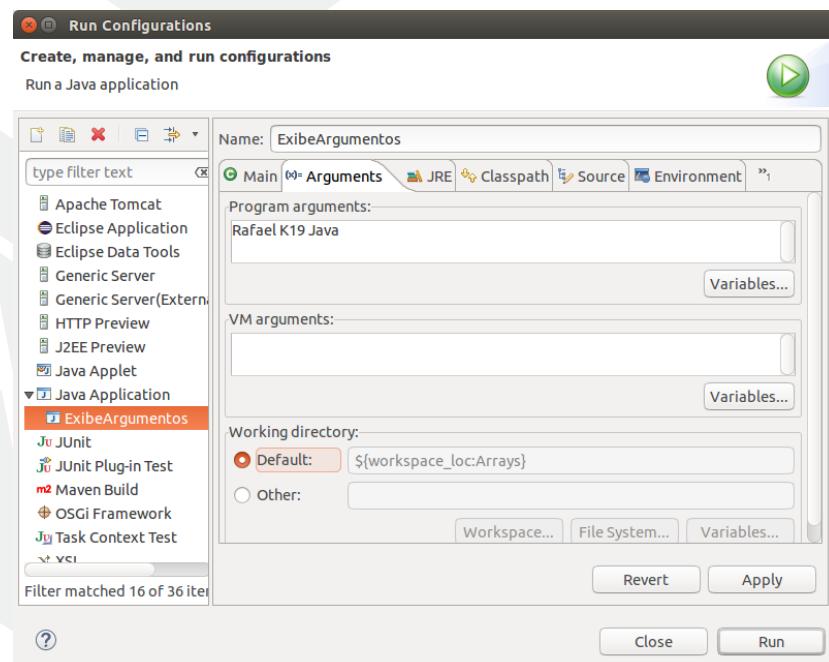
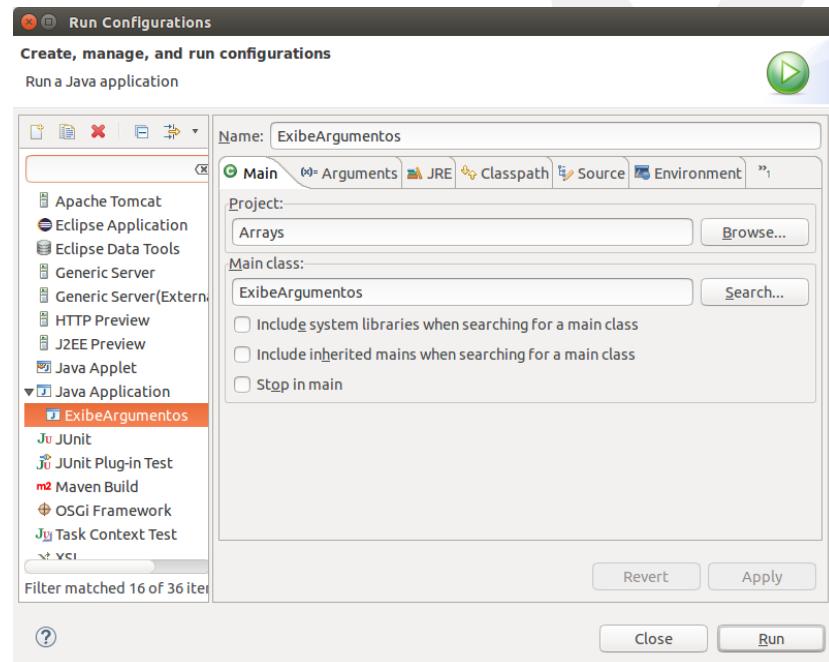
```

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

*Código Java 4.14: ExibeArgumentos.java*

Execute a classe ExibeArgumentos e observe que nada é exibido no Console já que nenhum parâmetro foi passado para o método main.

- 3 Defina os parâmetros que devem ser passados para o método main da classe ExibeArgumentos. Digite “CTRL + 3” e pesquise por “Run Configurations”. Selecione a opção correspondente e siga as imagens abaixo.



Observe os parâmetros exibidos no Console. Repita o processo algumas vezes para passar parâmetros diferentes para o método main da classe ExibeArgumentos.

- 4** Faça um programa que ordene o array de strings passado para o método main. Crie uma classe chamada **OrdenaArgumentos** com o seguinte conteúdo.

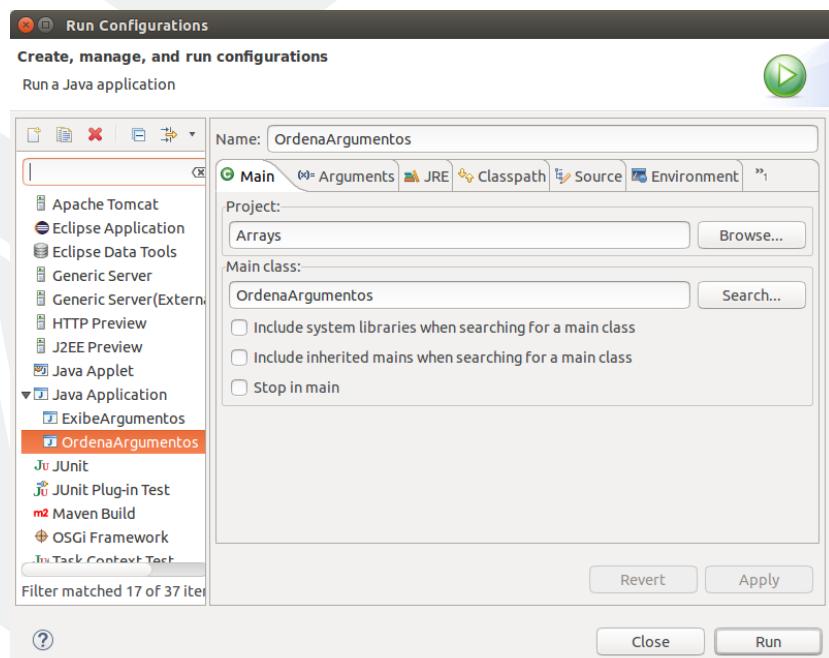
```

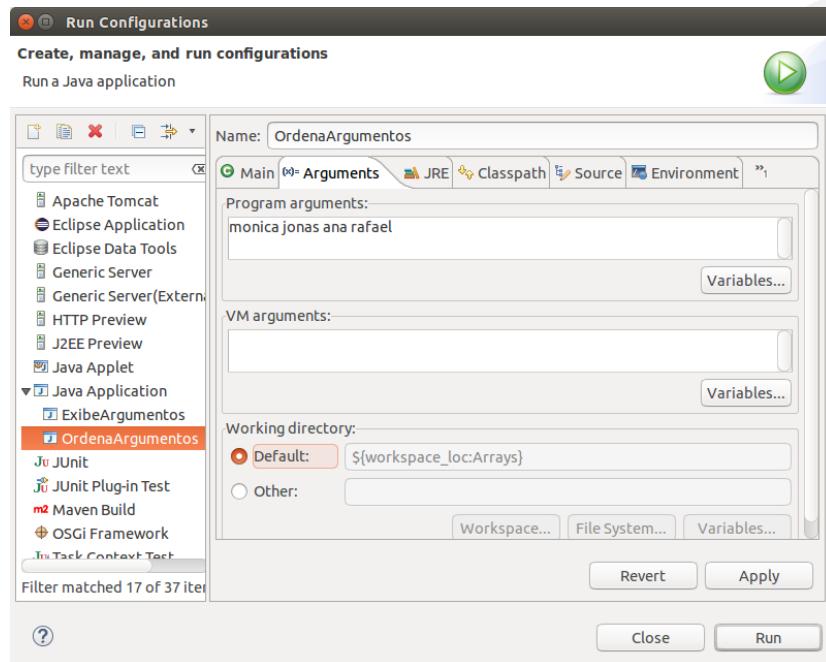
1 public class OrdenaArgumentos {
2     public static void main(String[] args) {
3         java.util.Arrays.sort(args);
4
5         for(String arg : args) {
6             System.out.println(arg);
7         }
8     }
9 }
```

Código Java 4.15: OrdenaArgumentos.java

Execute a classe OrdenaArgumentos e observe que nada é exibido no Console já que nenhum parâmetro foi passado para o método main.

- 5** Defina os parâmetros que devem ser passados para o método main da classe OrdenaArgumentos. Digite “CTRL + 3” e pesquise por “Run Configurations”. Selecione a opção correspondente e siga as imagens abaixo.





Observe os parâmetros exibidos no Console. Repita o processo algumas vezes para passar parâmetros diferentes para o método `main` da classe `OrdeneArgumentos`.



## Exercícios Complementares

- Faça um programa que calcule a média dos parâmetros passados para o método `main`. Crie uma classe chamada **Media** para implementar esse programa. Dica: para converter strings para double utilize o método `parseDouble()`

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

*Código Java 4.16: Exemplo de uso do método parseDouble*

- Faça um programa que encontre o maior número entre os parâmetros passados para o método `main`. Crie uma classe chamada **Maior** para implementar esse programa.

# ATRIBUTOS E MÉTODOS DE CLASSE



## Atributos Estáticos

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

```

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

*Código Java 5.1: Funcionario.java*

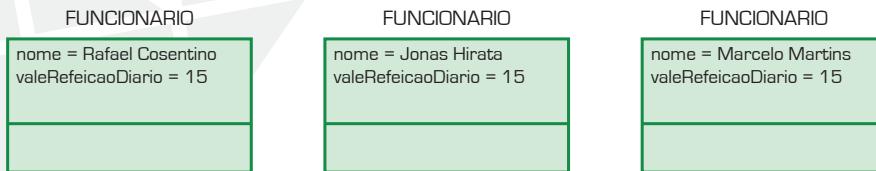
Suponha que o banco paga aos seus funcionários um valor padrão de vale refeição por dia trabalhado. O sistema do banco precisa guardar esse valor. Poderíamos definir um atributo na classe Funcionario para tal propósito.

```

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

*Código Java 5.2: Funcionario.java*

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



*Figura 5.1: Atributos de instância*

Para que o atributo valeRefeicaoDiario não se repita em cada objeto da classe Funcionario,

devemos torná-lo um atributo de classe ao invés de um atributo de instância. Para isso, devemos aplicar o modificador **static** na declaração do atributo.

```

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

*Código Java 5.3: Funcionario.java*

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

```
1 Funcionario.valeRefeicaoDiario = 15;
```

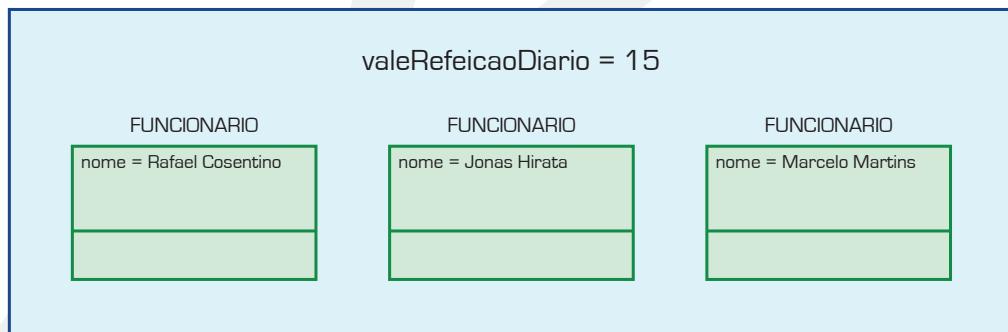
*Código Java 5.4: Acessando um atributo de classe*

Podemos acessar um atributo de classe através de uma referência de um objeto da classe na qual o atributo foi definido. Contudo, não seria a maneira conceitualmente correta já que o atributo pertence a classe e não ao objeto.

```

1 Funcionario f = new Funcionario();
2 // Válido, mas conceitualmente incorreto
3 f.valeRefeicaoDiario = 15;
```

*Código Java 5.5: Acessando um atributo de classe*



*Figura 5.2: Atributos de classe*



## Métodos Estáticos

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 valor do vale refeição baseado em uma taxa. Poderíamos definir um método na classe `Funcionario` para implementar esse reajuste.

```

1 void reajustaValeRefeicaoDiario(double taxa) {
2     Funcionario.valeRefeicaoDiario += Funcionario.valeRefeicaoDiario * taxa;
```

```
3 }
```

*Código Java 5.6: Método que reajusta o valor do vale refeição*

O método `reajustaValeRefeicaoDiario()` é de instância. Consequentemente, ele deve ser chamado a partir da referência de um objeto da classe `Funcionario`.

Contudo, como o reajuste do valor do vale refeição 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 esse reajuste.

Neste caso, poderíamos definir o `reajustaValeRefeicaoDiario()` como método de classe ao invés de método de instância. Aplicando o modificador `static` nesse método, ele se tornará um método de classe. Dessa forma, o reajuste poderia ser executado independentemente da existência de objetos da classe `Funcionario`.

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

*Código Java 5.7: Método que reajusta o valor do vale refeição*

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

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

*Código Java 5.8: Chamando um método de classe*

Podemos chamar um método de classe através de uma referência de um objeto da classe na qual o método foi definido. Contudo, não seria a maneira conceitualmente correta já que o método pertence a classe e não ao objeto.

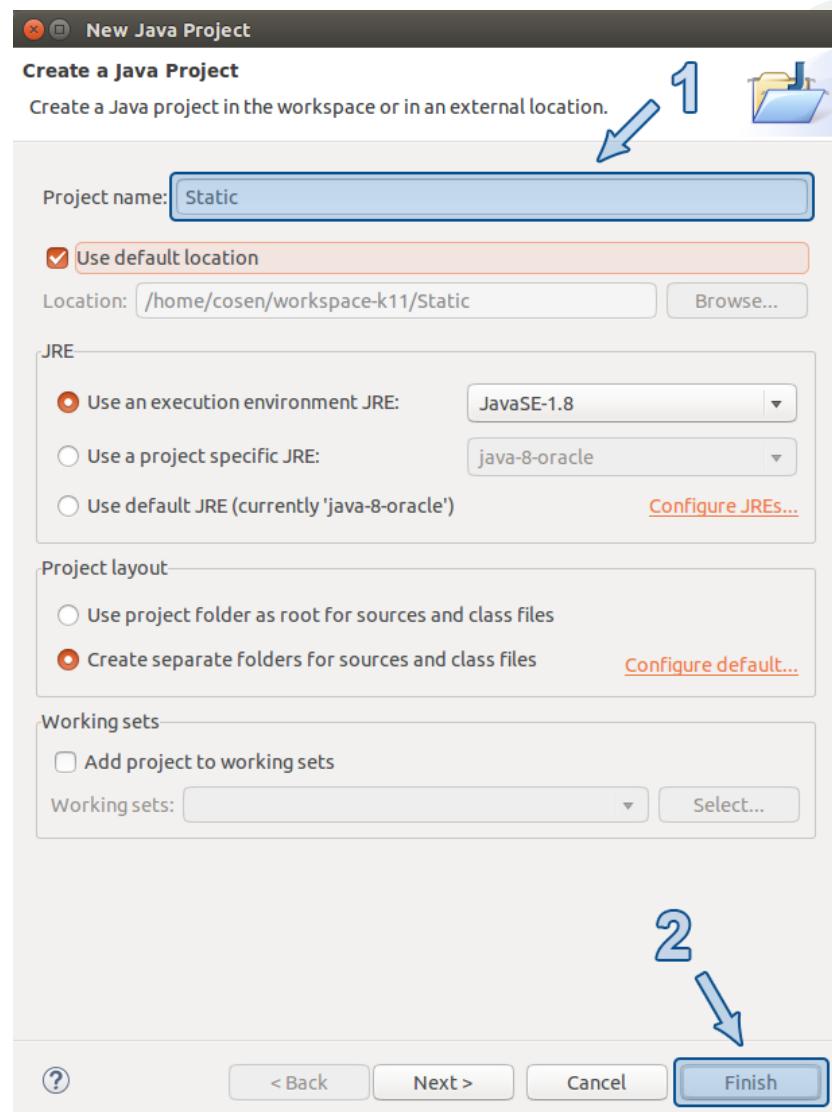
```
1 Funcionario f = new Funcionario();  
2 // Válido, mas conceitualmente incorreto  
3 f.reajustaValeRefeicaoDiario(0.1);
```

*Código Java 5.9: Chamando um método de classe*



## Exercícios de Fixação

- Para não confundir, feche o projeto **Arrays** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2** Crie uma classe chamada **Conta** no projeto **Static**. Defina um atributo para contabilizar o número de objetos instanciados a partir da classe Conta. Esse atributo deve ser incrementado toda vez que um objeto é criado. Utilize um construtor para fazer esse incremento.

```

1 public class Conta {
2     public int contador;
3
4     public Conta() {
5         this.contador++;
6     }
7 }
```

Código Java 5.10: Conta.java

- 3** Faça um teste criando dois objetos da classe Conta. Exiba o valor do atributo contador depois da criação de cada objeto. Crie uma classe chamada **TestaContador** com o seguinte conteúdo.

```
1 public class TestaContador {
```

```

1 public static void main(String[] args) {
2     Conta c1 = new Conta();
3
4     System.out.println("Contador: " + c1.contador);
5
6     Conta c2 = new Conta();
7
8     System.out.println("Contador: " + c2.contador);
9
10 }
11 }
```

Código Java 5.11: TestaContador.java

Execute a classe TestaContador e observe que o valor 1 é exibido duas vezes.

- 4 Altere a classe Conta. O atributo contador deve ser um atributo de classe. Então, adicione o modificador **static** nesse atributo. Além disso, no construtor, acesse o atributo contador com o nome da classe e não com a variável **this**.

```

1 public class Conta {
2     public static int contador;
3
4     public Conta() {
5         Conta.contador++;
6     }
7 }
```

Código Java 5.12: Conta.java

- 5 Altere a classe TestaContador. Acesse o atributo contador através do nome da classe e não através das variáveis locais que armazenam referências de objetos do tipo Conta. Além disso, exiba o valor do atributo contador antes dos objetos da classe Conta serem criados.

```

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

Código Java 5.13: TestaContador.java

Execute a classe TestaContador e observe que o valor do contador está correto.

- 6 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 os números das contas. Implemente no construtor a lógica para gerar esses números de forma única através do contador de contas.

```

1 public class Conta {
2     public static int contador;
3 }
```

```

1 public int numero;
2
3 public Conta() {
4     Conta.contador++;
5     this.numero = Conta.contador;
6 }
7
8 }
```

Código Java 5.14: Conta.java

- 7 Altere a classe TestaContador. Exiba no Console os números das contas.

```

1 public class TestaContador {
2     public static void main(String[] args) {
3         System.out.println("Contador: " + Conta.contador);
4
5         Conta c1 = new Conta();
6         System.out.println("Número da primeira conta: " + c1.numero);
7
8         System.out.println("Contador: " + Conta.contador);
9
10        Conta c2 = new Conta();
11        System.out.println("Número da segunda conta: " + c2.numero);
12
13        System.out.println("Contador: " + Conta.contador);
14    }
15 }
```

Código Java 5.15: TestaContador.java

Execute a classe TestaContador.

- 8 Adicione um método de classe na classe Conta para zerar o contador e exibir o total de contas anterior.

```

1 public static void zeraContador() {
2     System.out.println("Valor atual do contador: " + Conta.contador);
3     System.out.println("Zerando o contador de contas...");
4     Conta.contador = 0;
5 }
```

Código Java 5.16: Método zeraContador()

- 9 Altere a classe TestaContador. Utilize o método zeraContador().

```

1 public class TestaContador {
2     public static void main(String[] args) {
3         System.out.println("Contador: " + Conta.contador);
4
5         Conta c1 = new Conta();
6         System.out.println("Número da primeira conta: " + c1.numero);
7
8         System.out.println("Contador: " + Conta.contador);
9
10        Conta c2 = new Conta();
11        System.out.println("Número da segunda conta: " + c2.numero);
12
13        System.out.println("Contador: " + Conta.contador);
14
15        Conta.zeraContador();
```

```
16 }  
17 }
```

Código Java 5.17: TestaContador.java

Execute a classe TestaContador.



## Exercícios Complementares

---

- 1** Crie uma classe chamada **Funcionario** para modelar os funcionários do banco. Considere que esses funcionários possuem nome e salário. Defina nessa classe um atributo para armazenar o valor do vale refeição diário pago aos funcionários. Considere que esse valor é igual para todos os funcionários.
- 2** Faça um teste para verificar o funcionamento do atributo que armazena o valor do vale refeição dos funcionários. Altere e exiba no Console o valor desse atributo. Crie uma classe chamada **TestaValeRefeicao**.
- 3** Defina um método de classe na classe **Funcionario** para reajustar o vale refeição diário a partir de uma taxa.
- 4** Teste o método criado no exercício anterior alterando a classe **TestaValeRefeicao**.



# ENCAPSULAMENTO



## Atributos Privados

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

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

Código Java 6.1: Funcionario.java

O atributo `salario` pode ser acessado ou modificado por código escrito em qualquer classe que esteja no mesmo diretório que a classe `Funcionario`. Portanto, o controle desse atributo é descentralizado.

Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos da pasta onde a classe `Funcionario` está definida. Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.

Podemos obter um controle centralizado tornando o atributo `salario` **privado** e definindo métodos para implementar todas as lógicas que utilizam ou modificam 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 }
```

Código Java 6.2: Funcionario.java

Um atributo privado só pode ser acessado ou alterado por código escrito dentro da classe na qual ele foi definido. Se algum código fora da classe `Funcionario` tentar acessar ou alterar o valor do atributo privado `salario`, um erro de compilação será gerado.

Definir todos os atributos como privado 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 dos dados dos objetos para facilitar a manutenção do sistema e a detecção de erros.



## Métodos Privados

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

No exemplo abaixo, 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.

```

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     void descontaTarifa() {
15         this.saldo -= 0.1;
16     }
17 }
```

Código Java 6.3: *Conta.java*

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador `private`.

```

1 private void descontaTarifa() {
2     this.saldo -= 0.1;
3 }
```

Código Java 6.4: Método privado *descontaTarifa()*

Qualquer chamada ao método `descontaTarifa()` realizada fora da classe `Conta` gera um erro de compilação.



## Métodos Públicos

Os métodos que devem ser chamados 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 }
```

Código Java 6.5: Conta.java



## 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 saque, depósito, transferência e geração de 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 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 dessa classe. Os corpos dos métodos constituem a **implementação** das operações dos objetos.



## Por quê encapsular?

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 pois, uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo.

O desenvolvimento é favorecido pois, uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação.

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



## Celular - Escondendo a complexidade

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus de um celular formam a **interface de uso** do mesmo. Em outras palavras, o usuário interage com esses aparelhos através dos botões, da tela e dos menus. Os dispositivos internos de um 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 constituem a **implementação** do celular.

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

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

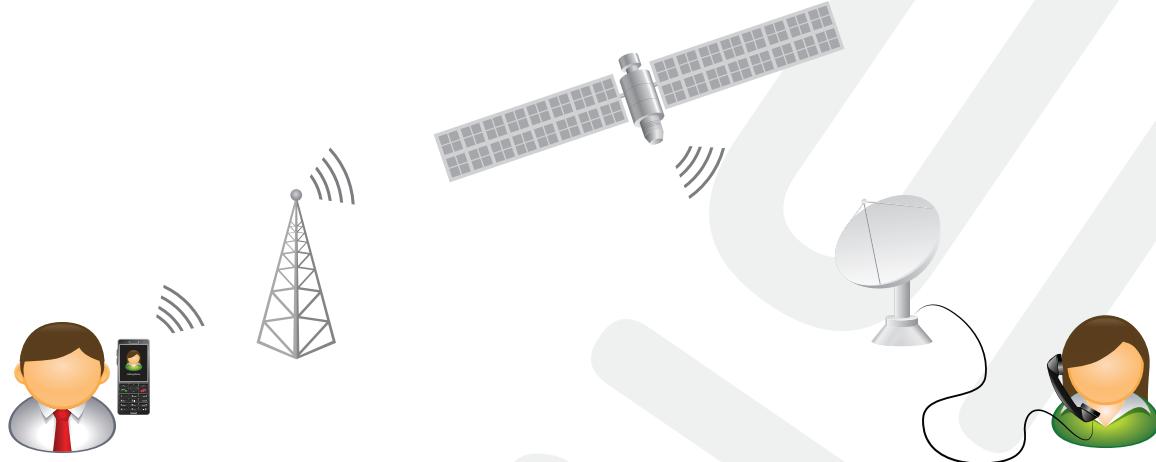


Figura 6.1: Celular



## Carro - Evitando efeitos colaterais

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

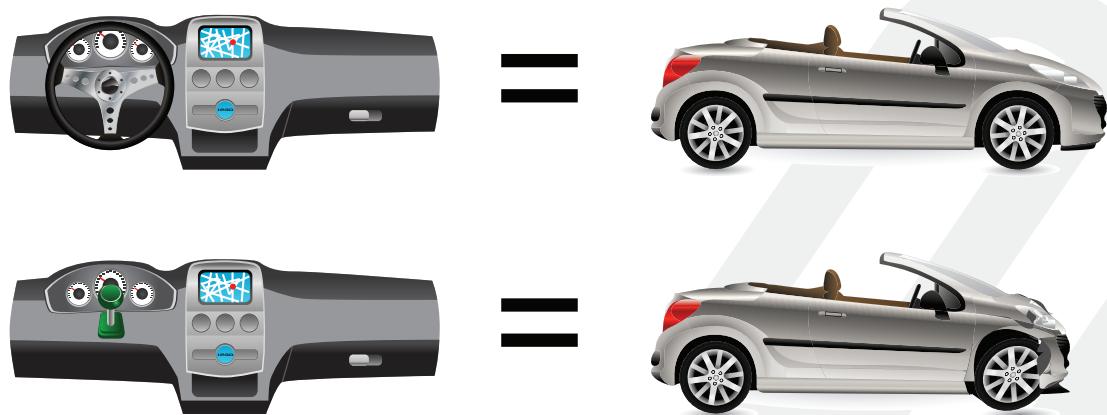
A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

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 também 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”. Dessa forma, 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.

Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.



*Figura 6.2: Substituição de um volante por um joystick*



## Máquinas de Porcarias - Aumentando o controle

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.

Normalmente, essas máquinas são extremamente protegidas. Elas garantem 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 seu funcionamento interno.



*Figura 6.3: Máquina de Porcarias*



## Acessando ou modificando atributos

Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Consequentemente, os atributos não podem ser acessados ou modificados 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 }
```

Código Java 6.6: Cliente.java

Da mesma forma, eventualmente, é necessário modificar 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 }
```

Código Java 6.7: Cliente.java

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 e outro de escrita) ou deixar o atributo público?

Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?

Quando queremos alterar o toque da campainha de um celular, utilizamos os menus do celular ou desmontamos o aparelho?

Acessar ou modificar as propriedades de um objeto manipulando diretamente os seus atributos é uma abordagem que normalmente gera problemas. Por isso, é mais seguro para a integridade dos objetos e, consequentemente, para a integridade da aplicação, que esse acesso ou essa modificação sejam realizados através de métodos do objeto. Utilizando métodos, podemos controlar como as alterações e as consultas são realizadas. Ou seja, temos um controle maior.



## Getters e Setters

Na linguagem Java, há uma convenção de nomenclatura para os métodos que têm como finalidade acessar ou alterar as propriedades de um objeto.

Segundo essa convenção, os nomes dos métodos que permitem a consulta das propriedades de um objeto devem possuir o prefixo **get**. Analogamente, os nomes dos métodos que permitem a alteração das propriedades de um objeto devem possuir o prefixo **set**.

Na maioria dos casos, é muito conveniente seguir essa convenção, pois os desenvolvedores Java já estão acostumados com essas regras de nomenclatura e o funcionamento de muitas bibliotecas do Java depende fortemente desse padrão.

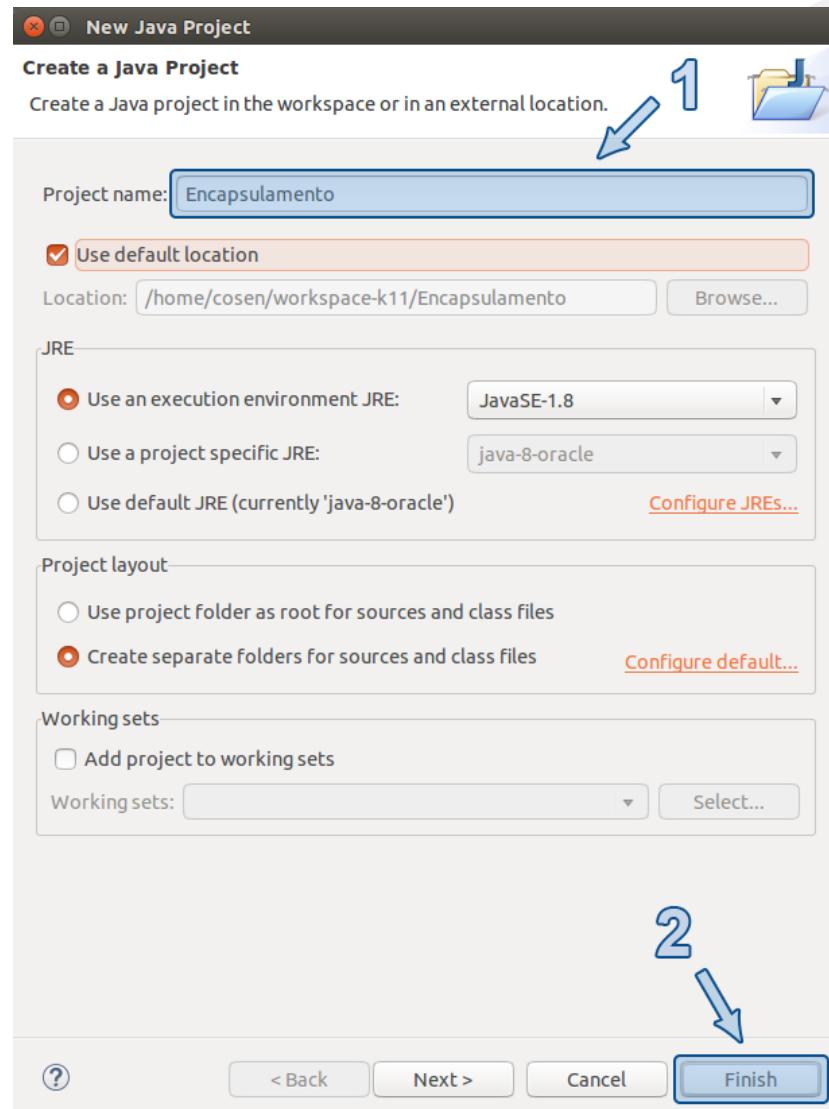
```
1 class Cliente {  
2     private String nome;  
3  
4     public String getNome() {  
5         return this.nome;  
6     }  
7  
8     public void setNome(String nome) {  
9         this.nome = nome;  
10    }  
11 }
```

Código Java 6.8: Cliente.java



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Static** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe chamada **Funcionario** para representar os funcionários de um banco com um atributo para guardar os salários e outro para os nomes.

```

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

Código Java 6.9: Funcionario.java

- 3 Crie um objeto da classe Funcionario. Altere e exiba os valores dos atributos desse objeto. Adicione uma classe chamada **Teste** no projeto **Encapsulamento**.

```

1 public class Teste {
2     public static void main(String[] args) {
3         Funcionario f = new Funcionario();
4         f.nome = "Rafael Cosentino";
5 }
```

```

6     f.salario = 2000;
7
8     System.out.println("Nome: " + f.nome);
9     System.out.println("Salário: " + f.salario);
10    }
11 }
```

*Código Java 6.10: Teste.java*

Execute a classe Teste. Observe que essa classe pode acessar e modificar os atributos do objeto criado com a classe Funcionario.

- 4** Aplique a ideia do encapsulamento tornando os atributos definidos na classe Funcionario privados.

```

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

*Código Java 6.11: Funcionario.java*

- 5** Observe os erros de compilação na classe Teste. Agora, essa classe não pode mais acessar ou modificar os atributos do objeto criado com a classe Funcionario.

- 6** Crie métodos de acesso com nomes padronizados para os atributos definidos na classe Funcionario.

```

1 public 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 }
```

*Código Java 6.12: Funcionario.java*

- 7** Altere a classe Teste para que ela utilize os métodos de acesso ao invés de manipular os atributos do objeto da classe Funcionario diretamente. Dica: Com o cursor de digitação sobre o primeiro erro de compilação digite “CTRL + 1” e selecione a opção “Replace f.nome with setter”. Analogamente, o mesmo processo pode ser utilizado para resolver os outros três erros de compilação.

```

1 public 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("Nome: " + f.getNome());
9         System.out.println("Salário: " + f.getSalario());
10    }
11 }
```

Código Java 6.13: Teste.java

- 8 Remova os métodos de acesso da classe Funcionario.

```

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

Código Java 6.14: Funcionario.java

- 9 Gere os getters e setters com os recursos do Eclipse. Digite get ou set seguidos de “CTRL + ESPAÇO” para completar o código.

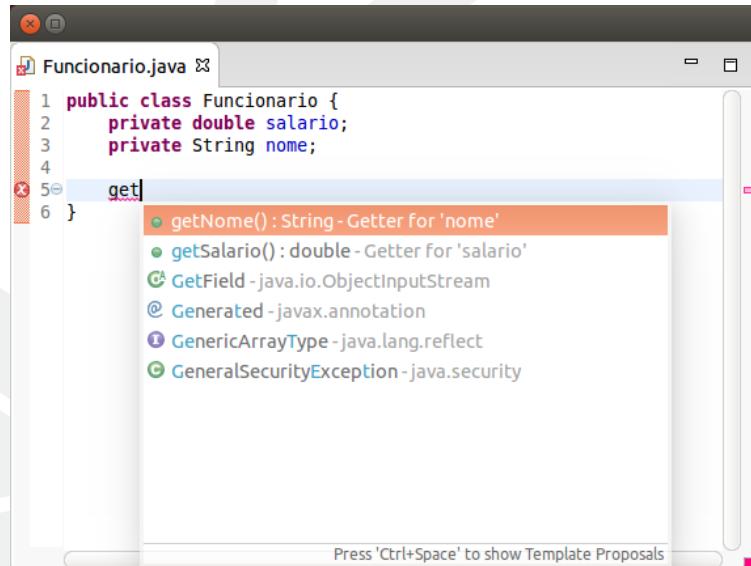


Figura 6.4: Gerando os getters e setters

Outra possibilidade é digitar “CTRL + 3” e pesquisar por **generate getters and setters** ou simplesmente **ggas**.

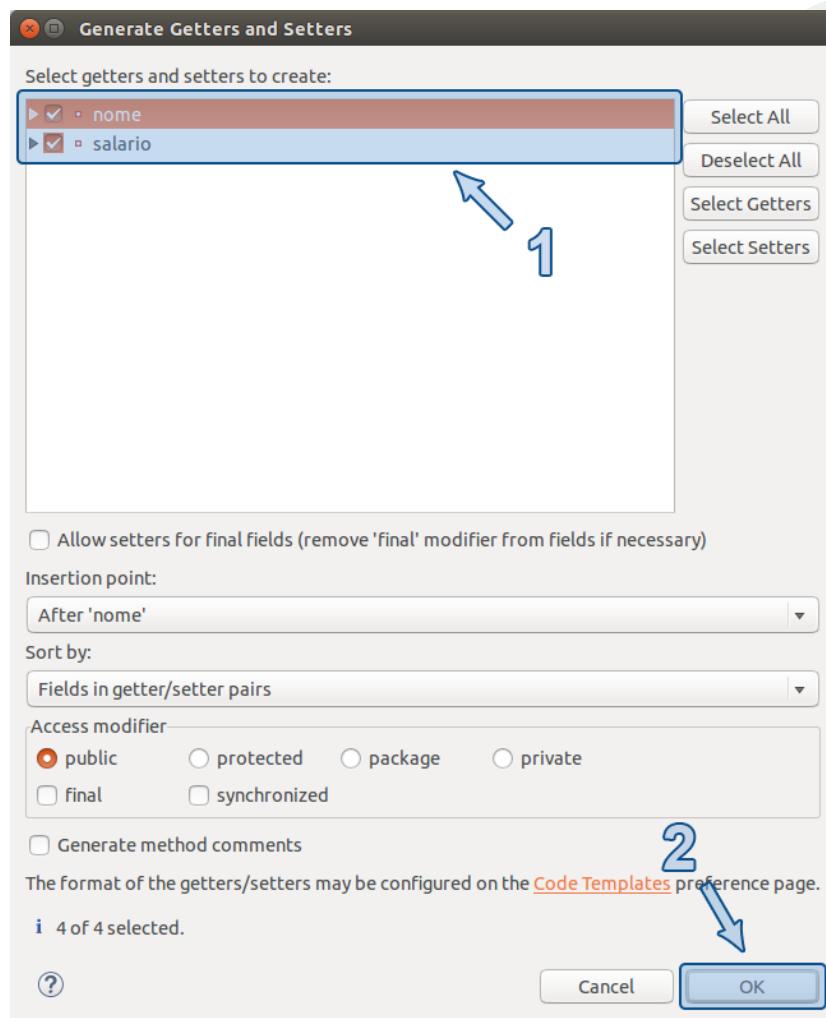


Figura 6.5: Gerando os getters e setters



## Exercícios Complementares

- 1 Implemente uma classe chamada **Conta** para modelar as contas de um banco. Considere que toda conta possui número, saldo e limite.
- 2 Adicione métodos de acesso com nomes padronizados para os atributos da classe Conta.
- 3 Crie objetos da classe Conta e utilize os métodos de acesso para alterar os valores dos atributos.
- 4 Utilize os mecanismos do Eclipse para gerar novamente os métodos de acesso.



# HERANÇA



## Reutilização de Código

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

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     // métodos  
7 }
```

Código Java 7.1: Servico.java

## 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 e outro para a taxa de juros do serviço de empréstimo.

```
1 class Servico {  
2     // GERAL  
3     private Cliente contratante;  
4     private Funcionario responsavel;  
5     private String dataDeContratacao;  
6  
7     // EMPRÉSTIMO  
8     private double valor;
```

```

9  private double taxa;
10 // métodos
11 }
12 }
```

*Código Java 7.2: Servico.java*

## 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.

```

1 class Servico {
2   // GERAL
3   private Cliente contratante;
4   private Funcionario responsavel;
5   private String dataDeContratacao;
6
7   // EMPRÉSTIMO
8   private double valor;
9   private double taxa;
10
11  // SEGURO DE VEICULO
12  private Veiculo veiculo;
13  private double valorDoSeguroDeVeiculo;
14  private double franquia;
15
16  // métodos
17 }
```

*Código Java 7.3: Servico.java*

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, eles provavelmente modificariam 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. Na verdade, ele 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 |     // métodos
13 |

```

*Código Java 7.4: SeguroDeVeiculo.java*

```

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

```

*Código Java 7.5: Emprestimo.java*

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 de orientação a objetos chamado **Herança**. A ideia é reutilizar o código de uma determinada classe em outras classes.

Aplicando herança, 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.

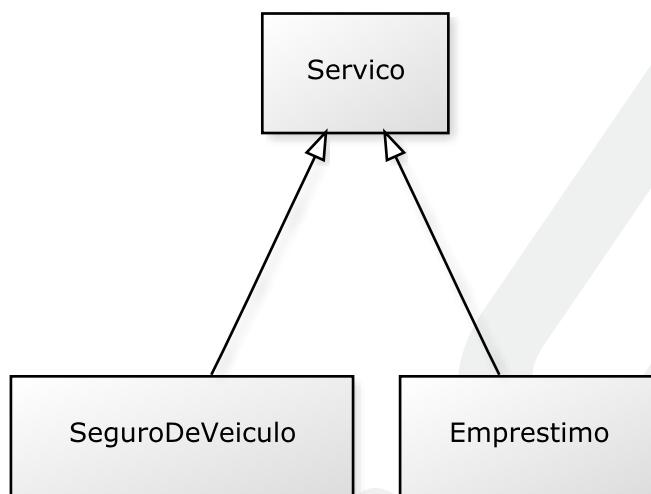


Figura 7.1: Árvore de herança dos serviços

Os objetos das classes específicas `Emprestimo` e `SeguroDeVeiculo` possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe `Serviço`.

```

1 Emprestimo e = new Emprestimo();
2
3 // Chamando um método da classe Serviço
4 e.setDataDeContratacao("10/10/2010");
5
6 // Chamando um método da classe Emprestimo
7 e.setValor(10000);
  
```

Código Java 7.6: Chamando métodos da classe genérica e da específica

As classes específicas são vinculadas a classe genérica utilizando o comando `extends`. Não é necessário redefinir o conteúdo já declarado na classe genérica.

```

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

Código Java 7.7: Serviço.java

```

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

Código Java 7.8: Emprestimo.java

```

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

Código Java 7.9: SeguroDeVeiculo.java

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

Quando o operador new é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.

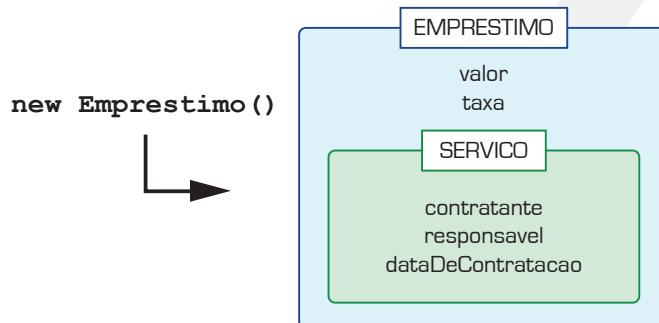


Figura 7.2: Criando um objeto a partir da sub classe



## 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 }

```

Código Java 7.10: Servico.java

```

1 Emprestimo e = new Emprestimo();
2 SeguroDeVeiculo sdv = new SeguroDeVeiculo();
3 System.out.println("Emprestimo: " + e.calculaTaxa());
4 System.out.println("SeguroDeVeiculo: " + sdv.calculaTaxa());
5
6
7

```

Código Java 7.11: Chamando o método calculaTaxa()



## Reescrita de Método

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 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 }
```

*Código Java 7.12: Emprestimo.java*

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 em um objeto da classe `Emprestimo`, pois ela herda esse método da classe `Servico`. Dessa forma, existe o risco de alguém erroneamente chamar o método incorreto.

Seria mais seguro “substituir” a implementação do método `calculaTaxa()` herdado da classe `Servico` na classe `Emprestimo`. 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 }
```

*Código Java 7.13: Emprestimo.java*

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.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de **Reescrita de Método**.



## 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 é 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 }
```

*Código Java 7.14: Emprestimo.java*

```

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

*Código Java 7.15: SeguraDeVeiculo.java*

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 }
```

*Código Java 7.16: Servico.java*

```

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

*Código Java 7.17: Emprestimo.java*

Dessa forma, quando o valor padrão do preço dos serviços é alterado, basta modificar o método na classe Servico.



## 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 é criado, pelo menos um construtor da própria classe Emprestimo e um da classe Servico devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```

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

*Código Java 7.18: Servico.java*

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

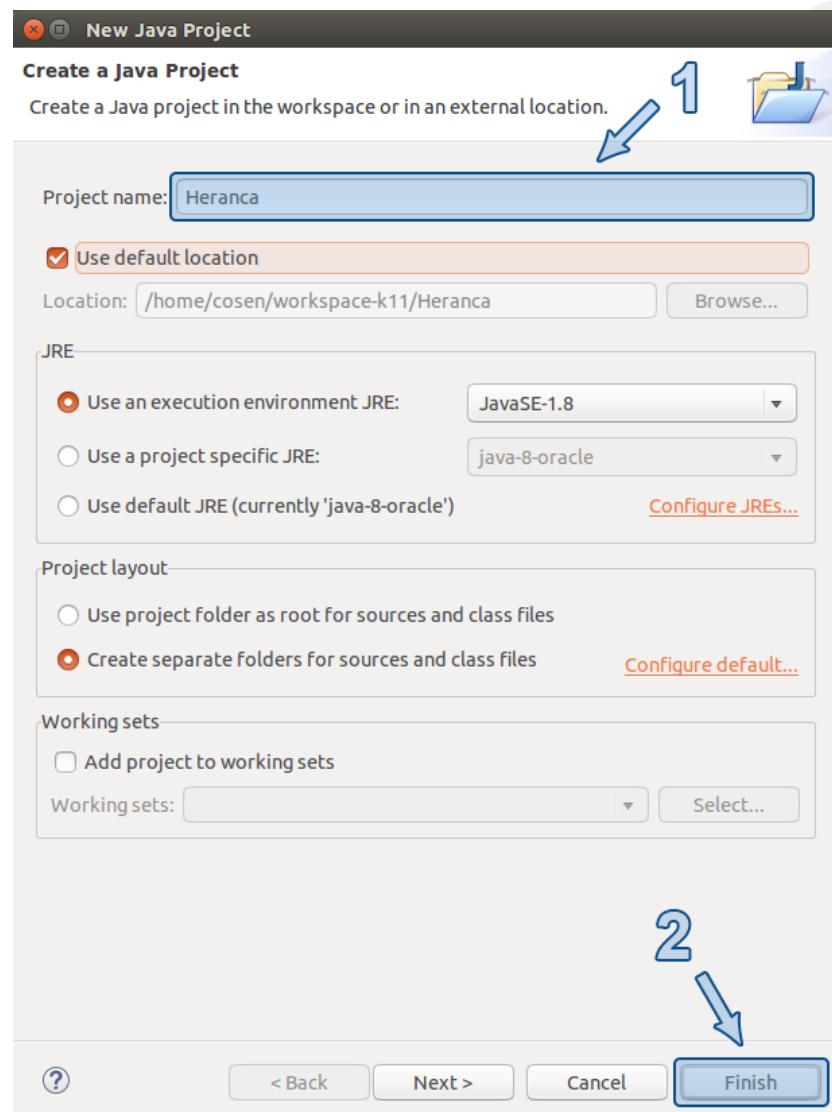
Código Java 7.19: *Emprestimo.java*

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



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Encapsulamento** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe chamada **Funcionario** para modelar os funcionários de um banco. Sabendo que todo funcionário possui nome e salário, inclua os getters e setters dos atributos.

```

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

Código Java 7.20: Funcionario.java

- 3 Crie uma classe para cada tipo específico de funcionário herdando da classe Funcionario. Considere apenas três tipos específicos de funcionários: gerentes, telefonistas e secretárias. 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 secretárias possuem um número de ramal.

```

1 public class Gerente extends Funcionario {
2     private String usuario;
3     private String senha;
4
5     // GETTERS AND SETTERS
6 }
```

Código Java 7.21: Gerente.java

```

1 public class Telefonista extends Funcionario {
2     private int estacaoDeTrabalho;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 7.22: Telefonista

```

1 public class Secretaria extends Funcionario {
2     private int ramal;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 7.23: Secretaria.java

- 4** Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: Gerente, Telefonista e Secretaria.

```

1 public 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        System.out.println("Nome: " + g.getNome());
21        System.out.println("Salário: " + g.getSalario());
22        System.out.println("Usuário: " + g.getUsuario());
23        System.out.println("Senha: " + g.getSenha());
24
25        System.out.println("TELEFONISTA");
26        System.out.println("Nome: " + t.getNome());
27        System.out.println("Salário: " + t.getSalario());
28        System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());
29
30        System.out.println("SECRETARIA");
31        System.out.println("Nome: " + s.getNome());
32        System.out.println("Salário: " + s.getSalario());
33        System.out.println("Ramal: " + s.getRamal());
34    }
35 }
```

Código Java 7.24: TestaFuncionarios.java

Execute a classe TestaFuncionarios.

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

```

1 public class Funcionario {
2     private String nome;
3     private double salario;
4
5     public double calculaBonificacao() {
6         return this.salario * 0.1;
7     }
8
9     // GETTERS AND SETTERS
10 }
```

Código Java 7.25: Funcionario.java

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

```

1 public 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        System.out.println("Nome: " + g.getNome());
21        System.out.println("Salário: " + g.getSalario());
22        System.out.println("Usuário: " + g.getUsuario());
23        System.out.println("Senha: " + g.getSenha());
24        System.out.println("Bonificação: " + g.calculaBonificacao());
25
26        System.out.println("TELEFONISTA");
27        System.out.println("Nome: " + t.getNome());
28        System.out.println("Salário: " + t.getSalario());
29        System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());
30        System.out.println("Bonificação: " + t.calculaBonificacao());
31
32        System.out.println("SECRETARIA");
33        System.out.println("Nome: " + s.getNome());
34        System.out.println("Salário: " + s.getSalario());
35        System.out.println("Ramal: " + s.getRamal());
36        System.out.println("Bonificação: " + s.calculaBonificacao());
37    }
38 }
```

Código Java 7.26: TestaFuncionarios.java

Execute a classe TestaFuncionarios e veja os valores das bonificações.

- 7 Suponha que os gerentes recebam uma bonificação maior que os outros funcionários. Reescreva o método `calculaBonificacao()` na classe `Gerente`.

```
1 public 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 }
```

Código Java 7.27: `Gerente.java`

Execute a classe `TestaFuncionarios` e veja os valores das bonificações.



## Exercícios Complementares

- 1 Defina na classe `Funcionario` um método chamado **mostraDados** para exibir no Console o nome, salário e bonificação dos funcionários.
- 2 Reescreva o método `mostraDados` nas classes `Gerente`, `Telefonista` e `Secretaria` para acrescentar os dados específicos de cada tipo de funcionário.
- 3 Modifique a classe `TestaFuncionarios` para utilizar o método `mostraDados()`.

# POLIMORFISMO



## Controle de Ponto

O sistema do banco deve possuir um controle de ponto para registrar a entrada e saída dos funcionários. O pagamento dos funcionários depende dessas informações. Podemos definir uma classe para implementar o funcionamento de um relógio de ponto.

```
1 class ControleDePonto {  
2  
3     public void registraEntrada(Gerente g) {  
4         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
5         Date agora = new Date();  
6  
7         System.out.println("ENTRADA: " + g.getCodigo());  
8         System.out.println("DATA: " + sdf.format(agora));  
9     }  
10  
11    public void registraSaida(Gerente g) {  
12        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
13        Date agora = new Date();  
14  
15        System.out.println("SAÍDA: " + g.getCodigo());  
16        System.out.println("DATA: " + sdf.format(agora));  
17    }  
18}
```

Código Java 8.1: ControleDePonto.java

A classe acima possui dois métodos: o primeiro para registrar a entrada e o segundo para registrar a saída dos gerentes do banco. Contudo, esses dois métodos não são aplicáveis aos outros tipos de funcionários.

Seguindo essa abordagem, a classe ControleDePonto precisaria de um par de métodos para cada cargo. Então, a quantidade de métodos dessa classe seria igual a quantidade de cargos multiplicada por dois. Imagine que no banco exista 30 cargos distintos. Teríamos 60 métodos na classe ControleDePonto.

Os procedimentos de registro de entrada e saída são idênticos para todos os funcionários. Consequentemente, qualquer alteração na lógica desses procedimentos implicaria na modificação de todos os métodos da classe ControleDePonto.

Além disso, se o banco definir um novo tipo de funcionário, dois novos métodos praticamente idênticos aos que já existem teriam de ser adicionados na classe ControleDePonto. Analogamente, se um cargo deixar de existir, os dois métodos correspondentes da classe ControleDePonto deverão ser retirados.

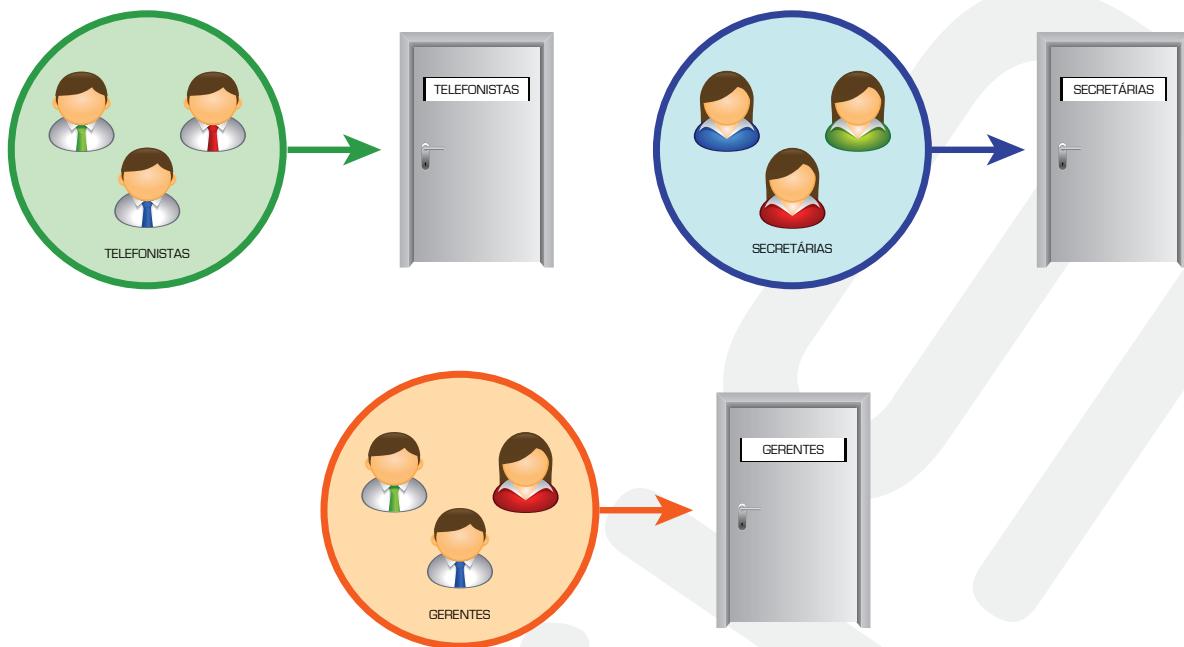


Figura 8.1: Métodos específicos



## Modelagem dos funcionários

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

```

1 class Funcionario {
2     private int codigo;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 8.2: Funcionario.java

```

1 class Gerente extends Funcionario {
2     private String usuario;
3     private String senha;
4
5     // GETTERS AND SETTERS
6 }
```

Código Java 8.3: Gerente.java

```

1 class Telefonista extends Funcionario {
2     private int ramal;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 8.4: Telefonista.java



## 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 os funcionários permite que objetos criados a partir das classes Gerente ou Telefonista sejam tratados como objetos da classe Funcionario.

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

```
1 class Gerente extends Funcionario
2 // TODO Gerente É UM Funcionario
```

*Código Java 8.5: Gerente.java*

Como está explícito no código que todo gerente é um funcionário então podemos criar um objeto da classe Gerente e tratá-lo como um objeto da classe Funcionario também.

```
1 // Criando um objeto da classe Gerente
2 Gerente g = new Gerente();
3
4 // Tratando um gerente como um objeto da classe Funcionario
5 Funcionario f = g;
```

*Código Java 8.6: Generalizando*

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



## Melhorando o controle de ponto

O registro da entrada ou saída não depende do cargo do funcionário. Não faz sentido criar um método que registre a entrada para cada tipo de funcionário, pois eles serão sempre idênticos. Analogamente, não faz sentido criar um método que registre a saída para cada tipo de funcionário.

Dado que podemos tratar os objetos das classes derivadas de Funcionario como sendo objetos dessa classe, podemos implementar um método que seja capaz de registrar a entrada de qualquer funcionário independentemente do cargo. Analogamente, podemos fazer o mesmo para o procedimento de saída.

```
1 class ControleDePonto {
2
3     public void registraEntrada(Funcionario f) {
4         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
5         Date agora = new Date();
6
7         System.out.println("ENTRADA: " + f.getCodigo());
8         System.out.println("DATA: " + sdf.format(agora));
9     }
10
11    public void registraSaida(Funcionario f) {
12        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
13        Date agora = new Date();
14    }
}
```

```

15     System.out.println("SAÍDA: " + f.getCodigo());
16     System.out.println("DATA: " + sdf.format(agora));
17 }
18 }
```

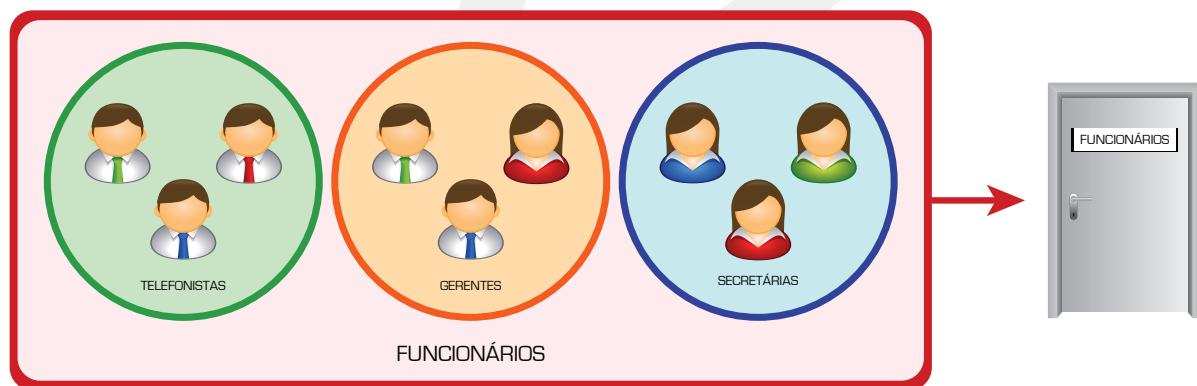
*Código Java 8.7: ControleDePonto.java*

Os métodos `registraEntrada()` e `registraSaida()` recebem referências de objetos da classe `Funcionario` como parâmetro. Consequentemente, podem receber referências de objetos de qualquer classe que deriva direta ou indiretamente da classe `Funcionario`.

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

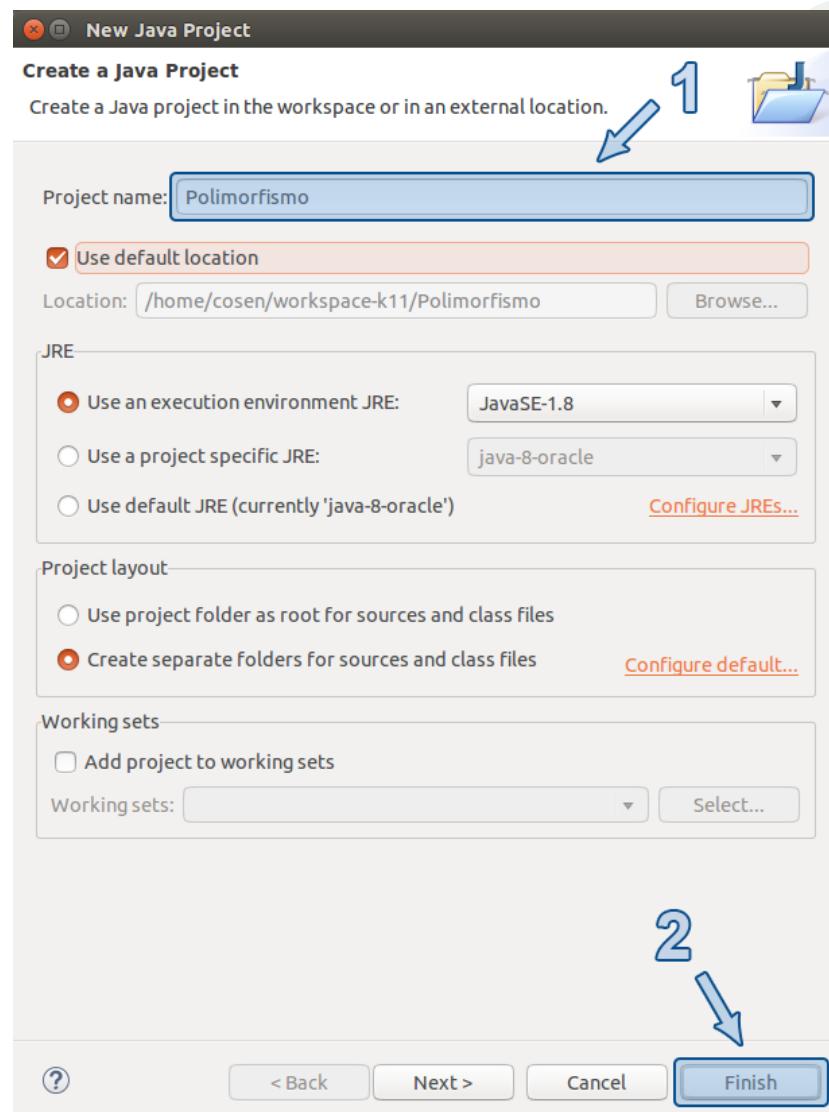
Aplicando a ideia do polimorfismo no controle de ponto, facilitamos a manutenção da classe `ControleDePonto`. Qualquer alteração no procedimento de entrada ou saída implica em alterações em métodos únicos.

Além disso, novos tipos de funcionários podem ser definidos sem a necessidade de qualquer alteração na classe `ControleDePonto`. Analogamente, se algum cargo deixar de existir, nada precisará ser modificado na classe `ControleDePonto`.

*Figura 8.2: Método genérico*

## Exercícios de Fixação

- Para não confundir, feche o projeto **Heranca** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe chamada **Conta** para modelar as contas de um banco.

```

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

Código Java 8.8: Conta.java

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

```

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

Código Java 8.9: ContaPoupanca.java

```

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

*Código Java 8.10: ContaCorrente.java*

- 4 Defina uma classe chamada **GeradorDeExtrato** com seguinte código.

```

1 import java.text.SimpleDateFormat;
2 import java.util.Date;
3
4 public 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}
```

*Código Java 8.11: GeradorDeExtrato.java*

Não se preocupe com o comando “import”. Discutiremos sobre ele posteriormente.

- 5 Teste a classe GeradorDeExtrato. Crie uma classe chamada **TestaGeradorDeExtrato**.

```

1 public 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}
```

*Código Java 8.12: TestaGeradorDeExtrato.java*

Execute a classe TestaGeradorDeExtrato.



## Exercícios Complementares

- 1 Defina uma classe chamada **Funcionario** para modelar os funcionários de um banco.
- 2 Implemente duas classes específicas para modelar dois tipos particulares de funcionários do

banco: os gerentes e as telefonistas.

- 3 Implemente o controle de ponto dos funcionários. Crie uma classe com dois métodos: o primeiro para registrar a entrada dos funcionários e o segundo para registrar a saída.
- 4 Teste a lógica do controle de ponto, registrando a entrada e a saída de um gerente e de uma telefonista.



# OBJECT

Todas as classes derivam direta ou indiretamente da classe 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. Ou seja, a ideia do polimorfismo pode ser aplicada para criar métodos genéricos que trabalham com objetos de qualquer classe.

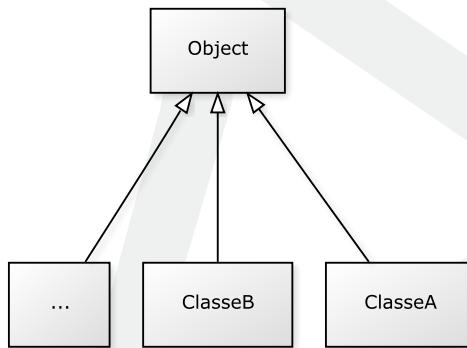


Figura 9.1: A classe Object



## 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 Repositorio {  
2     // código da classe  
3 }
```

Código Java 9.1: Repositorio.java

Um array de objetos pode ser utilizado como estrutura básica para manter os objetos no repositório.

```
1 class Repositorio {  
2     private Object[] objetos = new Object[100];  
3 }
```

Código Java 9.2: Repositorio.java

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

```

1 class Repositorio {
2     private Object[] objetos = new Object[100];
3
4     public void adiciona(Object o) {
5         // implementacao
6     }
7
8     public void remove(Object o) {
9         // implementacao
10    }
11
12    public Object pega(int posicao) {
13        // implementacao
14    }
15 }
```

*Código Java 9.3: Repositorio.java*

Com esses métodos o repositório teria a vantagem de armazenar objetos de qualquer tipo. Porem, 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);
```

*Código Java 9.4: Utilizando o repositório*

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 `String`. Nesse caso, em tempo de compilação é possível “forçar” o compilador a tratar os objetos como string através de casting de referência.

```

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

*Código Java 9.5: Casting de referência*

## O método `toString()`

Às vezes, é necessário trabalhar com uma descrição textual de determinados objetos. Por exemplo, suponha a seguinte classe:

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     // GETTERS E SETTERS
6 }
```

*Código Java 9.6: Conta.java*

Queremos gerar um documento no qual deve constar as informações de determinadas contas. Podemos implementar um método, na classe `Conta`, que gere uma descrição textual dos objetos

dessa classe.

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     public String geraDescricao() {
6         return "Conta número: " + this.numero + " possui saldo igual a " + this.saldo;
7     }
8
9     // GETTERS E SETTERS
10 }
```

*Código Java 9.7: Conta.java*

A utilização do método que gera a descrição textual das contas seria mais ou menos assim:

```

1 Conta conta = ...
2 String descricao = conta.geraDescricao();
3 System.out.println(descricao);
```

*Código Java 9.8: Utilizando o método geraDescricao()*

Contudo, a classe `Object` possui um método justamente com o mesmo propósito do `geraDescricao()` chamado `toString()`. Como todas as classes derivam direta ou indiretamente da classe `Object`, todos os objetos possuem o método `toString()`.

A implementação padrão do método `toString()` monta uma descrição genérica baseada no nome da classe mais específica e no **hash code** dos objetos.

```

1 Conta conta = ...
2 String descricao = conta.toString();
3 System.out.println(descricao);
```

*Código Java 9.9: Utilizando o método toString()*

No código acima, a descrição gerada pelo método `toString()` definido na classe `Object` é algo semelhante à string: “`Conta@4d5ef`”.

Para alterar o comportamento do método `toString()`, basta reescrevê-lo na classe `Conta`.

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     public String toString() {
6         return "Conta número: " + this.numero + " possui saldo igual a " + this.saldo;
7     }
8
9     // GETTERS E SETTERS
10 }
```

*Código Java 9.10: Conta.java*

A vantagem em reescrever o método `toString()` ao invés de criar um outro método com o mesmo propósito é que diversas classes das bibliotecas do Java utilizam o método `toString()`. Inclusive, quando passamos uma variável não primitiva para o método `println()`, o `toString()` é chamado internamente para definir o que deve ser impresso na tela.

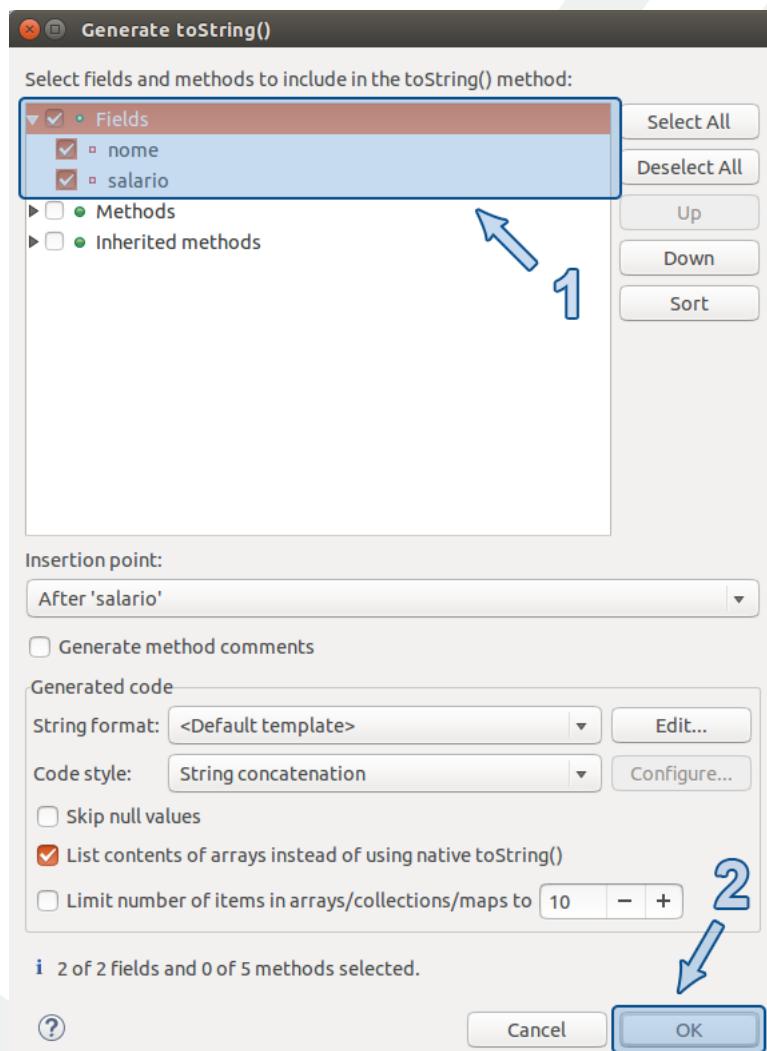
```

1 Conta conta = ...
2 // o método toString() será chamado internamente no println()
3 System.out.println(conta);

```

Código Java 9.11: Utilizando o método `toString()`

Outra vantagem em optar pelo método `toString()` é que ferramentas de desenvolvimento como o Eclipse oferecem recursos para que esse método seja reescrito automaticamente.

Figura 9.2: Gerando o `toString()` no Eclipse

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     // método gerado pelo eclipse
6     public String toString() {
7         return "Conta [numero=" + numero + ", saldo=" + saldo + "]";
8     }
9
10    // GETTERS E SETTERS
11 }

```

Código Java 9.12: `Conta.java`



## 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 Conta c1 = ...
2 Conta c2 = ...
3
4 System.out.println(c1 == c2);

```

*Código Java 9.13: Comparando com ==*

O operador “==”, aplicado à variáveis não primitivas, verifica se as referências armazenadas nessas variáveis apontam para o mesmo objeto na memória. Esse operador não compara o conteúdo dos objetos correspondentes às referências armazenadas nas variáveis submetidas à comparação.

Para comparar o conteúdo de objetos, é necessário utilizar métodos. Podemos implementar um método de comparação na classe Conta.

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     public boolean compara(Conta outra) {
6         return this.numero == outra.numero;
7     }
8
9     // GETTERS E SETTERS
10}

```

*Código Java 9.14: Conta.java*

A utilização do método compara() seria mais ou menos assim:

```

1 Conta c1 = ...
2 Conta c2 = ...
3
4 System.out.println(c1.compara(c2));

```

*Código Java 9.15: Comparando com compara()*

Contudo, na classe Object, já existe um método com o mesmo propósito. O método ao qual nos referimos é o equals(). A implementação padrão do método equals() na classe Object delega a comparação ao operador “==”. Dessa forma, o conteúdo dos objetos não é comparado. Podemos rescrever o método equals() para alterar esse comportamento e passar a considerar o conteúdo dos objetos na comparação.

```

1 class Conta {
2     private int numero;
3     private double saldo;
4
5     public boolean equals(Object obj) {
6         Conta outra = (Conta)obj;
7         return this.numero == outra.numero;
8     }
9 }

```

```
10 // GETTERS E SETTERS
11 }
```

Código Java 9.16: Conta.java

Porém, a reescrita do método `equals()` deve respeitar diversas regras definidas na documentação da classe `Object`(<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>).

Para não infringir nenhuma das regras de reescrita do método `equals()`, podemos utilizar recursos do eclipse para gerar esse método automaticamente.

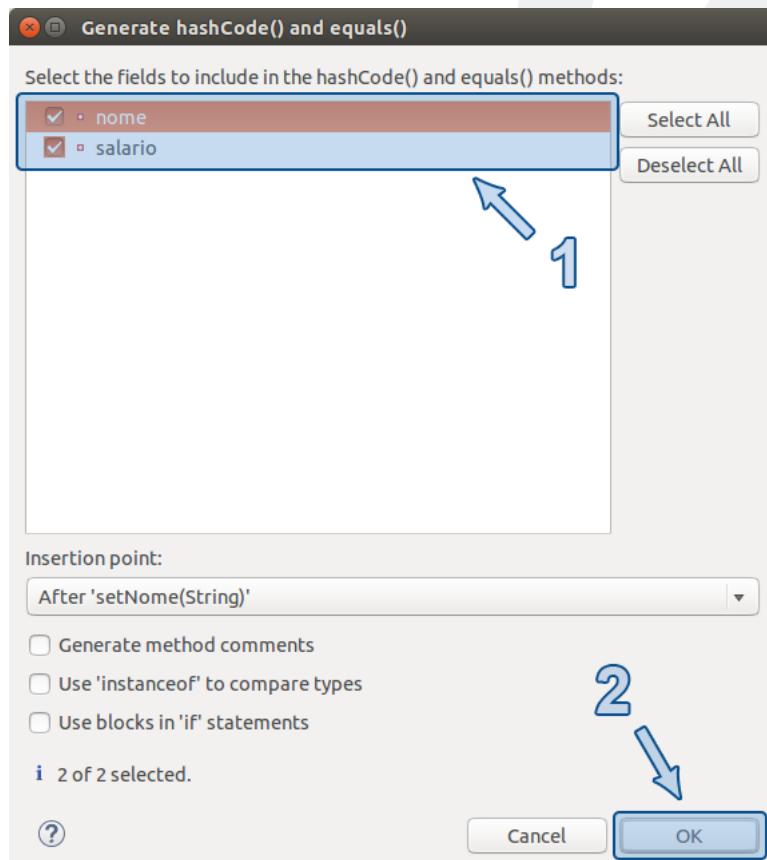


Figura 9.3: Gerando o equals() no eclipse

```
1 class Conta {
2     private int numero;
3     private double saldo;
4
5     // método gerado pelo eclipse
6     public int hashCode() {
7         final int prime = 31;
8         int result = 1;
9         result = prime * result + numero;
10        return result;
11    }
12
13    // método gerado pelo eclipse
14    public boolean equals(Object obj) {
15        if (this == obj) {
16            return true;
17        }
18    }
19}
```

```
18     if (obj == null) {  
19         return false;  
20     }  
21     if (!(obj instanceof Conta)) {  
22         return false;  
23     }  
24     Conta other = (Conta) obj;  
25     if (numero != other.numero) {  
26         return false;  
27     }  
28     return true;  
29 }  
30 // GETTERS E SETTERS  
31  
32 }
```

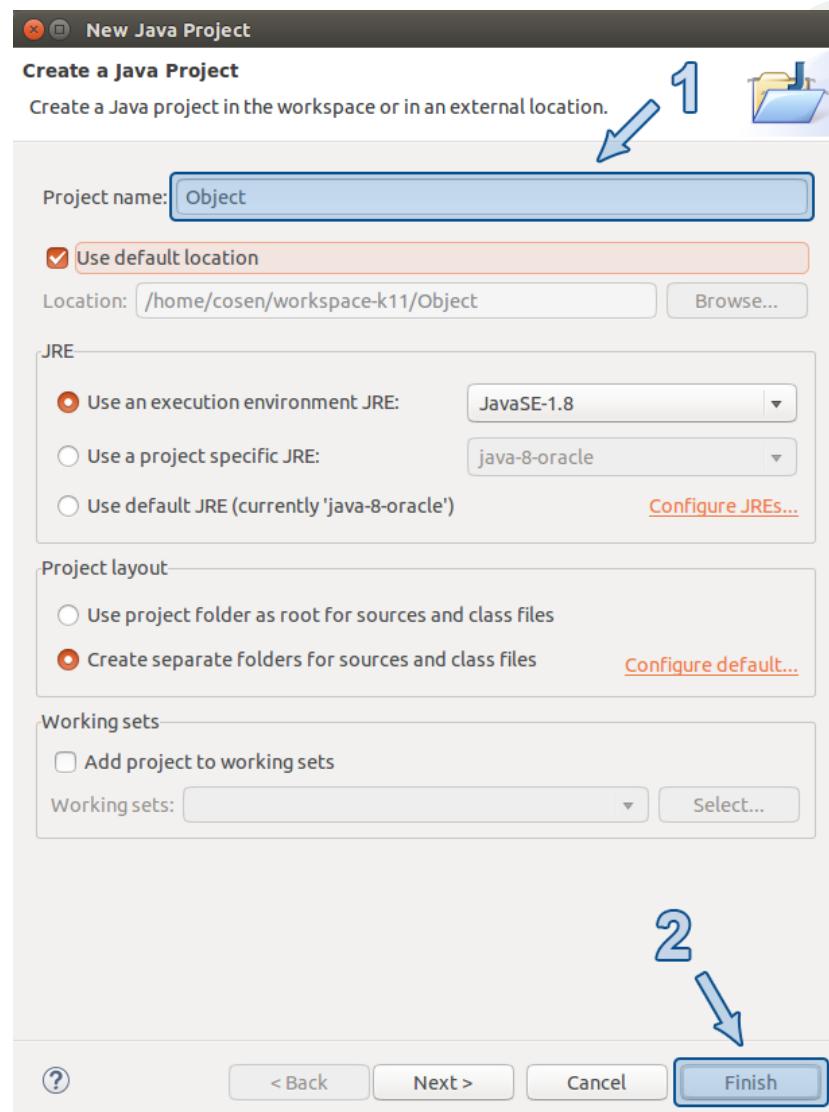
Código Java 9.17: Conta.java



## Exercícios de Fixação

---

- 1 Para não confundir, feche o projeto **Polimorfismo** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



2 Crie uma classe chamada **Funcionario**.

```
1 package br.com.k19.banco.sistema;
2
3 public class Funcionario {
4     private String nome;
5
6     private double salario;
7
8     // GETTERS E SETTERS
9 }
```

Código Java 9.18: Funcionario.java

3 Crie um objeto da classe Funcionario e exiba a referência desse objeto no Console.

```
1 package br.com.k19.banco.testes;
2
3 import br.com.k19.banco.sistema.Funcionario;
```

```

4
5 public class TestaFuncionario {
6     public static void main(String[] args) {
7         Funcionario f = new Funcionario();
8
9         f.setNome("Jonas Hirata");
10        f.setSalario(3000);
11
12        System.out.println(f);
13    }
14 }
```

*Código Java 9.19: TestaFuncionario.java*

Execute a classe `TestaFuncionario`.

- 4 Reescreva o método `toString()` na classe `Funcionario` para alterar a descrição textual dos objetos que representam os funcionários.

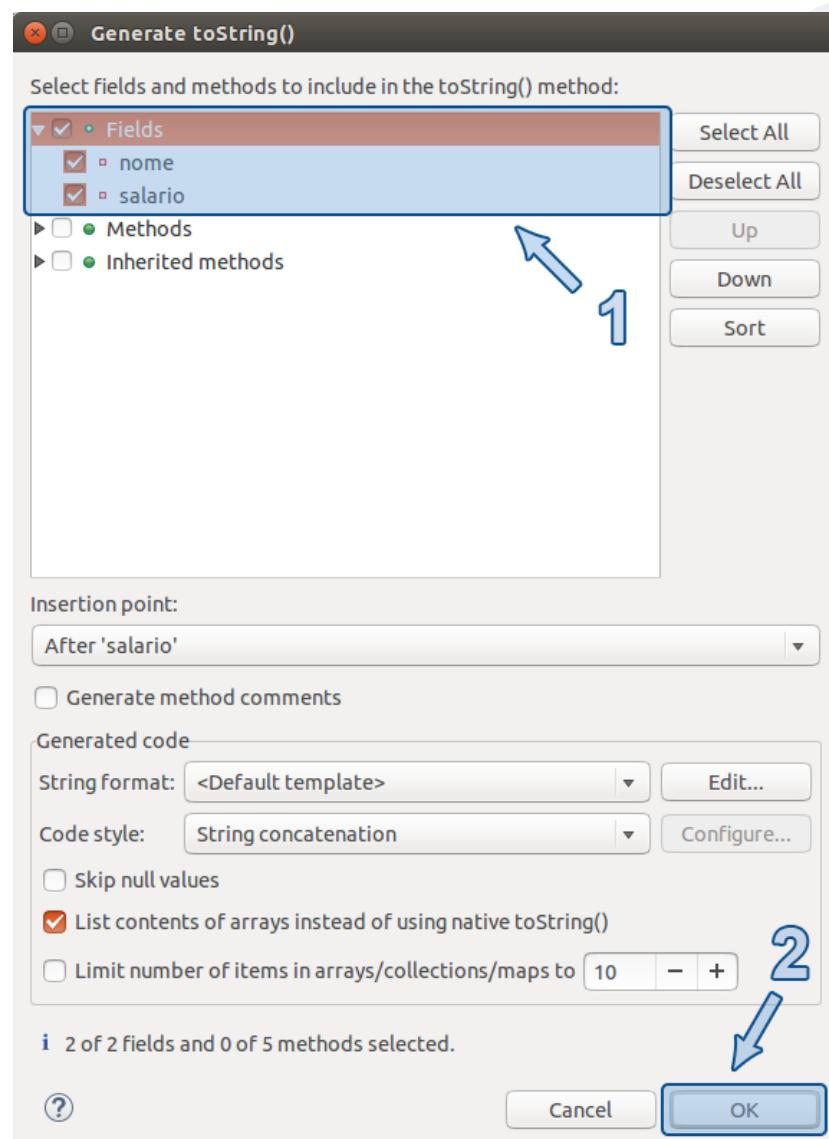
```

1 package br.com.k19.banco.sistema;
2
3 public class Funcionario {
4     private String nome;
5
6     private double salario;
7
8     public String toString() {
9         return "Funcionário: " + this.nome + " - Salário: " + this.salario;
10    }
11
12    // GETTERS E SETTERS
13 }
```

*Código Java 9.20: Funcionario.java*

- 5 Execute novamente a classe `TestaFuncionario`.

- 6 Apague o `toString()` implementando anteriormente. Utilize os recursos do eclipse para reescrever esse método automaticamente na classe `Funcionario`. Selecione a classe `Funcionario`, digite “`CTRL + 3`” e pesquise por “Generate `toString()`”. Selecione a opção correspondente e siga a imagem abaixo.



- 7 Execute novamente a classe **TestaFuncionario**.
- 8 Crie dois objetos da classe **Funcionario**. Utilize o operador “==” e o método **equals()** para compará-los. Adicione uma classe chamada **TestaEquals**.

```

1 package br.com.k19.banco.testes;
2
3 import br.com.k19.banco.sistema.Funcionario;
4
5 public class TestaEquals {
6     public static void main(String[] args) {
7         Funcionario f1 = new Funcionario();
8
9         f1.setNome("Jonas Hirata");
10        f1.setSalario(3000);
11
12        Funcionario f2 = new Funcionario();
13
14        f2.setNome("Jonas Hirata");

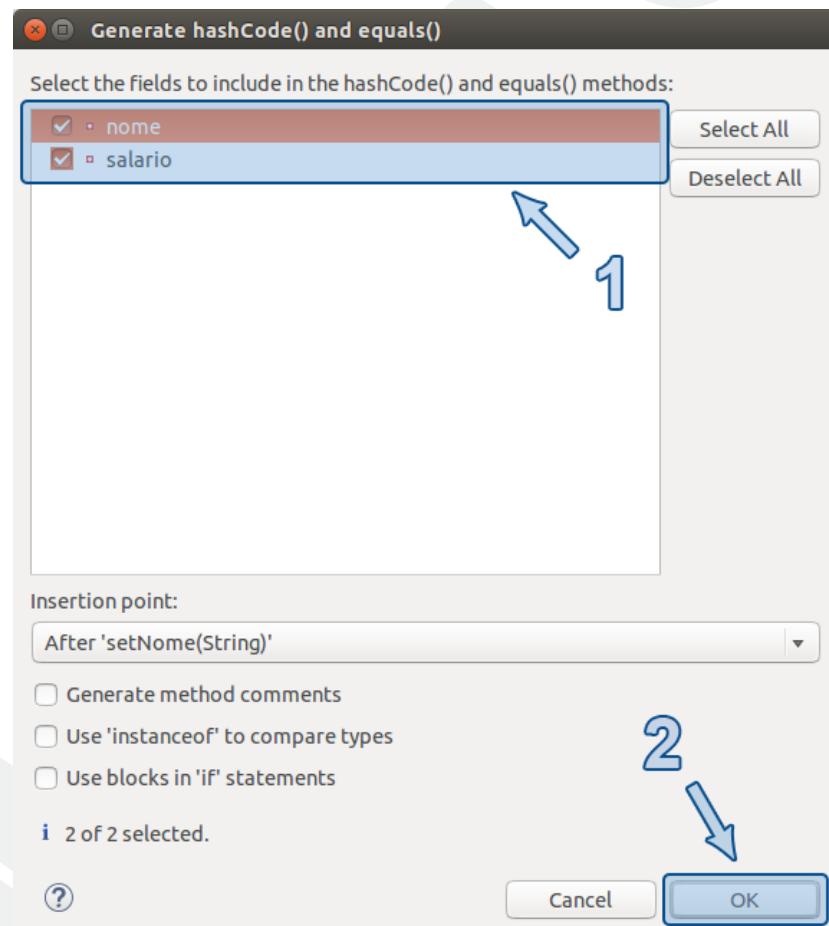
```

```
15     f2.setSalario(3000);
16
17     System.out.println(f1 == f2);
18     System.out.println(f1.equals(f2));
19 }
20 }
```

Código Java 9.21: TestaEquals.java

Execute a classe TestaEquals.

- 9 Utilize os recursos do eclipse para gerar o método equals() na classe Funcionario. Selecione a classe Funcionario, digite “CTRL + 3” e pesquise por “Generate hashCode() and equals()”. Selecione a opção correspondente e siga a imagem abaixo.



- 10 Execute novamente a classe TestaEquals.



# CLASSE ABSTRATAS



## Classes Abstratas

No banco, todas as contas são de um tipo específico. Por exemplo, conta poupança, conta corrente ou conta salário. Essas contas poderiam ser modeladas através das seguintes classes utilizando o conceito de herança:

```
1 class Conta {  
2     // Atributos  
3     // Construtores  
4     // Métodos  
5 }
```

Código Java 10.1: Conta.java

```
1 class ContaPoupanca extends Conta {  
2     // Atributos  
3     // Construtores  
4     // Métodos  
5 }
```

Código Java 10.2: ContaPoupanca.java

```
1 class ContaCorrente extends Conta {  
2     // Atributos  
3     // Construtores  
4     // Métodos  
5 }
```

Código Java 10.3: ContaCorrente.java

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();
```

Código Java 10.4: Criando um objeto da classe 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 concretas.

Não faz sentido criar um objeto da classe `Conta` pois estariamos instanciando 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 tornar a classe `Conta` **abstrata**.

Uma classe concreta pode ser diretamente utilizada para instanciar objetos. Por outro lado, uma classe abstrata não pode. Para definir uma classe abstrata, basta adicionar o modificador **abstract**.

```
1 abstract class Conta {  
2     // Atributos  
3     // Construtores  
4     // Métodos  
5 }
```

*Código Java 10.5: Conta.java*

Todo código que tenta criar um objeto de uma classe abstrata não compila.

```
1 // Erro de compilação  
2 Conta c = new Conta();
```

*Código Java 10.6: Erro de compilação*



## 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, parece não fazer sentido ter um método na classe `Conta` para gerar extratos detalhados pois ele seria reescrito nas classes específicas sem nem ser reaproveitado.

Poderíamos, simplesmente, não definir nenhum método para gerar extratos detalhados na classe `Conta`. Porém, não haveria nenhuma garantia que as classes que derivam direta ou indiretamente da classe `Conta` implementem métodos para gerar extratos detalhados.

Mas, mesmo supondo que toda classe derivada implemente um método para gerar os extratos que desejamos, ainda não haveria nenhuma garantia em relação as assinaturas desses métodos. As classes derivadas poderiam definir métodos com nomes ou parâmetros diferentes. Isso prejudicaria a utilização dos objetos que representam as contas devido a falta de padronização das operações.

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 não possui corpo (implementação).

```
1 abstract class Conta {  
2     // Atributos  
3     // Construtores  
4     // Métodos  
5  
6     public abstract void imprimeExtratoDetalhado();
```

7 }

*Código Java 10.7: Conta.java*

As classes concretas que derivam direta ou indiretamente da classe Conta devem possuir uma implementação para o método `imprimeExtratoDetalhado()`.

```

1 class ContaPoupanca extends Conta {
2     private int diaDoAniversario;
3
4     public void imprimeExtratoDetalhado() {
5         System.out.println("EXTRATO DETALHADO DE CONTA POUPANÇA");
6
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: " + this.getSaldo());
12        System.out.println("ANIVERSÁRIO: " + this.diaDoAniversario);
13    }
14 }
```

*Código Java 10.8: ContaPoupanca.java*

Se uma classe concreta derivada da classe Conta não possuir uma implementação do método `imprimeExtratoDetalhado()` ela não compilará.

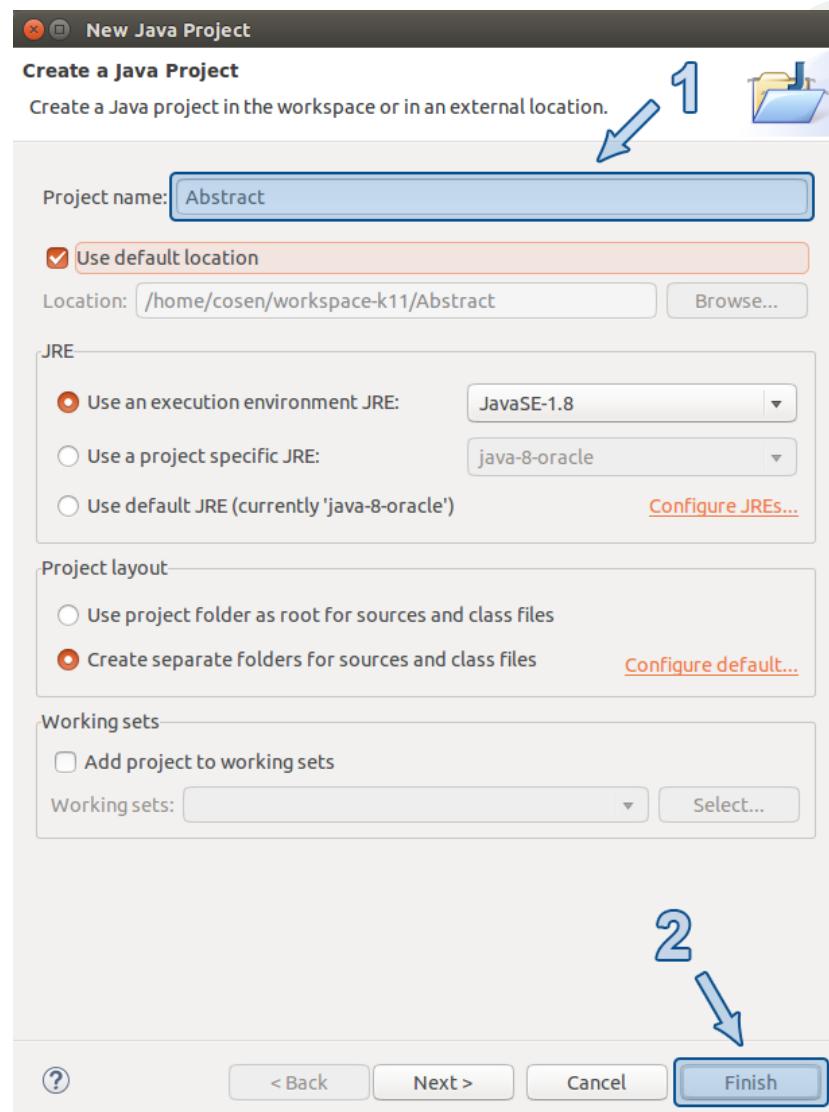
```

1 // ESSA CLASSE NÃO COMPILA
2 class ContaPoupanca extends Conta {
3
4 }
```

*Código Java 10.9: ContaPoupanca.java*

## Exercícios de Fixação

- Para não confundir, feche o projeto **Object** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe chamada **Conta** para modelar as contas de um banco.

```

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

Código Java 10.10: Conta.java

- 3 Crie um teste simples para utilizar objetos da classe Conta.

```

1 public class TestaConta {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         c.setSaldo(1000);
6
7         System.out.println(c.getSaldo());
```

```
8 } }
```

*Código Java 10.11: TestaConta.java*

Execute a classe TestaConta.

- 4 Torne a classe Conta abstrata e verifique o que acontece na classe TestaConta.

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

*Código Java 10.12: Conta.java*

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

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

*Código Java 10.13: ContaPoupanca.java*

- 6 Altere a classe TestaConta para corrigir o erro de compilação.

```
1 public class TestaConta {  
2     public static void main(String[] args) {  
3         Conta c = new ContaPoupanca();  
4  
5         c.setSaldo(1000);  
6  
7         System.out.println(c.getSaldo());  
8     }  
9 }
```

*Código Java 10.14: TestaConta.java*

Execute a classe TestaConta.

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

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

*Código Java 10.15: Conta.java*

- 8** Verifique o erro de compilação na classe ContaPoupanca.
- 9** Defina uma implementação do método `imprimeExtratoDetalhado()` na classe ContaPoupanca.

```

1 import java.text.SimpleDateFormat;
2 import java.util.Date;
3
4 public class ContaPoupanca extends Conta {
5     private int diaDoAniversario;
6
7     public void imprimeExtratoDetalhado() {
8         System.out.println("EXTRATO DETALHADO DE CONTA POUPANÇA");
9
10        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
11        Date agora = new Date();
12
13        System.out.println("DATA: " + sdf.format(agora));
14        System.out.println("SALDO: " + this.getSaldo());
15        System.out.println("ANIVERSÁRIO: " + this.diaDoAniversario);
16    }
17 }
```

Código Java 10.16: ContaPoupanca.java

- 10** Altere a classe TestaConta para chamar o método `imprimeExtratoDetalhado()`.

```

1 public class TestaConta {
2     public static void main(String[] args) {
3         Conta c = new ContaPoupanca();
4
5         c.setSaldo(1000);
6
7         c.imprimeExtratoDetalhado();
8     }
9 }
```

Código Java 10.17: TestaConta.java

Execute a classe TestaConta.



## Exercícios Complementares

- 1** Defina uma classe chamada **Funcionario** para modelar os funcionários de um banco.
- 2** Crie um objeto da classe Funcionario e utilize os métodos de acesso com nomes padronizados para alterar os valores dos atributos desse objeto. Faça uma classe chamada **TestaFuncionario**. Por fim, execute essa classe.
- 3** Adicione o modificador `abstract` na classe Funcionario. Verifique o erro de compilação na classe TestaFuncionario.
- 4** Defina uma classe chamada **Gerente** para modelar os gerentes do banco. Considere que os

gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. Além disso, considere que todo gerente é um funcionário.

- 5 Altere a classe TestaFuncionario e crie um objeto da classe Gerente no lugar do objeto da classe Funcionario. Por fim, execute a classe TestaFuncionario.
- 6 Defina um método abstrato na classe Funcionario chamado **calculaBonificacao** para calcular a bonificação dos colaboradores.
- 7 Verifique o erro de compilação na classe Gerente.
- 8 Implemente o método **calculaBonificacao** na classe Gerente. Considere que a bonificação dos gerentes é 20% do salário mais 300 reais.
- 9 Altere a classe TestaFuncionario para que o método **calculaBonificacao** seja chamada e o valor seja exibido no Console. Por fim, execute a classe TestaFuncionario.



## INTERFACES



### Padronização

No dia a dia, estamos acostumados a utilizar aparelhos que dependem de energia elétrica. Esses aparelhos possuem um plugue que deve ser conectado a uma tomada para obter a energia necessária.

Diversas empresas fabricam aparelhos elétricos com plugues. Analogamente, diversas empresas fabricam tomadas elétricas. Suponha que cada empresa decida por conta própria o formato dos plugues ou das tomadas que fabricará. Teríamos uma infinidade de tipos de plugues e tomadas que tornaria a utilização dos aparelhos elétricos uma experiência extremamente desagradável.

Inclusive, essa falta de padrão pode gerar problemas de segurança aos usuários. Os formatos dos plugues ou das tomadas pode aumentar o risco de uma pessoa tomar um choque elétrico.

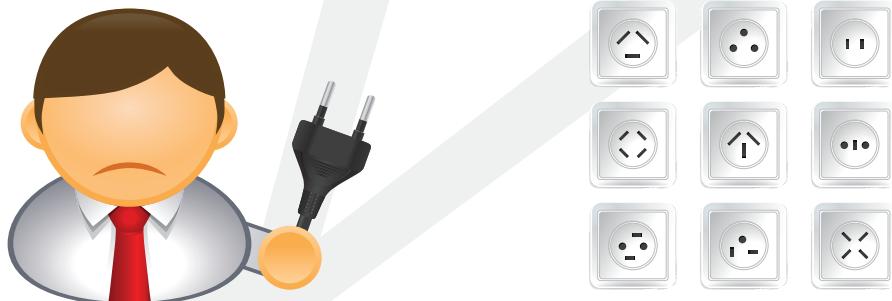


Figura 11.1: Tomadas despadrонizadas

Com o intuito de facilitar a utilização dos consumidores e aumentar a segurança dos mesmos, o governo através dos órgãos responsáveis estabelece padrões para os plugues e tomadas. Esses padrões estabelecem restrições que devem ser respeitadas pelos fabricantes dos aparelhos e das tomadas.

Em diversos contextos, padronizar pode trazer grandes benefícios. Inclusive, no desenvolvimento de aplicações. Mostraremos como a ideia de padronização pode ser contextualizada nos conceitos de orientação a objetos.



### Contratos

Num sistema orientado a objetos, os objetos interagem entre si através de chamadas de métodos (troca de mensagens). Podemos dizer que os objetos se “encaixam” através dos métodos públicos

assim como um plugue se encaixa em uma tomada através dos pinos.

Para os objetos de uma aplicação “conversarem” entre si mais facilmente é importante padronizar o conjunto de métodos oferecidos por eles. Assim como os plugues encaixam nas tomadas mais facilmente graças aos padrões definidos pelo governo.

Um padrão é definido através de especificações ou contratos. Nas aplicações orientadas a objetos, podemos criar um “contrato” para definir um determinado conjunto de métodos que deve ser implementado pelas classes que “assinarem” este contrato. Em orientação a objetos, um contrato é chamado de **interface**. Um interface é composta basicamente por métodos abstratos.



## 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 }
```

Código Java 11.1: Conta.java

Os métodos de uma interface não possuem corpo (implementação) pois serão implementados nas classes vinculadas a essa interface. Todos os métodos de uma interface devem ser 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 }
```

Código Java 11.2: ContaPoupanca.java

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

Código Java 11.3: ContaCorrente.java

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 }
```

Código Java 11.4: ContaCorrente.java

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.



## Polimorfismo

Se uma classe implementa uma interface, podemos aplicar a ideia do polimorfismo assim como quando aplicamos 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();
```

Código Java 11.5: Polimorfismo

Além disso, podemos passar uma variável do tipo ContaCorrente para um método que o parâmetro seja do tipo Conta.

```

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

Código Java 11.6: GeradorDeExtrato.java

```

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

Código Java 11.7: Aproveitando o polimorfismo

O método geraExtrato() pode ser utilizado para objetos criados a partir de classes que implementam direta ou indiretamente a interface Conta.



## Interface e Herança

As vantagens e desvantagens entre interface e herança, provavelmente, é um dos temas 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 desse tópico. Muitas pessoas se posicionam de forma radical defendendo a utilização de interfaces ao invés

de herança em qualquer situação.

Normalmente, esses debates são direcionados na análise do que é melhor para manutenção das aplicações: 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.

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.

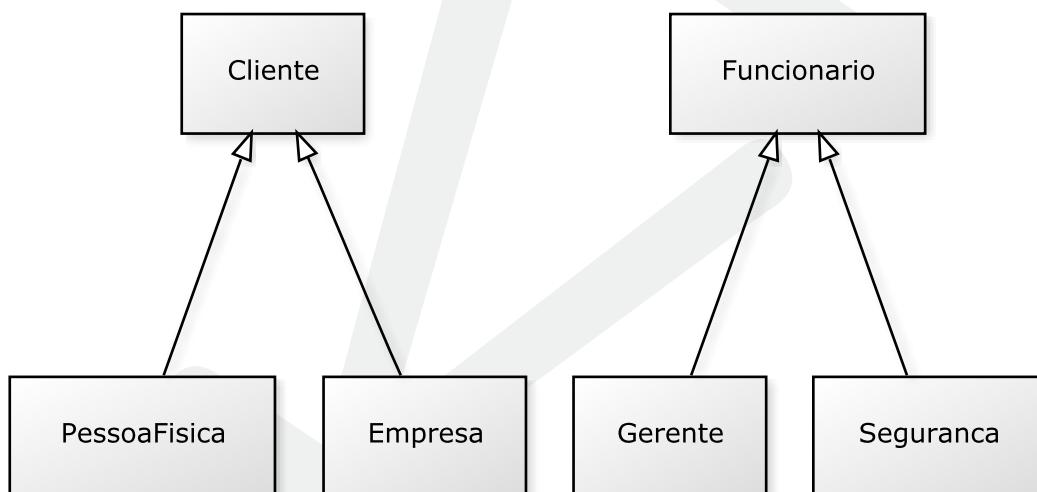


Figura 11.2: Duas árvores de herança independentes

Suponha que os gerentes e as empresas possam acessar o sistema do banco com 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 }
  
```

Código Java 11.8: AutenticadorDeUsuario.java

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 o polimorfismo desejado.

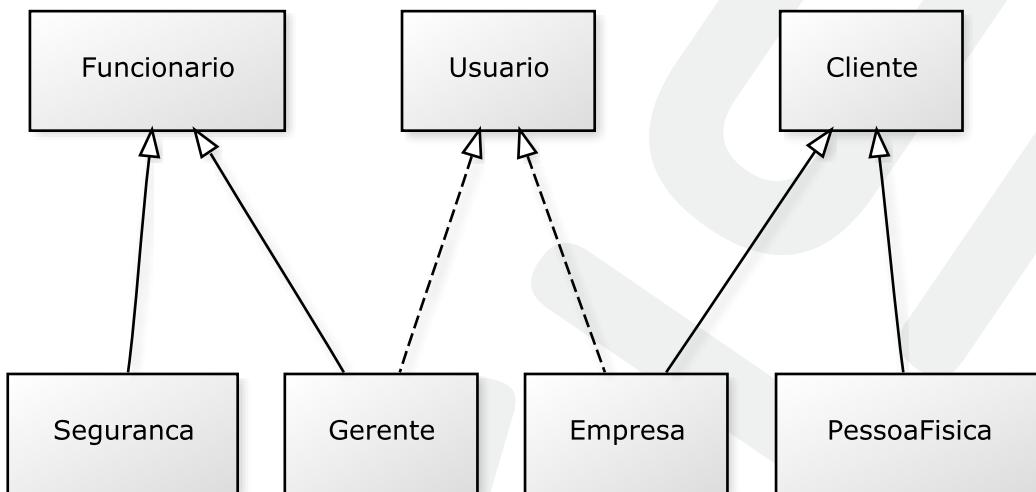


Figura 11.3: Obtendo mais polimorfismo

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.

```

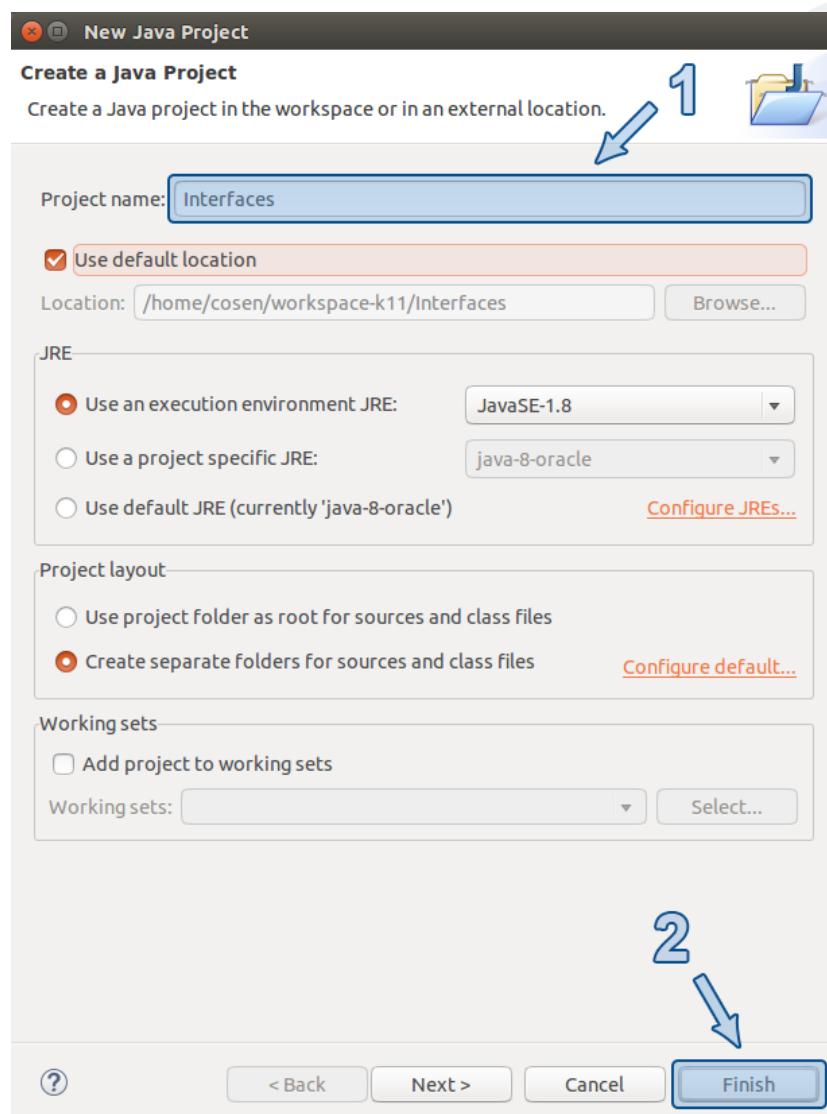
1 class AutenticadorDeUsuario {
2     public boolean autentica(Usuario u) {
3         // implementação
4     }
5 }
  
```

Código Java 11.9: AutenticadorDeUsuario.java



## Exercícios de Fixação

- Para não confundir, feche o projeto **Abstract** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma interface chamada **Conta** para padronizar as assinaturas dos métodos das contas de um banco.

```

1 public interface Conta {
2     void deposita(double valor);
3
4     void saca(double valor);
5
6     double getSaldo();
7 }
```

Código Java 11.10: Conta.java

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

```

1 public class ContaCorrente implements Conta {
2     private double saldo;
3     private double taxaPorOperacao = 0.45;
4 }
```

```

5  public void deposita(double valor) {
6      this.saldo += valor - this.taxaPorOperacao;
7  }
8
9  public void saca(double valor) {
10     this.saldo -= valor + this.taxaPorOperacao;
11 }
12
13 public double getSaldo() {
14     return this.saldo;
15 }
16 }
```

*Código Java 11.11: ContaCorrente.java*

```

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

*Código Java 11.12: ContaPoupanca.java*

- 4 Faça um teste simples com as classes criadas anteriormente.

```

1 public class TestaContas {
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         c1.saca(100);
10        c2.saca(100);
11
12        System.out.println(c1.getSaldo());
13        System.out.println(c2.getSaldo());
14    }
15 }
```

*Código Java 11.13: TestaContas.java*

Execute a classe TestaContas.

- 5 Altere a assinatura do método deposita() na classe ContaCorrente. Você pode acrescentar um “r” no final do nome desse 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.

Adicione uma classe chamada **GeradorDeExtrato** com um método chamado **geraExtrato**.

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

*Código Java 11.14: GeradorDeExtrato.java*

- 7 Teste o gerador de extrato. Crie uma classe chamada **TestaGeradorDeExtrato** para isso.

```
1 public 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 }
```

*Código Java 11.15: TestaGeradorDeExtrato.java*

# PACOTES



## Organização

O código fonte de uma aplicação é definido em diversos arquivos. Conforme a quantidade de arquivos cresce surge a necessidade de algum tipo de organização para poder encontrar os arquivos rapidamente quando for necessário modificá-los.

A ideia para organizar logicamente os arquivos de uma aplicação é bem simples e as pessoas que utilizam computadores já devem estar familiarizadas. Os arquivos são separados em pastas ou diretórios.



## O comando package

Na terminologia do Java, as pastas nas quais são organizadas as classes e interfaces de uma aplicação são chamadas de **pacotes**.

Para colocar uma classe ou interface em um pacote, devemos realizar dois passos:

1. Utilizar o comando **package** na primeira linha de código do arquivo contendo a classe ou interface que desejamos colocar em um determinado pacote.

```
1 package sistema;
2
3 class Conta {
4     // corpo da classe
5 }
```

Código Java 12.1: Conta.java

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

```
K19/workspace/Pacotes/src/sistema$ ls
Conta.java
```

Terminal 12.1: Salvando o arquivo na pasta com o mesmo do pacote

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



## 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 }
```

Código Java 12.2: Conta.java

Além disso, devemos criar uma estrutura de pastas que reflita os sub-pacotes definidos no código fonte.

```
K19/workspace/Pacotes/src/sistema/contas$ ls
Conta.java
```

Terminal 12.2: sub-pacotes



## Unqualified Name vs Fully Qualified Name

Com a utilização de pacotes é apropriado definir o que é o nome simples (**unqualified name**) 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 caractere “.”.

Por exemplo, suponha a seguinte código:

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

Código Java 12.3: Conta.java

O nome simples da classe acima é: `Conta` e o nome completo é: `sistema.contas.Conta`.



## Classes ou Interfaces públicas

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 as seguintes classes:

```
K19/workspace/Pacotes/src/sistema/contas$ ls
Conta.java ContaPoupanca.java
```

Terminal 12.3: Conta.java e ContaPoupanca.java

```

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

*Código Java 12.4: Conta.java*

```

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

*Código Java 12.5: ContaPoupanca.java*

Observe que a classe ContaPoupanca utiliza o nome simples da classe Conta para acessá-la.

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 as seguintes classes:

```

K19/workspace/Pacotes/src/sistema$ ls
contas clientes

K19/workspace/Pacotes/src/sistema$ ls contas/
Conta.java

K19/workspace/Pacotes/src/sistema$ ls clientes/
Cliente.java
```

*Terminal 12.4: Conta.java e Cliente.java*

```

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

*Código Java 12.6: Conta.java*

```

1 package sistema.clientes;
2
3 class Cliente {
4     private sistema.contas.Conta conta;
5 }
```

*Código Java 12.7: Cliente.java*

## 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.

```

1 // Arquivo: Cliente.java
2 package sistema.clientes;
3
4 import sistema.contas.Conta;
```

```
6 class Cliente {  
7     private Conta conta;  
8 }
```

*Código Java 12.8: Cliente.java*

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.



## Conflito de nomes

A reutilização é um dos principais argumentos para utilização do modelo orientado a objetos e da plataforma Java. Há muitas bibliotecas disponíveis para utilizarmos em nossas aplicações. Contudo, certos cuidados com os nomes dos pacotes são necessários para evitar conflito entre as classes e interfaces das nossas aplicações e as classes e interfaces das bibliotecas.

Com o intuito de resolver esse problema, há uma convenção para a definição dos nomes dos pacotes. Essa convenção é análoga aos domínios da internet.

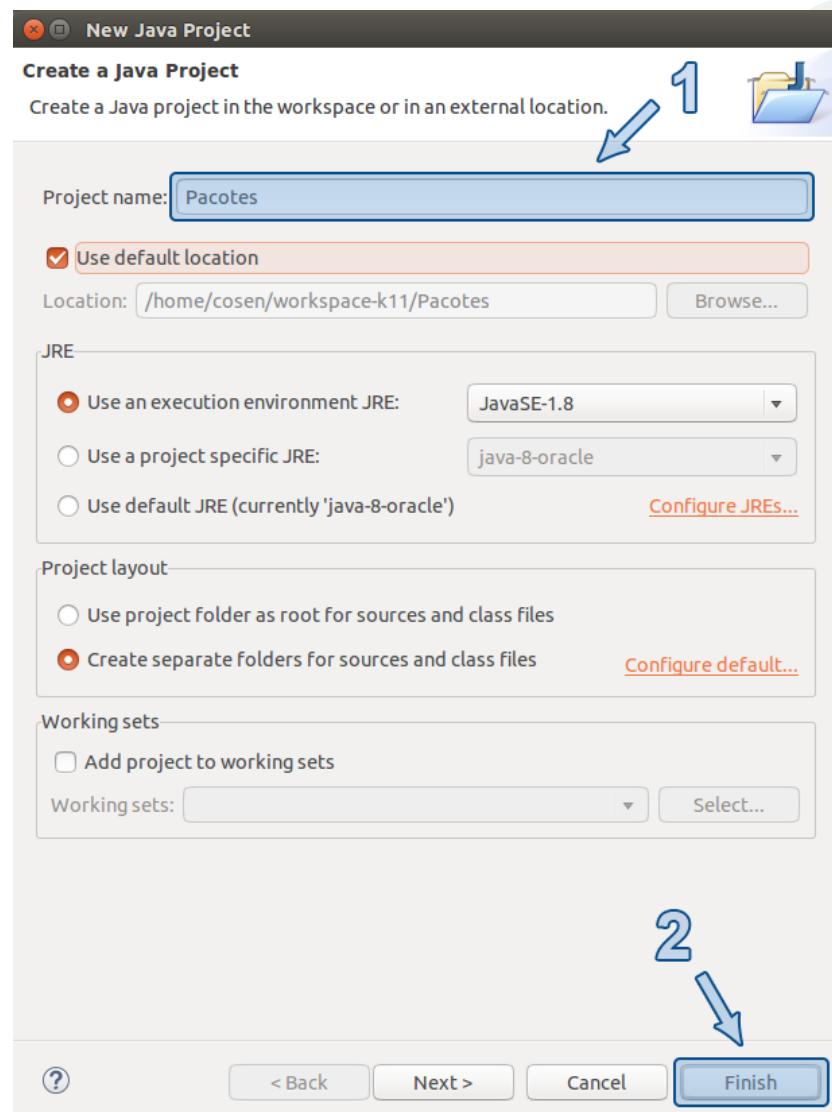
```
1 package br.com.k19.sistema.contas;
```

*Código Java 12.9: Padrão de nomenclatura de pacotes*



## Exercícios de Fixação

- Para não confundir, feche o projeto **Interfaces** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie um pacote chamado **br.com.k19.banco.sistema** e outro chamado **br.com.k19.banco.testes**.
- 3 Faça uma classe chamada **Conta** para modelar as contas de um banco. Essa classe deve ser colocada no pacote **br.com.k19.banco.sistema**.

```

1 package br.com.k19.banco.sistema;
2
3 public class Conta {
4     private double saldo;
5
6     public void deposita(double valor) {
7         this.saldo += valor;
8     }
9
10    // GETTERS AND SETTERS
11 }
```

Código Java 12.10: Conta.java

- 4 Faça uma classe chamada **Teste** no pacote **br.com.k19.banco.testes**.

```
1 package br.com.k19.banco.testes;
2
3 import br.com.k19.banco.sistema.Conta;
4
5 public class Teste {
6     public static void main(String[] args) {
7         Conta c = new Conta();
8         c.deposita(1000);
9         System.out.println(c.getSaldo());
10    }
11 }
```

*Código Java 12.11: Teste.java*

- 5 Retire o modificador **public** da classe **Conta** e observe o erro de compilação na classe **Teste**. Depois, faça a classe **Conta** ser pública novamente.

## 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         return 0; // sucesso  
7     }  
8 }
```

Código Java 13.1: Utilizando códigos de erro

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 à lógica natural do método. Não é possível fazer as duas coisas sem nenhum tipo de “gambiarras”.

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

Código Java 13.2: Código de erro e retorno lógico

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 distintos 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 lógicos dos métodos.



### Errors vs Exceptions

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 subclass `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 subclass `Exception` define erros para os quais as aplicações normalmente têm condições de definir um tratamento.

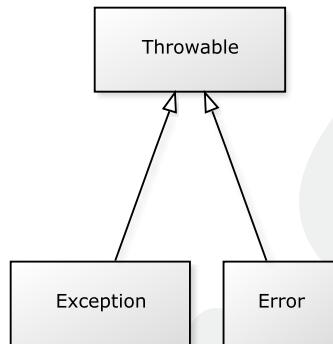


Figura 13.1: Tipos de erros de execução



## 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`.

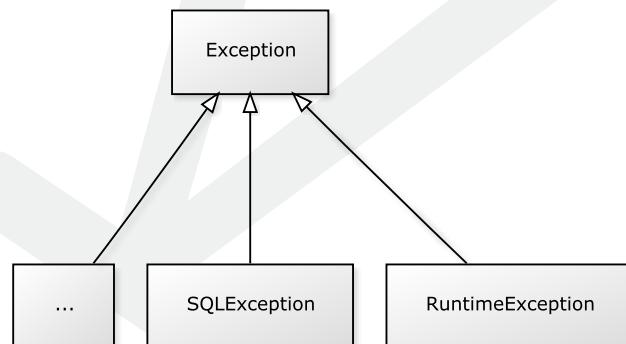


Figura 13.2: Tipos de erros de execução

As classes que estão na árvore da classe `Exception` mas não estão na árvore da `RuntimeException` são as chamadas **checked exceptions**. Por outro lado, as classes que estão na árvore da `RuntimeException` são as chamadas **unchecked exceptions**.



## Lançando uma unchecked exception

Quando identificamos um erro, podemos criar um objeto de alguma unchecked exception e “lançar” a referência dele com o comando `throw`.

Observe o exemplo abaixo que utiliza a classe `IllegalArgumentException` que deriva diretamente da classe `RuntimeException`.

```

1 public void deposita(double valor) {
2     if(valor < 0) {
3         IllegalArgumentException erro = new IllegalArgumentException();
4         throw erro;
5     } else {
6         ...
7     }
8 }
```

*Código Java 13.3: Lançado uma unchecked exception*



## Lançando uma checked exception

Quando identificamos um erro, podemos criar um objeto de alguma checked exception e “lançar” a referência dele com o comando `throw`. Contudo, antes de lançar uma checked exception, é necessário determinar de maneira explícita através do comando `throws` que o método pode lançar esse tipo de erro.

Observe o exemplo abaixo que utiliza a classe `Exception`.

```

1 public void deposita(double valor) throws Exception {
2     if(valor < 0) {
3         Exception erro = new Exception();
4         throw erro;
5     } else {
6         ...
7     }
8 }
```

*Código Java 13.4: Lançado uma checked exception*



## Capturando exceptions

Quando queremos capturar exceptions, devemos utilizar o comando `try-catch`.

```

1 class Teste {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         try {
6             c.deposita(100);
7         } catch (IllegalArgumentException e) {
8             System.out.println("Houve um erro ao depositar");
9         }
10    }
11 }
```

*Código Java 13.5: Capturando um tipo de exceptions*

Podemos encadear vários blocos `catch` para capturar exceptions de classes diferentes.

```

1 class Teste {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         try {
```

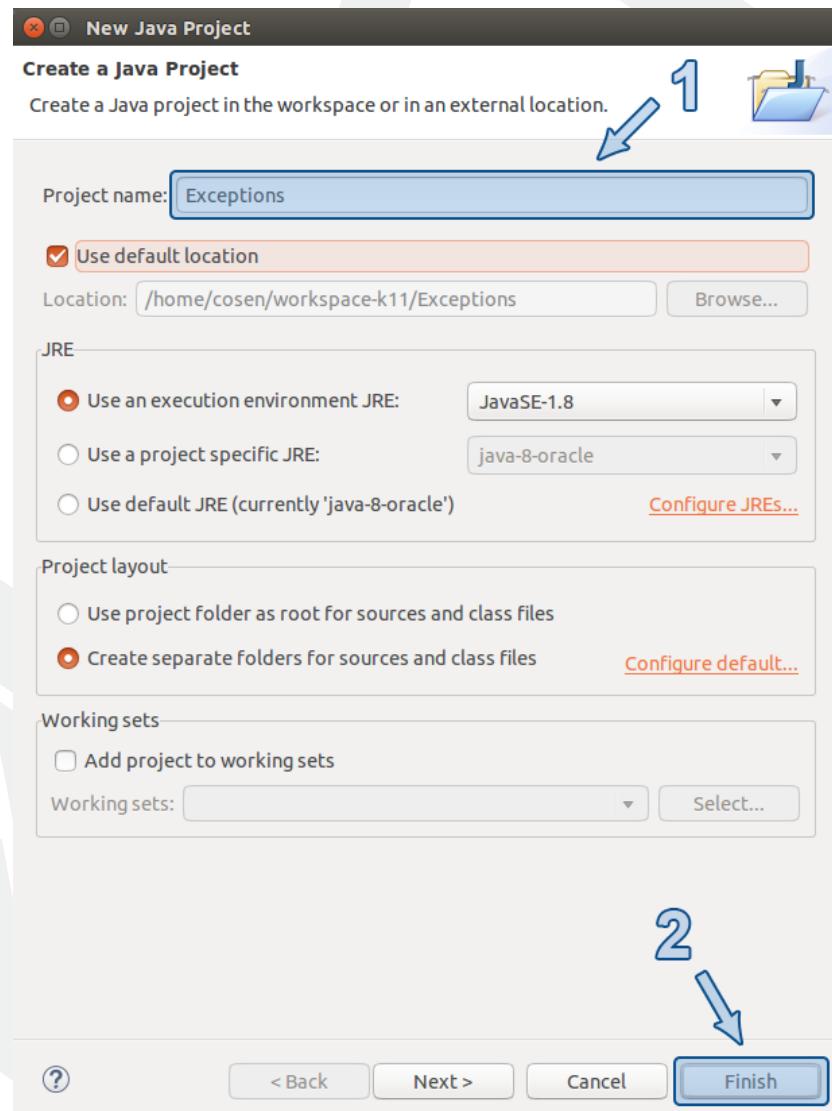
```
6     c.deposita(100);
7 } catch (IllegalArgumentException e) {
8     System.out.println("Houve uma IllegalArgumentException ao depositar");
9 } catch (SQLException e) {
10    System.out.println("Houve uma SQLException ao depositar");
11 }
12 }
13 }
```

Código Java 13.6: Capturando dois tipos de exceptions



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Pacotes** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



**2** Crie uma classe chamada **Funcionario**.

```

1 package br.com.k19.banco.sistema;
2
3 public class Funcionario {
4     private double salario;
5
6     public void aumentaSalario(double aumento) {
7         if (aumento < 0) {
8             IllegalArgumentException erro = new IllegalArgumentException();
9             throw erro;
10        }
11    }
12
13    // GETTERS E SETTERS
14 }
```

*Código Java 13.7: Funcionario.java*

**3** Crie uma classe chamada **TestaFuncionario**.

```

1 package br.com.k19.banco.testes;
2
3 import br.com.k19.banco.sistema.Funcionario;
4
5 public class TestaFuncionario {
6     public static void main(String[] args) {
7         Funcionario f = new Funcionario();
8         f.aumentaSalario(-1000);
9     }
10 }
```

*Código Java 13.8: TestaFuncionario.java*

Execute a classe **TestaFuncionario** e observe o erro no Console.

**4** Altere a classe **TestaFuncionario** para capturar o erro.

```

1 package br.com.k19.banco.testes;
2
3 import br.com.k19.banco.sistema.Funcionario;
4
5 public class TestaFuncionario {
6     public static void main(String[] args) {
7         Funcionario f = new Funcionario();
8         try {
9             f.aumentaSalario(-1000);
10        } catch (IllegalArgumentException e) {
11            System.out.println("Houve uma IllegalArgumentException ao aumentar o salário");
12        }
13    }
14 }
```

*Código Java 13.9: TestaFuncionario.java*



# STRING

A classe `String` é utilizada em praticamente todas as aplicações Java. Consequentemente, os programadores Java devem conhecer bem o funcionamento dela. A documentação da classe `String` pode ser consultada na url <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>.



## Pool de Strings

A maneira mais simples para criar ou obter um objeto da classe `String` é definir diretamente o seu conteúdo no código fonte com "".

```
1 String nome = "Rafael Cosentino";
```

*Código Java 14.1: string literal*

As strings criadas ou obtidas de forma literal são mantidas dentro da classe `String` no chamado **Pool de Strings**. Dentro do Pool de Strings, não há strings “repetidas”, ou seja, não há dois objetos com o mesmo conteúdo. Esse aspecto da linguagem Java tem por objetivo diminuir o consumo de memória.

Para comparar o conteúdo de duas strings que estão dentro do Pool de Strings, podemos utilizar o operador “==” ou o método `equals()`.

```
1 String nome1 = "Rafael Cosentino";
2 String nome2 = "Rafael Cosentino";
3
4 // imprime true
5 System.out.println(nome1 == nome2);
6 // imprime true
7 System.out.println(nome1.equals(nome2));
```

*Código Java 14.2: Comparando strings do Pool de Strings*

Também podemos utilizar os construtores da classe `String` para criar objetos.

```
1 String nome = new String("Rafael Cosentino");
```

*Código Java 14.3: strings não literais*

Quando criamos strings através dos construtores da classe `String`, elas não são mantidas no Pool de Strings. Portanto, não podemos comparar o conteúdo delas através do operador “==”. Nesse caso, devemos utilizar o método `equals()`.

```
1 String nome1 = new String("Rafael Cosentino");
2 String nome2 = new String("Rafael Cosentino");
3
4 // imprime false
```

```

5 System.out.println(nome1 == nome2);
6 // imprime true
7 System.out.println(nome1.equals(nome2));

```

*Código Java 14.4: Comparando strings fora do Pool de Strings*



## Imutabilidade

Uma característica fundamental dos objetos da classe `String` é que eles são imutáveis. Em outras palavras, o conteúdo de uma string não altera.

Alguns métodos das strings podem dar a impressão errada de que o conteúdo do objeto será alterado. Por exemplo, o método `toUpperCase()` que é utilizado para obter uma string com letras maiúsculas. Esse método não altera a string original, ele cria uma nova string com o conteúdo diferente.

```

1 String nome = "Rafael Cosentino";
2
3 nome.toUpperCase();
4
5 // imprime Rafael Cosentino
6 System.out.println(nome);

```

*Código Java 14.5: Pegadinha...*

```

1 String nome = "Rafael Cosentino";
2
3 String nomeAlterado = nome.toUpperCase();
4
5 // imprime RAFAEL COSENTINO
6 System.out.println(nomeAlterado);

```

*Código Java 14.6: Guardando o resultado do toUpperCase()*



### Mais Sobre

Podemos alterar o conteúdo de qualquer objeto Java de forma invasiva utilizando reflection. Não seria uma boa prática utilizar esses mecanismos para “forçar” modificações nos objetos da classe `String` pois os efeitos colaterais causados no restante do código podem ser drásticos.



## Métodos principais

Todos os métodos da classe `String` podem ser consultados na url <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>. Discutiremos aqui o funcionamento dos principais métodos dessa classe.

### length()

O método `length()` devolve a quantidade de caracteres de uma string.

```

1 String nome = "K19 Treinamentos";
2
3 // imprime 16
4 System.out.println(nome.length());

```

*Código Java 14.7: length()*

## toUpperCase()

O método `toUpperCase()` é utilizado para obter uma cópia de uma string com letras maiúsculas.

```

1 String nome = "Rafael Cosentino";
2
3 String nomeAlterado = nome.toUpperCase();
4
5 // imprime RAFAEL COSENTINO
6 System.out.println(nomeAlterado);

```

*Código Java 14.8: toUpperCase()*

## toLowerCase()

O método `toLowerCase()` é utilizado para obter uma cópia de uma string com letras minúsculas.

```

1 String nome = "Rafael Cosentino";
2
3 String nomeAlterado = nome.toLowerCase();
4
5 // imprime rafael cosentino
6 System.out.println(nomeAlterado);

```

*Código Java 14.9: toLowerCase()*

## trim()

O método `trim()` é utilizado para obter uma cópia de uma string sem os espaços em branco do início e do final.

```

1 String nome = "      Formação Desenvolvedor Java      ";
2
3 String nomeAlterado = nome.trim();
4
5 // imprime Formação Desenvolvedor Java
6 System.out.println(nomeAlterado);

```

*Código Java 14.10: trim()*

## split()

O método `split()` divide uma string em várias de acordo com um delimitador e devolve um array com as strings obtidas.

```

1 String texto = "K11 ,K12 ,K21 ,K22 ,K23";
2
3 String[] cursos = texto.split(",");
4
5 // imprime K11
6 System.out.println(cursos[0]);
7

```

```
8 // imprime K12
9 System.out.println(cursos[1]);
10
11 // imprime K21
12 System.out.println(cursos[2]);
13
14 // imprime K22
15 System.out.println(cursos[3]);
16
17 // imprime K23
18 System.out.println(cursos[4]);
```

*Código Java 14.11: split()*

## replaceAll()

O método `replaceAll()` cria uma cópia de uma string substituindo “pedaços” internos por outro conteúdo.

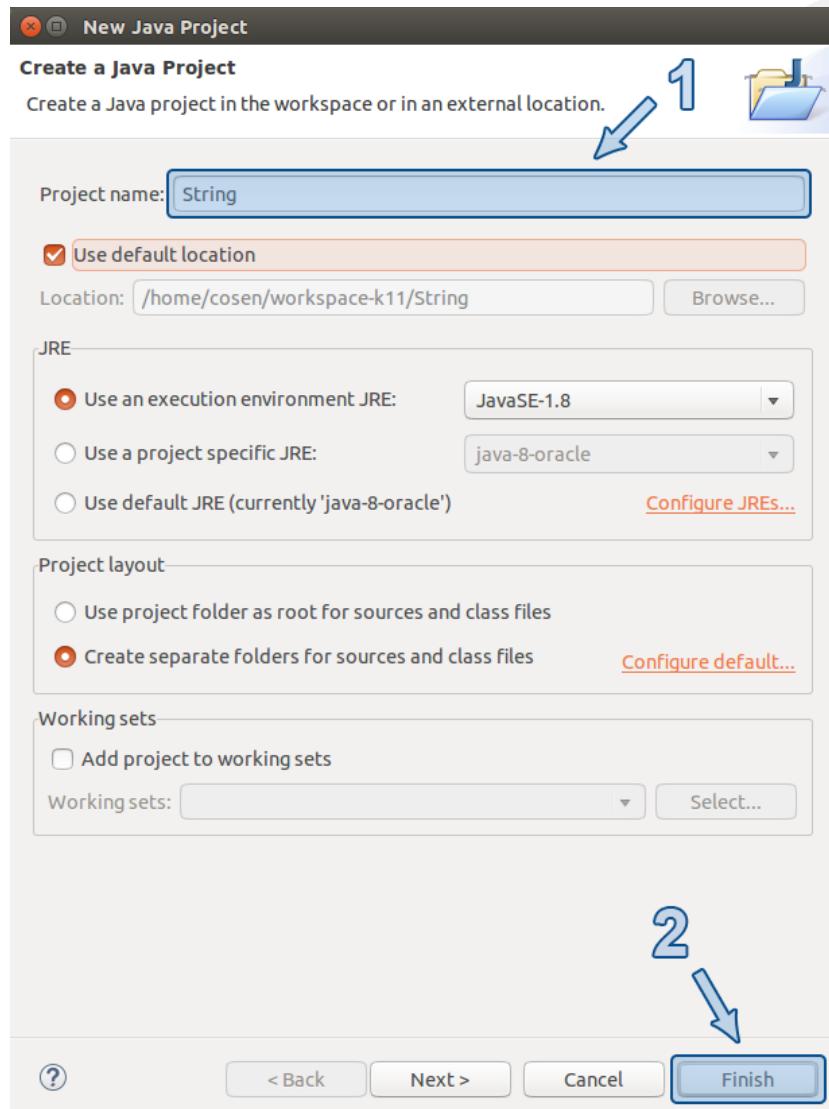
```
1 String texto = "Curso de Java da K19, Curso de JSF da K19";
2
3 String textoAlterado = texto.replaceAll("Curso", "Treinamento");
4
5 // imprime Treinamento de Java da K19, Treinamento de JSF da K19
6 System.out.println(textoAlterado);
```

*Código Java 14.12: replaceAll()*



## Exercícios de Fixação

- Para não confundir, feche o projeto **Exceptions** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie uma classe para testar o Pool de Strings e a comparação com “==” e com equals().

```

1 package br.com.k19.testes;
2
3 public class TestaPoolDeStrings {
4     public static void main(String[] args) {
5         String nome1 = "Rafael Cosentino";
6         String nome2 = "Rafael Cosentino";
7
8         // exibe true
9         System.out.println(nome1 == nome2);
10        // exibe true
11        System.out.println(nome1.equals(nome2));
12
13        String nome3 = new String("Rafael Cosentino");
14        String nome4 = new String("Rafael Cosentino");
15
16        // exibe false
17        System.out.println(nome3 == nome4);
18        // exibe true
19        System.out.println(nome3.equals(nome4));
20    }

```

21 }

*Código Java 14.13: TestaPoolDeStrings.java*

Execute a classe TestaPoolDeStrings.

- 3 Crie uma classe para testar a imutabilidade das strings.

```
1 package br.com.k19.testes;
2
3 public class TestaImutabilidade {
4     public static void main(String[] args) {
5         String nome = "Rafael Cosentino";
6
7         String nomeAlterado = nome.toUpperCase();
8
9         // exibe Rafael Cosentino
10        System.out.println(nome);
11
12        // exibe RAFAEL COSENTINO
13        System.out.println(nomeAlterado);
14    }
15 }
```

*Código Java 14.14: TestaImutabilidade.java*

## ENTRADA E SAÍDA

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, o conteúdo obtido de um arquivo ou os dados recebidos pela rede são exemplos de entrada de dados. A impressão de mensagens no console, a 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.



### Byte a Byte

Em determinadas situações, 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 } while (i != -1);
```

Código Java 15.1: Leitura byte a byte do teclado

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();
```

Código Java 15.2: Escrita byte a byte no console

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 de 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`



## 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. Essa 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 }
```

Código Java 15.3: Leitura utilizando a classe `Scanner`

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);
```

Código Java 15.4: Associando scanners a fontes distintas de dados



## 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. Essa classe possui métodos mais sofisticados para enviar dados para uma saída.

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");
```

*Código Java 15.5: Escrita utilizando PrintStream*

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

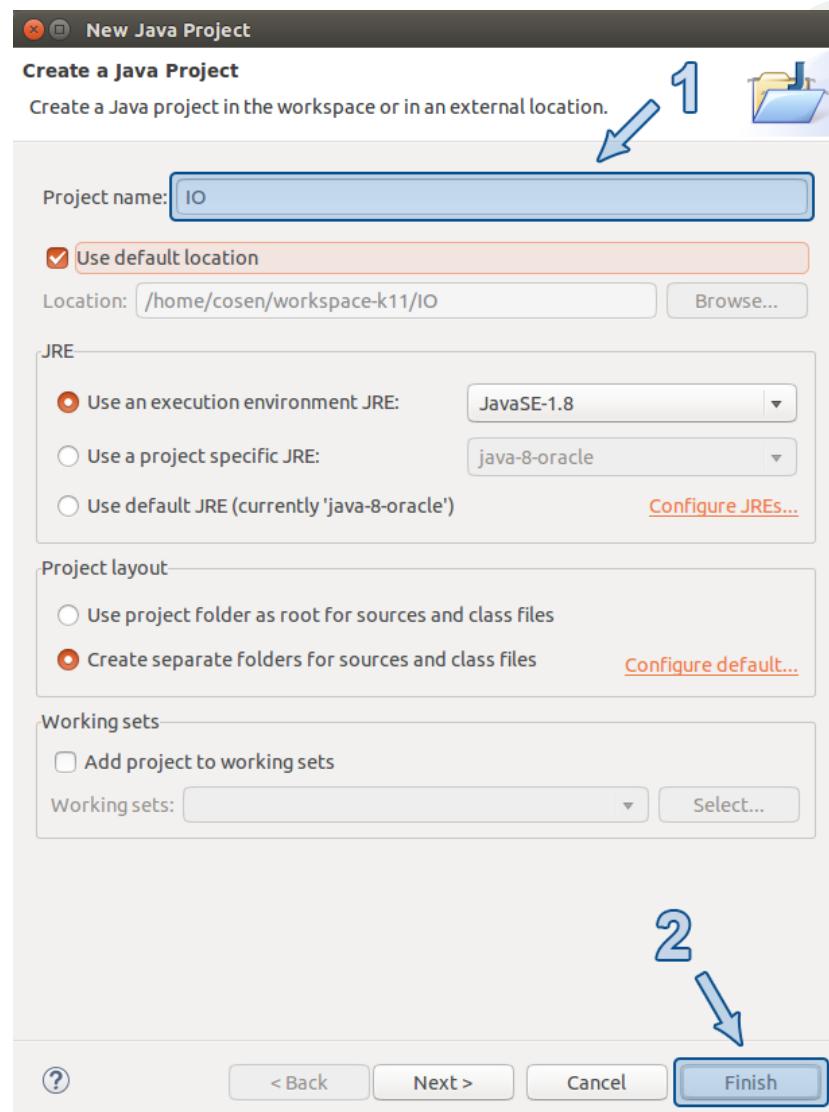
```
1 OutputStream console = System.out;
2 PrintStream printStreamConsole = new PrintStream(console);
3
4 FileOutputStream arquivo = new FileOutputStream("arquivo.txt");
5 PrintStream printStreamArquivo = new PrintStream(arquivo);
```

*Código Java 15.6: Associando printstreams a destinos de dados distintos*



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **String** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “**CTRL + 3**” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie um teste para recuperar e exibir no Console o conteúdo digitado pelo usuário no teclado.

```
1 package br.com.k19.testes;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.util.Scanner;
6
7 public class LeituraDoTeclado {
8     public static void main(String[] args) {
9         InputStream teclado = System.in;
10        Scanner scanner = new Scanner(teclado);
11
12        while (scanner.hasNextLine()) {
13            String linha = scanner.nextLine();
14            System.out.println(linha);
15        }
16
17        scanner.close();
18    }
19 }
```

*Código Java 15.7: LeituraDoTeclado.java*

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 exibir no Console o conteúdo de um arquivo.

```

1 package br.com.k19.testes;
2
3 import java.io.FileInputStream;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.util.Scanner;
7
8 public class LeituraDeArquivo {
9     public static void main(String[] args) throws IOException {
10         InputStream arquivo = new FileInputStream("entrada.txt");
11         Scanner scanner = new Scanner(arquivo);
12
13         while (scanner.hasNextLine()) {
14             String linha = scanner.nextLine();
15             System.out.println(linha);
16         }
17
18         scanner.close();
19     }
20 }
```

*Código Java 15.8: LeituraDeArquivo.java*

OBS: O arquivo “entrada.txt” deve ser criado no diretório **raiz** do projeto **IO**.

- 4 Crie um teste para exibir algumas linhas em um arquivo.

```

1 package br.com.k19.testes;
2
3 import java.io.FileOutputStream;
4 import java.io.IOException;
5 import java.io.PrintStream;
6
7 public class EscritaDeArquivo {
8     public static void main(String[] args) throws IOException {
9         FileOutputStream arquivo = new FileOutputStream("saida.txt");
10        PrintStream printStream = new PrintStream(arquivo);
11
12        printStream.println("Primeira linha!!!!");
13        printStream.println("Segunda linha!!!!");
14        printStream.println("Terceira linha!!!!");
15
16        printStream.close();
17    }
18 }
```

*Código Java 15.9: EscritaDeArquivo.java*

OBS: O projeto **IO** deve ser atualizado para que o arquivo “saida.txt” seja mostrado no eclipse.



## Exercícios Complementares

- 1 Crie um teste que faça a leitura do conteúdo de um arquivo e grave em outro arquivo.
- 2 Crie um teste que faça a leitura do teclado e grave em arquivo.

## 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.



### 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();
```

Código Java 16.1: `ArrayList`

```
1 LinkedList linkedList = new LinkedList();
```

Código Java 16.2: `LinkedList`

```
1 Vector vector = new Vector();
```

Código Java 16.3: `Vector`

Podemos aplicar o polimorfismo e referenciar objetos criados a partir das classes: `ArrayList`, `LinkedList` e `Vector` como `List`.

```
1 List list = new ArrayList();
```

*Código Java 16.4: ArrayList*

```
1 List list = new LinkedList();
```

*Código Java 16.5: LinkedList*

```
1 List list = new Vector();
```

*Código Java 16.6: Vector*

### 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");
```

*Código Java 16.7: Adicionando elementos em uma lista*

### 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");
```

*Código Java 16.8: Adicionando elementos em posições específicas de uma lista*

### Método: size()

O método `size()` informa a quantidade de elementos armazenado na lista.

```
1 List list = ...  
2  
3 list.add("Jonas Hirata");  
4 list.add("Rafael Cosentino");  
5 list.add("Marcelo Martins");  
6 list.add("Thiago Thies");  
7  
8 // quantidade = 4  
9 int quantidade = list.size();
```

*Código Java 16.9: Recuperando a quantidade de elementos de uma lista*

### Método: clear()

O método `clear()` remove todos os elementos da lista.

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4 list.add("Rafael Cosentino");
5 list.add("Marcelo Martins");
6 list.add("Thiago Thies");
7
8 // quantidade = 4
9 int quantidade = list.size();
10
11 list.clear();
12
13 // quantidade = 0
14 quantidade = list.size();

```

*Código Java 16.10: Removendo todos os elementos de uma lista*

### Método: contains(Object)

Para verificar se um elemento está contido em uma lista podemos utilizar o método `contains(Object)`

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4 list.add("Rafael Cosentino");
5
6 // x = true
7 boolean x = list.contains("Jonas Hirata");
8
9 // x = false
10 x = list.contains("Daniel Machado");

```

*Código Java 16.11: Verificando se um elemento está contido em uma lista*

### 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.

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4
5 // x = true
6 boolean x = list.contains("Jonas Hirata");
7
8 list.remove("Jonas Hirata");
9
10 // x = false
11 x = list.contains("Jonas Hirata");

```

*Código Java 16.12: Removendo a primeira ocorrência de um elemento em uma lista*

### Método: remove(int)

Outra forma de retirar elementos de uma lista é através do método `remove(int)`.

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4
5 // x = true

```

```

6 boolean x = list.contains("Jonas Hirata");
7
8 list.remove(0);
9
10 // x = false
11 x = list.contains("Jonas Hirata");

```

*Código Java 16.13: Removendo um elemento pela sua posição em uma lista*

### Método: get(int)

Para recuperar um elemento de uma determinada posição de uma lista podemos utilizar o método `get(int)`.

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4
5 // nome = "Jonas Hirata"
6 String nome = list.get(0);

```

*Código Java 16.14: Recuperando o elemento de uma determinada posição de uma lista*

### Método: indexOf(Object)

Para descobrir o índice da primeira ocorrência de um determinado elemento podemos utilizar o método `indexOf(Object)`.

```

1 List list = ...
2
3 list.add("Jonas Hirata");
4
5 // indice = 0
6 int indice = list.indexOf("Jonas Hirata");

```

*Código Java 16.15: Descobrindo o índice da primeira ocorrência de um elemento em uma lista*

### 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.

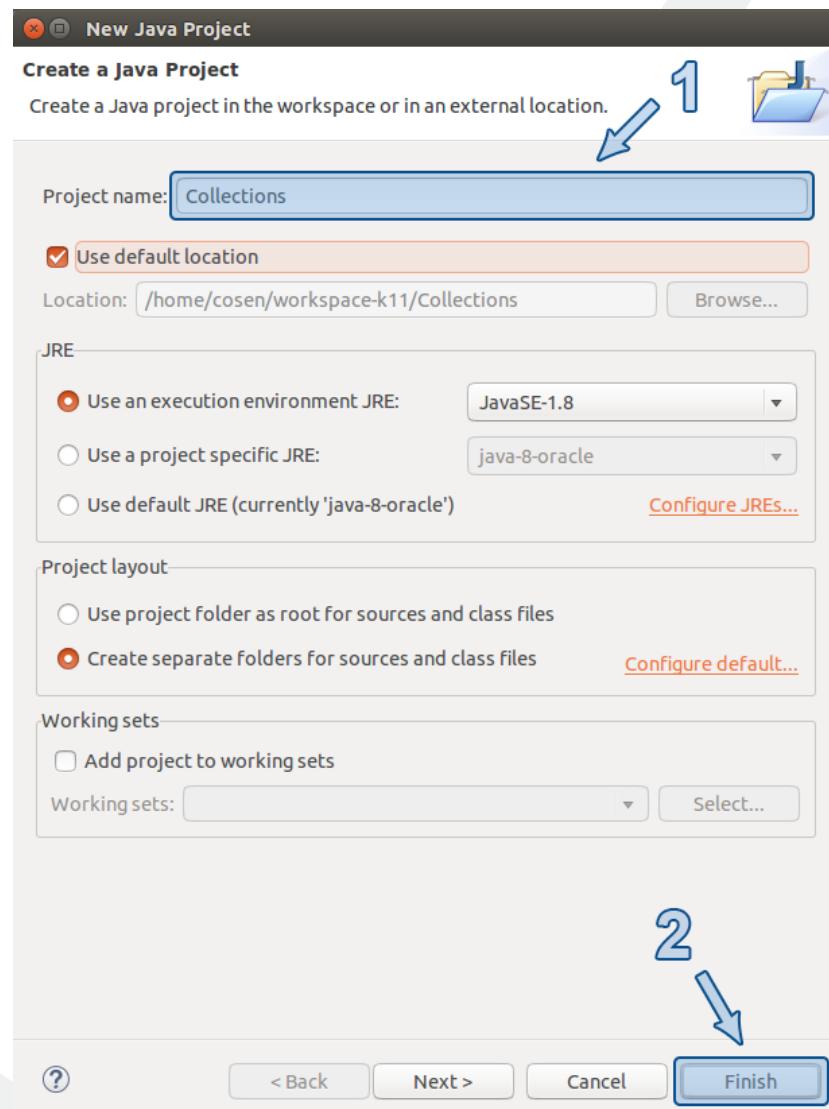
Operação	ArrayList	LinkedList
Adicionar ou Remover do final da lista	☺	☺
Adicionar ou Remover do começo da lista	☺	☺
Acessar elementos pela posição	☺	☺

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.



### Exercícios de Fixação

- 1 Para não confundir, feche o projeto **IO** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Vamos calcular o tempo das operações principais das listas.

```

1 package br.com.k19.testes;
2
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
6
7 public class TestaAdicionaNoFinal {
8     public static void main(String[] args) {
9         ArrayList arrayList = new ArrayList();
10        LinkedList linkedList = new LinkedList();
11
12        long tempo = TestaAdicionaNoFinal.adicionaNoFinal(arrayList);
13        System.out.println("ArrayList: " + tempo + "ms");

```

```

14     tempo = TestaAdicionaNoFinal.adicionaNoFinal(linkedList);
15     System.out.println("LinkedList: " + tempo + "ms");
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
32 }
33 }
```

*Código Java 16.16: TestaAdicionaNoFinal.java*

```

1 package br.com.k19.testes;
2
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
6
7 public class TestaAdicionaNoComeco {
8     public static void main(String[] args) {
9         ArrayList arrayList = new ArrayList();
10        LinkedList linkedList = new LinkedList();
11
12        long tempo = TestaAdicionaNoComeco.adicionaNoComeco(arrayList);
13        System.out.println("ArrayList: " + tempo + "ms");
14
15        tempo = TestaAdicionaNoComeco.adicionaNoComeco(linkedList);
16        System.out.println("LinkedList: " + tempo + "ms");
17
18    }
19
20    public static long adicionaNoComeco(List lista) {
21        long inicio = System.currentTimeMillis();
22
23        int size = 100000;
24
25        for (int i = 0; i < size; i++) {
26            lista.add(0, i);
27        }
28
29        long fim = System.currentTimeMillis();
30
31        return fim - inicio;
32    }
33 }
```

*Código Java 16.17: TestaAdicionaNoComeco.java*

```

1 package br.com.k19.testes;
2
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
6
7 public class TestaGet {
8     public static void main(String[] args) {
9         ArrayList arrayList = new ArrayList();
```

```

10    LinkedList linkedList = new LinkedList();
11
12    long tempo = TestaGet.get(arrayList);
13    System.out.println("ArrayList: " + tempo + "ms");
14
15    tempo = TestaGet.get(linkedList);
16    System.out.println("LinkedList: " + tempo + "ms");
17
18 }
19
20 public static long get(List lista) {
21
22     int size = 100000;
23
24     for (int i = 0; i < size; i++) {
25         lista.add(i);
26     }
27
28     long inicio = System.currentTimeMillis();
29
30     for (int i = 0; i < size; i++) {
31         lista.get(i);
32     }
33
34     long fim = System.currentTimeMillis();
35
36     return fim - inicio;
37 }
38 }
```

*Código Java 16.18: TestaGet.java*

- 3 Teste o desempenho para remover elementos do começo ou do fim das listas.



## 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.



## Coleções

Há semelhanças conceituais entre os conjuntos e as listas por isso existe uma super interface chamada Collection para as interfaces List e Set.

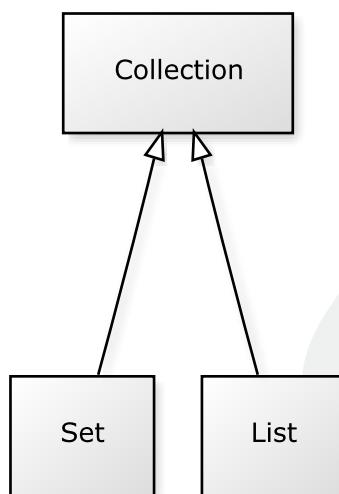


Figura 16.1: Coleções

Dessa forma, podemos referenciar como Collection qualquer lista ou conjunto.



## Exercícios de Fixação

- 4 Vamos comparar o tempo do método contains() das listas e dos conjuntos.

```
1 package br.com.k19.testes;
2
3 import java.util.ArrayList;
4 import java.util.Collection;
5 import java.util.HashSet;
6
7 public class TestaContains {
8     public static void main(String[] args) {
9         ArrayList arrayList = new ArrayList();
10        HashSet hashSet = new HashSet();
11
12        long tempo = TestaContains.contains(arrayList);
13        System.out.println("ArrayList: " + tempo + "ms");
14
15        tempo = TestaContains.contains(hashSet);
16        System.out.println("HashSet: " + tempo + "ms");
17    }
18
19
20    public static long contains(Collection colecao) {
21
22        int size = 100000;
23
24        for (int i = 0; i < size; i++) {
25            colecao.add(i);
26        }
27
28        long inicio = System.currentTimeMillis();
29
30        for (int i = 0; i < size; i++) {
31            colecao.contains(i);
32        }
33
34        long fim = System.currentTimeMillis();
```

```

35     return fim - inicio;
36 }
37 }
38 }
```

Código Java 16.19: TestaContains.java



## Laço foreach

As listas podem ser iteradas com um laço for tradicional.

```

1 List lista = ...
2
3 for(int i = 0; i < lista.size(); i++) {
4     Object x = lista.get(i);
5 }
```

Código Java 16.20: for tradicional

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**.

```

1 Collection colecao = ...
2
3 for(Object x : colecao) {
4 }
```

Código Java 16.21: foreach



## 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.

```

1 Collection colecao = ...
2
3 colecao.add("Rafael Cosentino");
4
5 for(Object x : colecao) {
6     System.out.println(x);
7 }
```

Código Java 16.22: Coleção de elementos genéricos

Porém, normalmente, precisamos tratar os objetos de forma específica pois queremos ter acesso aos métodos específicos desses objetos. Nesses casos, devemos fazer casting de referências.

```

1 Collection colecao = ...
2
3 colecao.add("Rafael Cosentino");
4
```

```

5 for(Object x : colecao) {
6     String s = (String)x;
7     System.out.println(s.toUpperCase());
8 }
```

*Código Java 16.23: Aplicando casting de referências*

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 }
```

*Código Java 16.24: Coleção com Generics*

## Exercícios de Fixação

- 5 Vamos testar o desempenho do for tradicional e do foreach.

```

1 package br.com.k19.testes;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class TestaForeach {
7     public static void main(String[] args) {
8         LinkedList<Integer> linkedList = new LinkedList<Integer>();
9         int size = 100000;
10
11         for (int i = 0; i < size; i++) {
12             linkedList.add(i);
13         }
14
15         long tempo = TestaForeach.forTradisional(linkedList);
16         System.out.println("For Tradisional: " + tempo + "ms");
17
18         tempo = TestaForeach.foreach(linkedList);
19         System.out.println("Foreach: " + tempo + "ms");
20
21     }
22
23     public static long forTradisional(List<Integer> lista) {
24         long inicio = System.currentTimeMillis();
25
26         for (int i = 0; i < lista.size(); i++) {
27             int x = lista.get(i);
28         }
29     }
}
```

```
30     long fim = System.currentTimeMillis();
31
32     return fim - inicio;
33 }
34
35 public static long foreach(List<Integer> lista) {
36     long inicio = System.currentTimeMillis();
37
38     for (int x : lista) {
39
40     }
41
42     long fim = System.currentTimeMillis();
43
44     return fim - inicio;
45 }
46 }
```

Código Java 16.25: TestaForeach.java





# DOCUMENTAÇÃO

Na plataforma Java SE 8, há cerca de 4000 classes e interfaces disponíveis para utilizarmos em nossas aplicações. Podemos visualizar a documentação dessas classes e interfaces na seguinte url <http://docs.oracle.com/javase/7/docs/api/>.

Package	Description
<a href="#">java.applet</a>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<a href="#">java.awt</a>	Contains all of the classes for creating user interfaces and for painting graphics and images.
<a href="#">java.awt.color</a>	Provides classes for color spaces.
<a href="#">java.awt.datatransfer</a>	Provides interfaces and classes for transferring data between and within applications.
<a href="#">java.awt.dnd</a>	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
<a href="#">java.awt.event</a>	Provides interfaces and classes for dealing with different types of events fired by AWT components.
<a href="#">java.awt.font</a>	Provides classes and interface relating to fonts.
<a href="#">java.awt.geom</a>	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
<a href="#">java.awt.im</a>	Provides classes and interfaces for the input method framework.
<a href="#">java.awt.im.spi</a>	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
<a href="#">java.awt.image</a>	Provides classes for creating and modifying images.
<a href="#">java.awt.image.renderable</a>	Provides classes and interfaces for producing rendering-independent images.
<a href="#">java.awt.print</a>	Provides classes and interfaces for a general printing API.
<a href="#">java.beans</a>	Contains classes related to developing beans – components based on the JavaBeans™ architecture.
<a href="#">java.beans.beancontext</a>	Provides classes and interfaces relating to bean context.
<a href="#">java.io</a>	Provides system input and output through data streams, serialization and the file system.
<a href="#">java.lang</a>	Provides classes that are fundamental to the design of the Java programming language.
<a href="#">java.lang.annotation</a>	Provides library support for the Java programming language annotation facility.
<a href="#">java.lang.instrument</a>	Provides services that allow Java programming language agents to instrument programs running on the JVM.
<a href="#">java.lang.invoke</a>	The <code>java.lang.invoke</code> package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
<a href="#">java.lang.management</a>	Provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime.
<a href="#">java.lang.ref</a>	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
<a href="#">java.lang.reflect</a>	Provides classes and interfaces for obtaining reflective information about classes and objects.
<a href="#">java.math</a>	Provides classes for performing arbitrary-precision integer arithmetic ( <code>BigInteger</code> ) and arbitrary-precision decimal arithmetic ( <code>BigDecimal</code> ).
<a href="#">java.net</a>	Provides the classes for implementing networking applications.
<a href="#">java.nio</a>	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
<a href="#">java.nio.channels</a>	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
<a href="#">java.nio.channels.spi</a>	Service-provider classes for the <code>java.nio.channels</code> package.
<a href="#">java.nio.charset</a>	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.

Figura A.1: Documentação Java SE 7

Não é necessário ficar assustado com a quantidade de classes e interfaces contidas na plataforma Java SE 7. Na prática, utilizamos diretamente apenas uma pequena parte dessa gigantesca biblioteca no desenvolvimento de uma aplicação.

Além das bibliotecas da plataforma Java SE 7, há várias outras bibliotecas que podemos utilizar em nossas aplicações. Por exemplo, algumas aplicações utilizam a biblioteca **JFreeChart** para gerar gráficos profissionais. Essa biblioteca também possui uma documentação com a mesma estrutura da documentação da plataforma Java SE 7. Consulte a url <http://www.jfree.org/jfreechart/api/javadoc/index.html>.

The screenshot shows the JavaDoc interface for the JFreeChart library. The left sidebar lists packages and classes. The main area displays the package structure for org.jfree.chart, including annotations, axis, chart, editor, labels, plot, and demo. Each package has a brief description and a list of its components.

Packages	Description
org.jfree.chart.annotations	Core classes, including JFreeChart and ChartPanel.
org.jfree.chart.axis	A framework for adding annotations to charts.
org.jfree.chart.block	Axis classes and interfaces.
org.jfree.chart.demo	Blocks and layout classes used extensively by the LegendTitle class.
org.jfree.chart.editor	Some basic demos to get you started.
org.jfree.chart.labels	Provides a simple (but so far incomplete) framework for editing chart properties.
org.jfree.chart.plot	Classes related to the encoding of charts to different image formats.
org.jfree.chart.plot3d	Classes representing components (or entities) in a chart.
org.jfree.chart.event	Event classes and listener interfaces, used to provide a change notification mechanism so that charts are automatically redrawn whenever changes are made to any chart component.
org.jfree.chart.image	Classes, including ImageUtilities, for creating HTML, image maps.
org.jfree.chart.labels	Generators and other classes used for the display of item labels and tooltips.
org.jfree.chart.plot	Generators and other classes used for the display of item labels and tooltips.
org.jfree.chart.plot3d	A range of objects that can be used to represent the needle on a CompassPlot.
org.jfree.chart.renderer	Classes related to the ChartPanel class.
org.jfree.chart.renderer3d	Plot classes and related interfaces.
org.jfree.chart.plot	Classes for creating dial plots.
org.jfree.chart.plot3d	Core support for the plug-in renderers used by the CategoryPlot and XYPlot classes.
org.jfree.chart.renderer.category	Plug-in renderers for the CategoryPlot class.
org.jfree.chart.renderer.xy	Plug-in renderers for the XYPlot class.
org.jfree.chart.resources	Localised resources for the JFreeChart class library.
org.jfree.chart.servlet	Classes for providing useful servlet and JSP functionality.
org.jfree.chart.title	Classes used to display chart titles and subtitles.
org.jfree.chart.urls	Classes for adding URLs to charts for HTML, image map generation.
org.jfree.chart.util	Utility classes used by JFreeChart.
org.jfree.data	The base package for classes that represent various types of data.
org.jfree.data.category	A package containing the CategoryDataset interface and related classes.
org.jfree.data.contour	Datasets used by the ContourPlot class.
org.jfree.data.function	Representation for simple functions.
org.jfree.data.gantt	Data interfaces and classes for Gantt charts.
org.jfree.data.general	Data interfaces and classes.
org.jfree.data.io	Miscellaneous support for input/output of data.
org.jfree.data.jdbc	Datasets that fetch data from a database via JDBC.
org.jfree.data.resources	Resource bundles for items that require localisation.
org.jfree.data.statistics	Classes for representing statistical data.

Figura A.2: Documentação do JFreeChart

Podemos gerar uma documentação que segue essa estrutura padrão do Java para as nossas classes e interfaces.



## A ferramenta javadoc

Com o intuito de padronizar a estrutura e o visual das documentações das bibliotecas Java, no JDK (Java Development Kit), há uma ferramenta chamada **javadoc** que extrai comentários “especiais” contidos no código fonte Java e gera a documentação das classes e interfaces.

### Comentários javadoc

Os comentários extraídos do código fonte pela ferramenta javadoc devem iniciar com “`/**`”.

```

1 /**
2 *
3 * Comentário javadoc
4 *
5 */

```

Código Java A.1: Comentário javadoc

### Documentando uma pacote

Para documentar um pacote, é necessário criar um arquivo chamado **package-info.java** dentro do pacote que desejamos documentar. Nesse arquivo, devemos adicionar apenas o comando `package` e um comentário javadoc.

```

1 /**
2 *
3 * Documentação do pacote br.com.k19.contas
4 */

```

```

5  */
6 package br.com.k19.contas;

```

*Código Java A.2: package-info.java*

## Documentando uma classe ou interface

Para documentar uma classe ou interface, basta adicionar um comentário javadoc imediatamente acima da declaração da classe ou interface que desejamos documentar.

```

1 package br.com.k19.contas;
2
3 /**
4 *
5 * Documentação da classe br.com.k19.contas.Conta
6 *
7 */
8 public class Conta {
9     // corpo da classe
10 }

```

*Código Java A.3: Conta.java*

Podemos definir os autores de uma classe ou interface, através da tag **@author**. A versão pode ser declarada com a tag **@version**.

```

1 package br.com.k19.contas;
2
3 /**
4 *
5 * Documentacao da classe br.com.k19.contas.Conta
6 *
7 * @author Rafael Cosentino
8 *
9 * @author Jonas Hirata
10 *
11 * @author Marcelo Martins
12 *
13 * @version 1.0
14 *
15 */
16 public class Conta {
17     // corpo da classe
18 }

```

*Código Java A.4: Documentacao.java*

## Documentando um atributo

Para documentar um atributo, basta adicionar um comentário javadoc imediatamente acima da declaração do atributo que desejamos documentar.

```

1 /**
2 *
3 * Documentacao do atributo numero
4 *
5 */
6 private int numero;

```

*Código Java A.5: Documentando um atributo*

## Documentando um construtor

Para documentar um construtor, basta adicionar um comentário javadoc imediatamente acima da declaração do construtor que desejamos documentar.

Os parâmetros de um construtor podem ser documentados através da tag **@param**. As exceptions que podem ser lançadas por um construtor podem ser documentadas através da tag **@throws**.

Essas duas tags podem se repetir no comentário javadoc caso o construtor tenha mais do que um parâmetro ou lance mais do que uma exception.

```
1 /**
2 *
3 * Documentacao do construtor
4 *
5 * @param numero
6 *     documentação do parâmetro numero
7 *
8 * @throws IllegalArgumentException
9 *     documentação da situação que gera a exception IllegalArgumentException
10 */
11 public Conta(int numero) {
12     if(numero < 0) {
13         throw new IllegalArgumentException("número negativo");
14     }
15     this.numero = numero
16 }
```

Código Java A.6: Documentando um construtor

## Documentando um método

Para documentar um método, basta adicionar um comentário javadoc imediatamente acima da declaração do método que desejamos documentar.

Os parâmetros de um método podem ser documentados através da tag **@param**. As exceptions que podem ser lançadas por um método podem ser documentadas através da tag **@throws**. O valor de retorno de um método pode ser documentado através da tag **@return**.

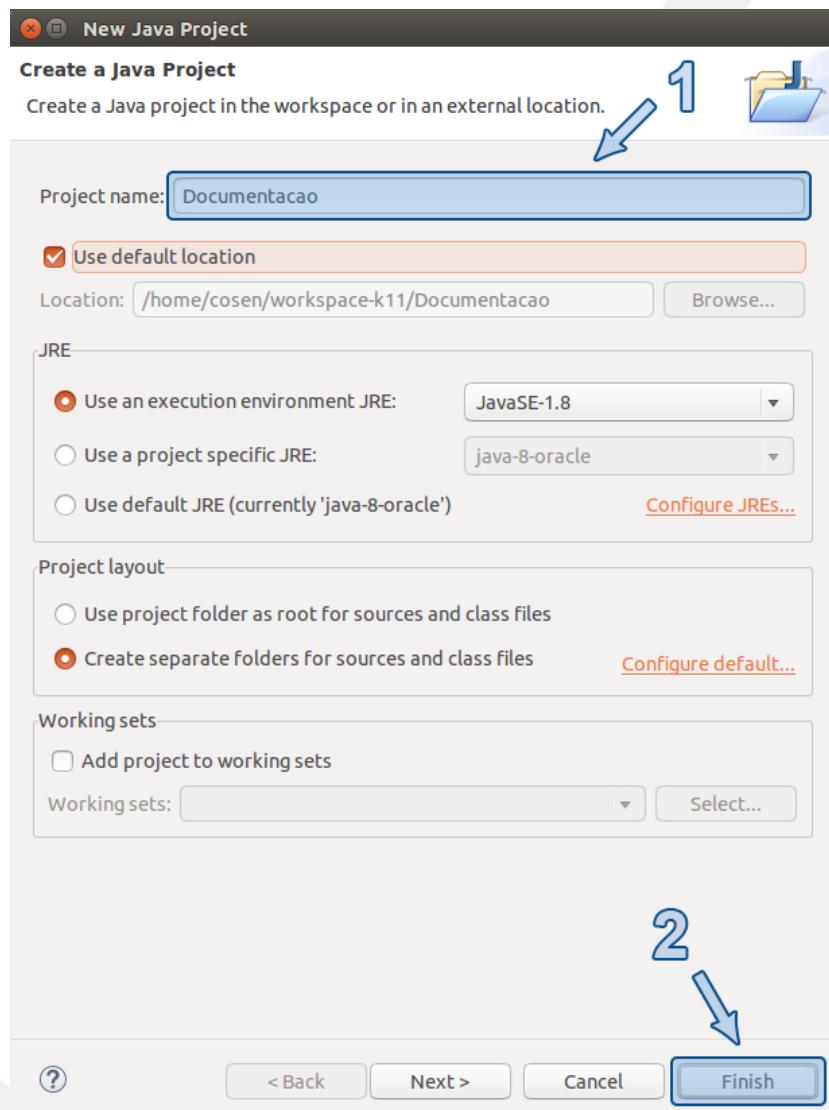
```
1 /**
2 *
3 * Documentacao do método calculaTaxaDeEmprestimo
4 *
5 * @param valor
6 *     documentação do parâmetro valor
7 *
8 * @throws IllegalArgumentException
9 *     documentação da situação que gera a exception IllegalArgumentException
10 *
11 * @return documentação do valor de retorno no método
12 */
13 public double calculaTaxaDeEmprestimo(double valor) {
14     // corpo do método
15 }
```

Código Java A.7: Documentando um método



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Collections** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Adicione um pacote chamado **br.com.k19.banco.contas** e outro chamado **br.com.k19.banco.funcionarios** no projeto **Documentacao**.
- 3 Crie um arquivo no pacote **br.com.k19.banco.contas** chamado **package-info.java** com o seguinte conteúdo.

```
1 /**
2  * Documentação do pacote br.com.k19.banco.contas
3 */
4 package br.com.k19.banco.contas;
```

*Código Java A.8: package-info.java*

- 4 Crie um arquivo no pacote **br.com.k19.banco.funcionarios** chamado **package-info.java** com o seguinte conteúdo.

```
1 /**
2  * Documentação do pacote br.com.k19.banco.funcionarios
3  */
4 package br.com.k19.banco.funcionarios;
```

*Código Java A.9: package-info.java*

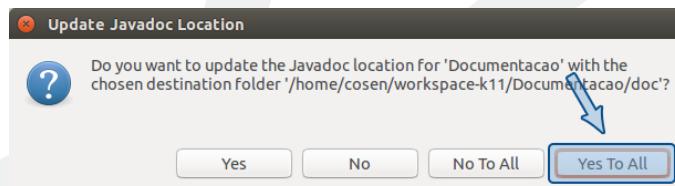
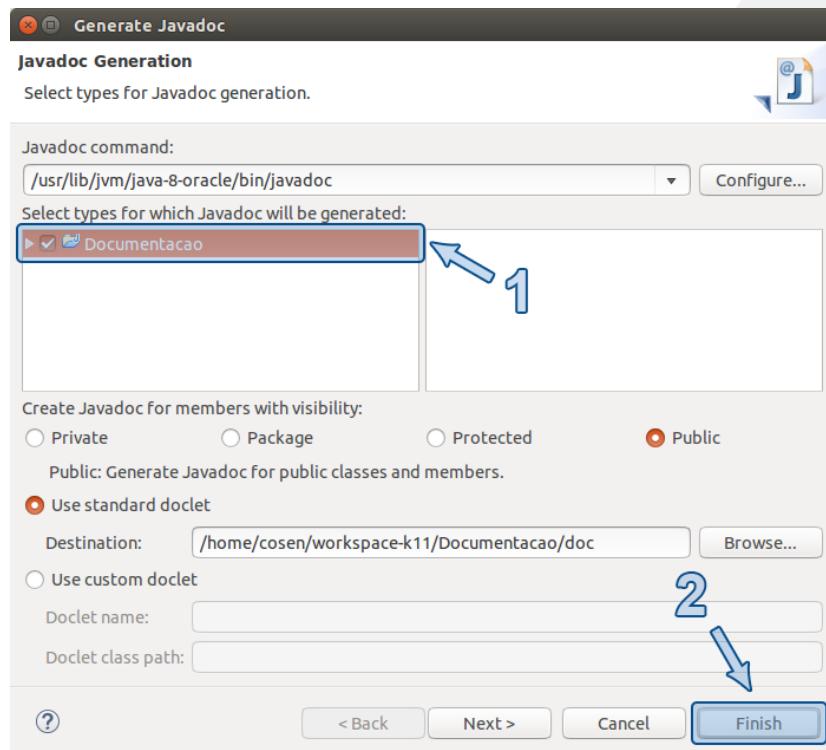
- 5 Adicione uma classe no pacote **br.com.k19.banco.contas** com o seguinte conteúdo:

```
1 package br.com.k19.banco.contas;
2
3 /**
4  *
5  * Documentação da classe Conta
6  *
7  * @author Rafael Cosentino
8  *
9  * @author Jonas Hirata
10 *
11 * @author Marcelo Martins
12 *
13 * @version 1.0
14 */
15 public class Conta {
16
17 /**
18  * Documentação do atributo numero
19  */
20 private int numero;
21
22 /**
23  * Documentação do construtor
24  *
25  * @param numero
26  *          documentação do atributo numero
27  *
28  * @throws IllegalArgumentException
29  *          documentação da situação que gera a exception
30  *          IllegalArgumentException
31 */
32 public Conta(int numero) {
33     if (numero < 0) {
34         throw new IllegalArgumentException("número negativo");
35     }
36     this.numero = numero;
37 }
38
39 /**
40  * Documentação do método getNumero
41  *
42  * @return documentação do valor de retorno
43  */
44 public int getNumero() {
45     return numero;
46 }
47 }
```

*Código Java A.10: Conta.java*

- 6 Gere a documentação do projeto. Digite “CTRL + 3” e pesquise por “generate javadoc”. Selecione

a opção correspondente e siga as imagens abaixo.



- 7 Abra em um navegador o arquivo **Documentacao/doc/index.html**





# 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.



## 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.

### 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 00");  
2 frame.setSize(300, 200);  
3 frame.setVisible(true);
```

*Código Java B.1: Criando uma Janela*



Figura B.1: Janela

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);
```

Código Java B.2: Associando uma ação ao botão de fechar janela

Por padrão, o visual das janelas utiliza o estilo definido pelo sistema operacional. Mas, podemos alterar esse comportamento padrão.

## 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 JPanel panel = new JPanel();
3
4 frame.setContentPane(panel);
```

Código Java B.3: JPanel

## 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);

```

*Código Java B.4: JTextField**Figura B.2: Janela*

## JTextArea

Para textos maiores podemos aplicar o componente definido pela classe JTextArea.

```
1 JTextArea textArea = new JTextArea(10, 20);
```

*Código Java B.5: JTextArea**Figura B.3: Janela*

## 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);
```

*Código Java B.6: JPasswordField**Figura B.4: Janela*

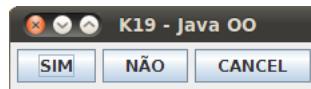
## JButton

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 JButton button2 = new JButton("NÃO");
3 JButton button3 = new JButton("CANCEL");

```

*Código Java B.7: JButton**Figura B.5: Janela*

## JCheckBox

Podemos criar formulários com checkbox's utilizando o componente da classe JCheckBox.

```

1 JCheckBox checkBox1 = new JCheckBox("Rafael Cosentino");
2 JCheckBox checkBox2 = new JCheckBox("Jonas Hirata");
3 JCheckBox checkBox3 = new JCheckBox("Marcelo Martins");

```

*Código Java B.8: JCheckBox**Figura B.6: Janela*

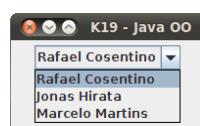
## JComboBox

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);

```

*Código Java B.9: JComboBox**Figura B.7: Janela*

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.



Figura B.8: Janela

**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.

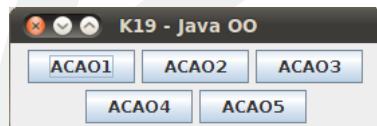


Figura B.9: Janela

**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.

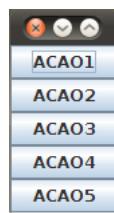


Figura B.10: Janela

**GridLayout:** Esse Layout Manager divide a área de um componente de background em células semelhantemente a uma tabela. As células possuem o mesmo tamanho.



Figura B.11: Janela



## 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.

### 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 }
```

Código Java B.10: MeuListener.java

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 
```

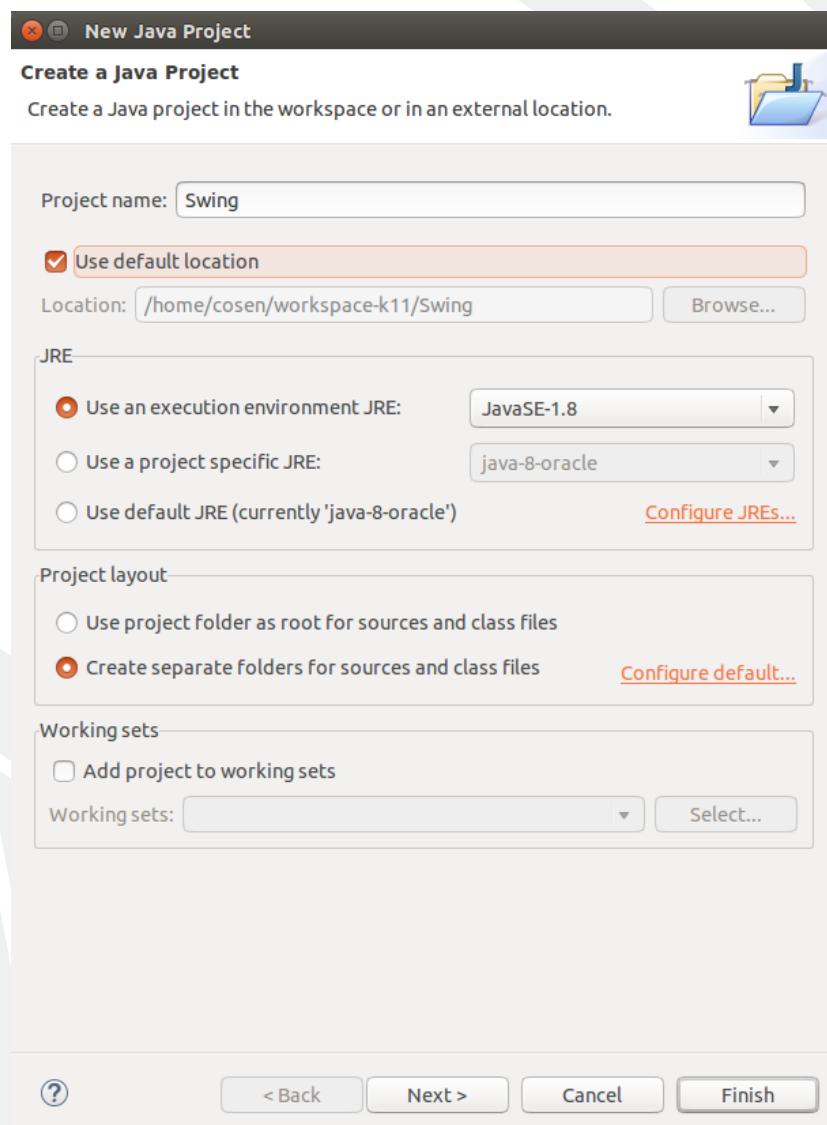
```
4 MeuListener listener = new MeuListener();  
5  
6 button1.addActionListener(listener);  
7 button2.addActionListener(listener);
```

Código Java B.11: Associando listeners e botões



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Documentacao** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 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 package br.com.k19.testes;
2
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7 import javax.swing.JPasswordField;
8 import javax.swing.JTextField;
9
10 public class Teste {
11     public static void main(String[] args) {
12         JFrame frame = new JFrame("K19 - Login");
13         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15         JPanel panel = new JPanel();
16
17         JLabel label1 = new JLabel("Usuário: ");
18
19         JTextField textField = new JTextField(20);
20
21         JLabel label2 = new JLabel("Senha: ");
22
23         JPasswordField passwordField = new JPasswordField(20);
24
25         JButton button = new JButton("Logar");
26
27         panel.add(label1);
28         panel.add(textField);
29         panel.add(label2);
30         panel.add(passwordField);
31         panel.add(button);
32
33         frame.setContentPane(panel);
34
35         frame.pack();
36         frame.setVisible(true);
37     }
38 }
```

Código Java B.12: Teste.java

Execute a classe Teste.

- 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));
```

Código Java B.13: Definindo um Layout Manager

Execute a classe Teste. Depois, tente redimensionar a tela para observar o comportamento.

- 5** 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.



# EMPACOTAMENTO

Para distribuir uma aplicação ou biblioteca Java, devemos utilizar a ferramenta **jar** (Java Archive Tool) para empacotar o código compilado. Essa ferramenta faz parte do JDK (Java Development Kit).



## Empacotando uma biblioteca

Na linha de comando, podemos acionar a ferramenta *jar*.

```
K19$ jar cf biblioteca.jar *
```

*Terminal C.1: Criando um biblioteca*

O empacotamento gera um arquivo com a extensão **.jar**. Esse arquivo pode ser adicionado no *classpath* de uma aplicação.



## Empacotando uma aplicação

Para empacotar uma aplicação, é necessário selecionar a classe que possui o método *main* que desejamos executar. Essa classe deve ser definida em um arquivo chamado **MANIFEST.MF** que deve estar em uma pasta **META-INF**.

```
1 Manifest-Version: 1.0
2 Created-By: 1.6.0_26 (Sun Microsystems Inc.)
3 Main-Class: br.com.k19.App
```

*Código Java C.1: MANIFEST.MF*

Através da ferramenta *jar*, podemos determinar o valor da propriedade *Main-Class*. O arquivo **MANIFEST.MF** é gerado automaticamente.

```
K19$ jar cfe app.jar br.com.k19.App *
```

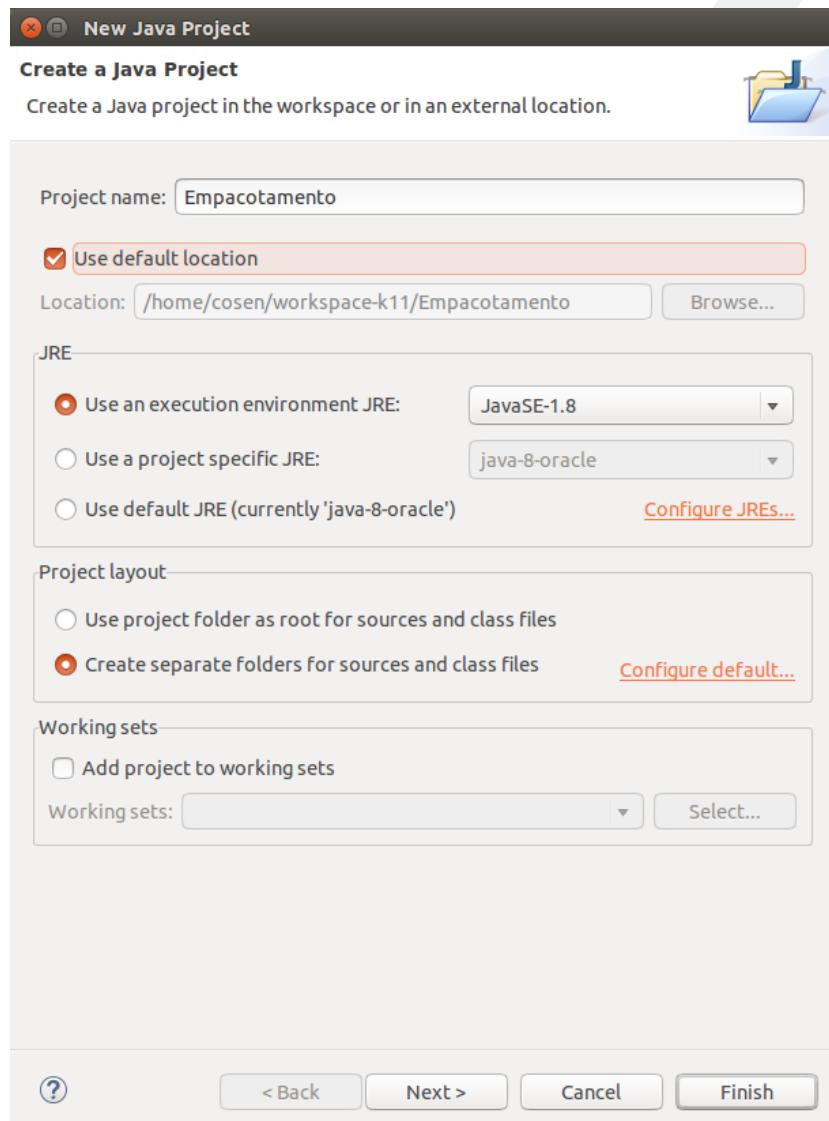
*Terminal C.2: Criando um Aplicação*



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Swing** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios

desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Adicione no projeto **Empacotamento** uma pacote chamado **br.com.k19.testes** com a seguinte classe.

```
1 package br.com.k19.testes;
2
3 import javax.swing.JFrame;
4
5 public class App {
6     public static void main(String[] args) {
7         JFrame janela = new JFrame("K19 - Empacotamento");
8         janela.setSize(300, 300);
9         janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10        janela.setVisible(true);
11    }
12 }
```

Código Java C.2: App.java

- 3 Abra um terminal, entre na pasta **workspace/Empacotamento/bin** e execute o seguinte comando:

```
K19$ jar cfe app.jar br.com.k19.App *
```

*Terminal C.3: Criando um Aplicação*

- 4 Execute a aplicação através do seguinte comando:

```
K19$ java -jar app.jar
```

*Terminal C.4: Executando a aplicação*





## 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 hyperthreading 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.



### 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 }
```

Código Java D.1: TarefaImprimeOi.java

```

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 }
```

Código Java D.2: TarefaSomaAte100.java



## Executando Tarefas

As tarefas são executadas “dentro” de objetos da classe Thread. Para cada tarefa que desejamos executar, devemos criar um objeto da classe Thread e associá-lo ao objeto que define 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);
```

Código Java D.3: Associando tarefas e threads

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();
```

Código Java D.4: Executando uma thread

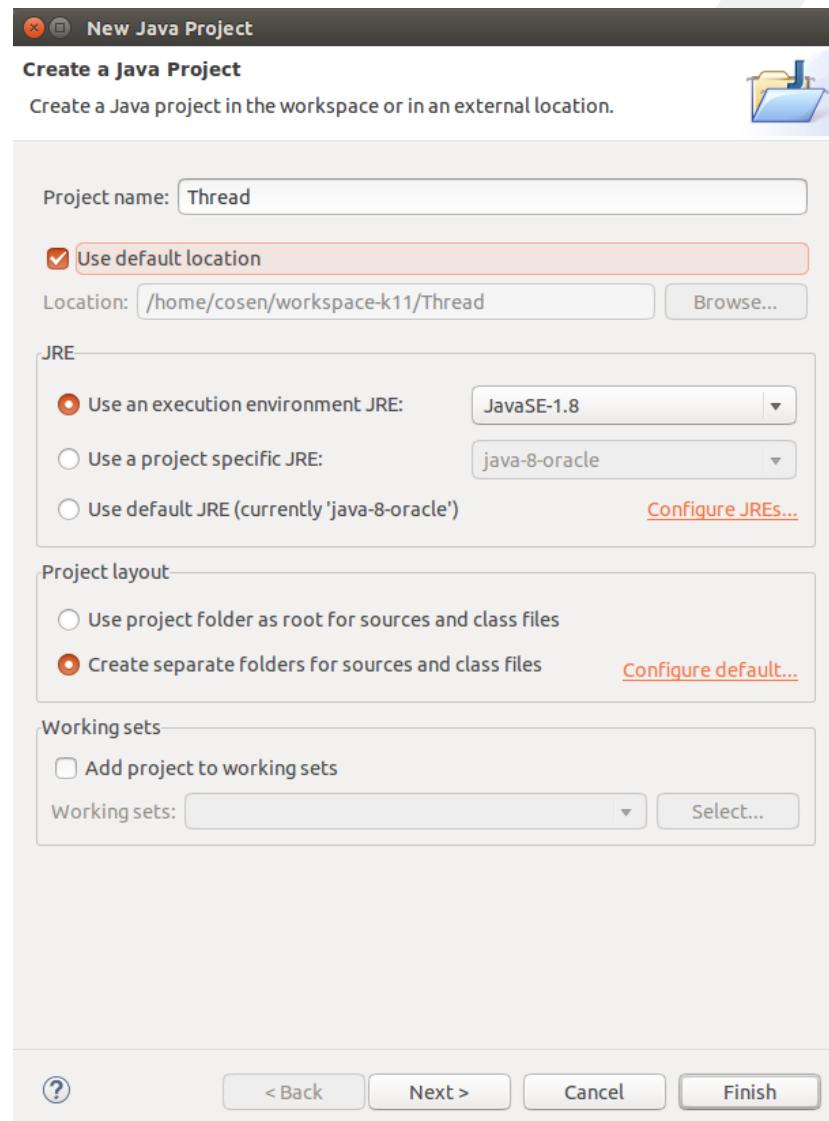
Podemos “disparar” diversas threads e elas poderão ser executadas simultaneamente de acordo com o revezamento que a máquina virtual e o sistema operacional aplicarem.



## Exercícios de Fixação

- Para não confundir, feche o projeto **Empacotamento** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os

exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



**2** Defina uma tarefa para exibir mensagens na Console.

```
1 package br.com.k19.testes;
2
3 public class TarefaExibeMensagens implements Runnable {
4
5     private String msg;
6
7     public TarefaExibeMensagens(String msg) {
8         this.msg = msg;
9     }
10
11    public void run() {
12        for (int i = 0; i < 100000; i++) {
13            System.out.println(i + " : " + this.msg);
14        }
15    }
16 }
```

*Código Java D.5: TarefaExibeMensagens.java*

- 3 Crie tarefas e associe-as com threads para executá-las.

```

1 package br.com.k19.testes;
2
3 public class Teste {
4     public static void main(String[] args) {
5         TarefaExibeMensagens tarefa1 = new TarefaExibeMensagens("K19");
6         TarefaExibeMensagens tarefa2 = new TarefaExibeMensagens("Java");
7         TarefaExibeMensagens tarefa3 = new TarefaExibeMensagens("Web");
8
9         Thread thread1 = new Thread(tarefa1);
10        Thread thread2 = new Thread(tarefa2);
11        Thread thread3 = new Thread(tarefa3);
12
13        thread1.start();
14        thread2.start();
15        thread3.start();
16    }
17 }
```

*Código Java D.6: Teste.java*

Execute a classe Teste.



## 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.

### 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);
```

*Código Java D.7: Sleep*

### InterruptedException

Uma thread que está “dormindo” pode ser interrompida por outra thread. Quando isso ocorrer, a thread que está “dormindo” recebe uma InterruptedException.

```

1 try {
2     Thread.sleep(3000);
3 } catch (InterruptedException e) {
4 }
```

5 }

Código Java D.8: Sleep

**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`.

```

1 TarefaImprimeMensagens tarefa = new TarefaImprimeMensagens("K19");
2 Thread thread = new Thread(tarefa);
3 thread.start();
4
5 try {
6     thread.join();
7 } catch (InterruptedException e) {
8 }

```

Código Java D.9: Join

**Exercícios de Fixação**

- 4 Altere a classe `TarefaExibeMensagens` do projeto `Thread`, adicionando uma chamada ao método `sleep()`.

```

1 package br.com.k19.testes;
2
3 public class TarefaExibeMensagens implements Runnable {
4
5     private String msg;
6
7     public TarefaExibeMensagens(String msg) {
8         this.msg = msg;
9     }
10
11    public void run() {
12        for (int i = 0; i < 100000; i++) {
13            System.out.println(i + " : " + this.msg);
14
15            if(i % 1000 == 0) {
16                try {
17                    Thread.sleep(100);
18                } catch (InterruptedException e) {
19                }
20            }
21        }
22    }
23 }

```

Código Java D.10: `TarefaImprimeMensagens.java`

Execute a classe `Teste`.





# 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.

## 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.

## 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.

## 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.

## 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.





# 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 envie dados para outros computadores da mesma rede.



## Socket

A classe Socket define o funcionamento dos sockets em Java.

```
1 Socket socket = new Socket("184.72.247.119", 1000);
```

*Código Java F1: Abrindo um socket*

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());
```

*Código Java F2: Associando scanners e printstreams a sockets*

O funcionamento da classe PrintStream e Scanner foi visto no capítulo [15](#).



## 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.

```
1 ServerSocket severSocket = new ServerSocket(1000);
2
3 Socket socket = severSocket.accept();
```

*Código Java F3: Aguardando uma conexão*

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 = severSocket.accept();
5 }
```

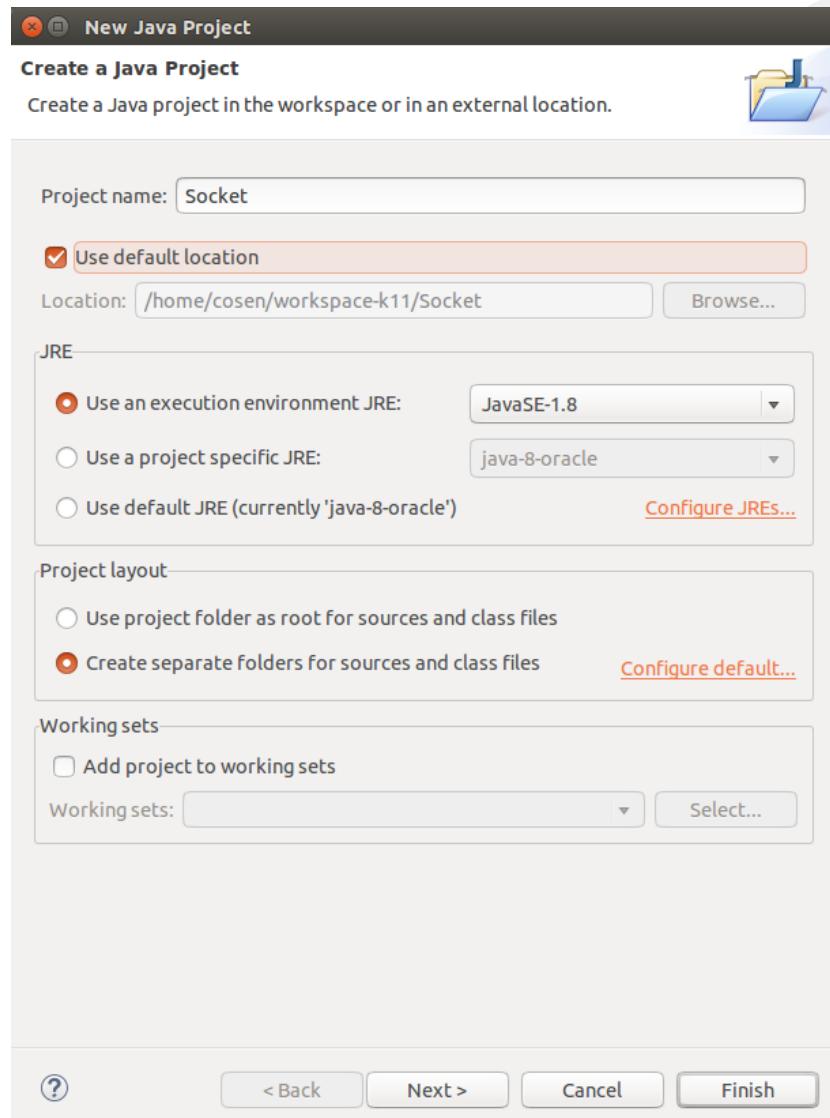
Código Java F4: Aguardando conexões

Cada iteração do laço acima estabelece uma conexão nova com uma aplicação cliente.



## Exercícios de Fixação

- Para não confundir, feche o projeto **Thread** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



2 Crie o código de uma aplicação servidora.

```
1 package br.com.k19.testes;
2
3 import java.io.IOException;
4 import java.io.PrintStream;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7
8 public class Servidor {
9     public static void main(String[] args) throws IOException {
10         ServerSocket serverSocket = new ServerSocket(10000);
11
12         Socket socket = serverSocket.accept();
13
14         System.out.println("Um cliente chegou!");
15
16         PrintStream saida = new PrintStream(socket.getOutputStream());
17
18         saida.println("Você se conectou ao servidor da K19!");
19
20         serverSocket.close();
```

```
21 }  
22 }
```

Código Java E5: Servidor.java

- 3 Crie o código de uma aplicação cliente.

```
1 package br.com.k19.testes;  
2  
3 import java.net.Socket;  
4 import java.util.Scanner;  
5  
6 public class Cliente {  
7     public static void main(String[] args) throws Exception {  
8         Socket socket = new Socket("127.0.0.1", 10000);  
9  
10        Scanner entrada = new Scanner(socket.getInputStream());  
11  
12        String linha = entrada.nextLine();  
13  
14        System.out.println(linha);  
15  
16        socket.close();  
17  
18        entrada.close();  
19    }  
20}
```

Código Java E6: Cliente.java

- 4 Abra um terminal, entre na pasta bin do projeto Sockets e execute a classe Servidor.

```
K19/Rafael/workspace/Socket/bin$ java br.com.k19.testes.Servidor
```

Terminal E1: executando o servidor

- 5 Abra outro terminal, entre na pasta bin do projeto Sockets e execute a classe Cliente.

```
K19/Rafael/workspace/Socket/bin$ java br.com.k19.testes.Cliente  
Um cliente chegou!  
K19/Rafael/workspace/Socket/bin$
```

Terminal E2: servidor recebendo um cliente

```
K19/Rafael/workspace/Socket/bin$ java br.com.k19.testes.Cliente  
Você se conectou ao servidor da K19!  
K19/Rafael/workspace/Socket/bin$
```

Terminal E3: executando um cliente



# CHAT K19



## 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.



## Aplicação servidora

### 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 }
```

*Código Java G.1: Registrador.java*

### 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 }
```

*Código Java G.2: Receptor*

### 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 }
```

Código Java G.3: *Emissor.java*

## 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 }
```

Código Java G.4: *Distribuidor.java*



## Aplicação cliente

### 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 }
```

Código Java G.5: *EmissorDeMensagem.java*

### ReceptorDeMensagem

Na aplicação cliente, deve existir um objeto da classe ReceptorDeMensagem que aguarda as mensagens enviadas pela aplicação servidora e as apresenta para o usuário.

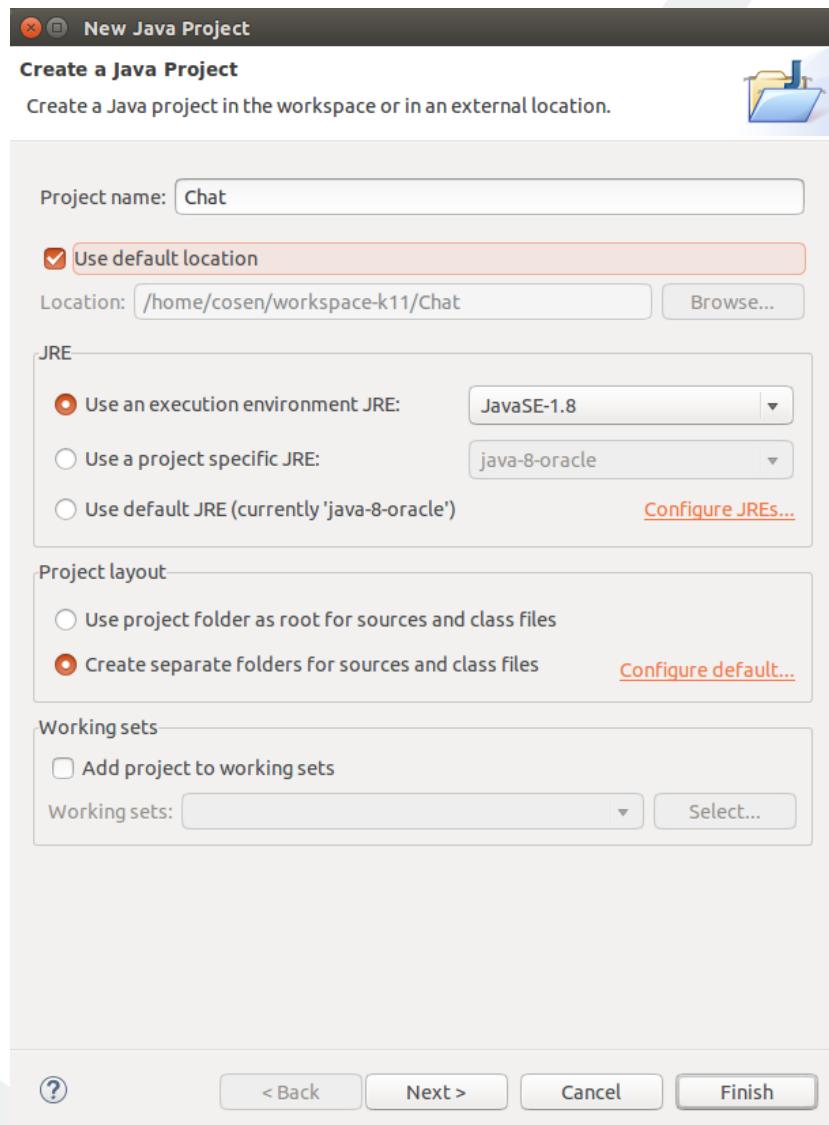
```
1 public class ReceptorDeMensagem {  
2     public void aguardaMensagem() {  
3     }  
4 }  
5 }
```

Código Java G.6: *ReceptorDeMensagem.java*



## Exercícios de Fixação

- 1 Para não confundir, feche o projeto **Socket** clicando com o botão direito do mouse sobre esse projeto e selecionando a opção “Close Project”. Em seguida, crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie uma classe para definir os emissores.

```

1 package br.com.k19.testes;
2
3 import java.io.PrintStream;
4
5 public class Emissor {
6
7     private PrintStream saida;
8
9     public Emissor(PrintStream saida) {
10        this.saida = saida;
11    }
12
13    public void envia(String mensagem) {

```

```

14     this.saida.println(mensagem);
15 }
16 }
```

*Código Java G.7: Emissor.java*

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.

- 3** Crie uma classe para definir o distribuidor de mensagens.

```

1 package br.com.k19.testes;
2
3 import java.util.Collection;
4 import java.util.ArrayList;
5
6 public class Distribuidor {
7     private Collection<Emissor> emissores = new ArrayList<Emissor>();
8
9     public void adicionaEmissor(Emissor emissor) {
10         this.emissores.add(emissor);
11     }
12
13     public void distribuiMensagem(String mensagem) {
14         for (Emissor emissor : this.emissores) {
15             emissor.envia(mensagem);
16         }
17     }
18 }
```

*Código Java G.8: Distribuidor.java*

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.

- 4** Crie uma classe para definir os receptores.

```

1 package br.com.k19.testes;
2
3 import java.util.Scanner;
4
5 public class Receptor implements Runnable {
6     private Scanner entrada;
7     private Distribuidor distribuidor;
8
9     public Receptor(Scanner entrada, Distribuidor distribuidor) {
10         this.entrada = entrada;
11         this.distribuidor = distribuidor;
12     }
13
14     public void run() {
```

```

15     while (this.entrada.hasNextLine()) {
16         String mensagem = this.entrada.nextLine();
17         this.distribuidor.distribuiMensagem(mensagem);
18     }
19 }
20 }
```

Código Java G.9: Receptor.java

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.

### 5 Crie uma classe para definir o registrador.

```

1 package br.com.k19.testes;
2
3 import java.io.IOException;
4 import java.io.PrintStream;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7 import java.util.Scanner;
8
9 public class Registrador implements Runnable {
10
11     private Distribuidor distribuidor;
12     private ServerSocket serverSocket;
13
14     public Registrador(Distribuidor distribuidor, ServerSocket serverSocket) {
15         this.distribuidor = distribuidor;
16         this.serverSocket = serverSocket;
17     }
18
19
20     public void run() {
21         while (true) {
22             try {
23                 Socket socket = this.serverSocket.accept();
24
25                 Scanner entrada = new Scanner(socket.getInputStream());
26                 PrintStream saida = new PrintStream(socket.getOutputStream());
27
28                 Receptor receptor = new Receptor(entrada, this.distribuidor);
29                 Thread pilha = new Thread(receptor);
30                 pilha.start();
31
32                 Emissor emissor = new Emissor(saida);
33
34                 this.distribuidor.adicionaEmissor(emissor);
35
36             } catch (IOException e) {
37                 System.out.println("ERRO");
38             }
39         }
40     }
41 }
```

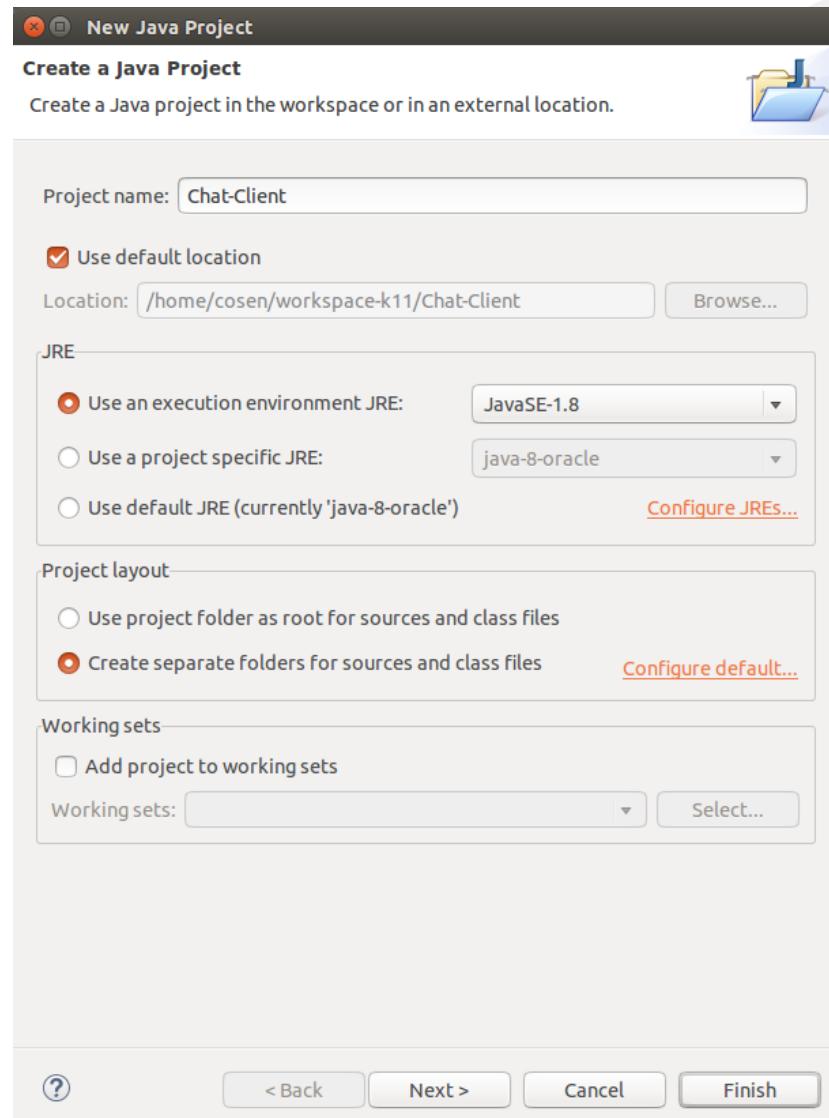
Código Java G.10: Registrador.java

- 6 Crie uma classe para inicializar o servidor.

```
1 package br.com.k19.testes;
2
3 import java.io.IOException;
4 import java.net.ServerSocket;
5
6 public class Server {
7     public static void main(String[] args) throws IOException {
8         Distribuidor distribuidor = new Distribuidor();
9
10        ServerSocket serverSocket = new ServerSocket(10000);
11
12        Registrador registrador = new Registrador(distribuidor, serverSocket);
13        Thread pilha = new Thread(registrador);
14        pilha.start();
15    }
16 }
```

Código Java G.11: Server.java

- 7 Crie mais um novo projeto para os exercícios desse capítulo. Digite “CTRL + 3” e pesquise por “Create a Java project”. Selecione a opção correspondente e siga a imagem abaixo.



- 8 No projeto **Chat-Client**, crie uma classe para definir o emissor de mensagens.

```

1 package br.com.k19.testes;
2
3 import java.io.PrintStream;
4
5 public class EmissorDeMensagem {
6     private PrintStream saida;
7
8     public EmissorDeMensagem(PrintStream saida) {
9         this.saida = saida;
10    }
11
12    public void envia(String mensagem) {
13        this.saida.println(mensagem);
14    }
15 }
```

Código Java G.12: EmissorDeMensagem.java

- 9 No projeto **Chat-Client**, crie uma classe para definir a tela em Java Swing do chat.

```
1 package br.com.k19.testes;
2
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5
6 import javax.swing.JButton;
7 import javax.swing.JFrame;
8 import javax.swing.JLabel;
9 import javax.swing.JPanel;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextArea;
12 import javax.swing.JTextField;
13
14 public class TelaK19Chat {
15
16     private final JFrame frame;
17     private final JPanel panel;
18     private final JScrollPane scrollPane;
19     private final JTextArea textArea1;
20     private final JLabel label1;
21     private final JTextField textField;
22     private final JButton button;
23
24     private final EmissorDeMensagem emissorDeMensagem;
25
26     public TelaK19Chat(EmissorDeMensagem emissor) {
27         this.emissorDeMensagem = emissor;
28
29         this.frame = new JFrame("K19 - Chat");
30         this.panel = new JPanel();
31         this.textArea1 = new JTextArea(10, 60);
32         this.textArea1.setEditable(false);
33         this.scrollPane = new JScrollPane(this.textArea1);
34         this.label1 = new JLabel("Digite uma mensagem...");
35         this.textField = new JTextField(60);
36         this.button = new JButton("Enviar");
37
38         this.frame.setContentPane(this.panel);
39         this.panel.add(this.scrollPane);
40         this.panel.add(this.label1);
41         this.panel.add(this.textField);
42         this.panel.add(button);
43
44         class EnviaMensagemListener implements ActionListener {
45
46             public void actionPerformed(ActionEvent e) {
47                 emissorDeMensagem.envia(textField.getText());
48                 textField.setText("");
49             }
50         }
51
52         this.button.addActionListener(new EnviaMensagemListener());
53
54         this.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
55         this.frame.setSize(700, 300);
56         this.frame.setVisible(true);
57
58     }
59
60     public void adicionaMensagem(String mensagem) {
61         this.textArea1.append(mensagem + "\n");
62     }
63 }
64 }
```

Código Java G.13: TelaK19Chat.java

- 10 No projeto **Chat-Client**, crie uma classe para definir o receptor de mensagens.

```

1 package br.com.k19.testes;
2
3 import java.util.Scanner;
4
5 public class ReceptorDeMensagem implements Runnable {
6     private Scanner entrada;
7
8     private TelaK19Chat telaK19Chat;
9
10    public ReceptorDeMensagem(Scanner entrada, TelaK19Chat telaK19Chat) {
11        this.entrada = entrada;
12        this.telaK19Chat = telaK19Chat;
13    }
14
15    public void run() {
16        while (this.entrada.hasNextLine()) {
17            String mensagem = this.entrada.nextLine();
18            this.telaK19Chat.adicionaMensagem(mensagem);
19        }
20    }
21 }
```

*Código Java G.14: ReceptorDeMensagem.java*

- 11 No projeto **Chat-Client**, crie uma classe para inicializar o cliente.

```

1 package br.com.k19.testes;
2
3 import java.io.PrintStream;
4 import java.net.Socket;
5 import java.util.Scanner;
6
7 public class Client {
8     public static void main(String[] args) throws Exception {
9
10        Socket socket = new Socket("IP DO SERVIDOR", 10000);
11
12        PrintStream saida = new PrintStream(socket.getOutputStream());
13
14        Scanner entrada = new Scanner(socket.getInputStream());
15
16        EmissorDeMensagem emissor = new EmissorDeMensagem(saida);
17
18        TelaK19Chat telaK19Chat = new TelaK19Chat(emissor);
19
20        ReceptorDeMensagem receptor = new ReceptorDeMensagem(entrada,
21                      telaK19Chat);
22        Thread pilha = new Thread(receptor);
23        pilha.start();
24    }
25 }
```

*Código Java G.15: Client.java*





## QUIZZES



### Quiz 1

Considere o trecho de código em Java a seguir:

```
1 int i = 10;
2 boolean resultado = i++ < 10 & i++ < 20;
3 System.out.println(i++);
```

O que será impresso na tela?

- a) 10
- b) 11
- c) 12
- d) 13

Na linha 1, a variável *i* é inicializada com o valor 10.

Na linha 2, as duas comparações da expressão booleana “*i++ < 10 & i++ < 20*” são processadas pois utilizamos o operador lógico **&**. Portanto, a variável *i* é incrementada duas vezes, assumindo o valor 12.

Na linha 3, como o operador **++** foi aplicado à direita da variável *i*, o incremento ocorre depois da impressão. Dessa forma, o valor 12 é impresso na tela, ou seja, a resposta correta é a *c*.

**Obs:** Se o operador **&&** tivesse sido utilizado na linha 2 no lugar do operador **&**, o valor impresso seria 11. Com o operador **&&** a segunda comparação da expressão booleana “*i++ < 10 & i++ < 20*” não seria processada porque o resultado da primeira comparação é **false**. Dessa forma, a variável *i* seria incrementada apenas uma vez na linha 2.





## RESPOSTAS

### Exercício Complementar 2.1

```
1 public class Triangulo {  
2     public static void main(String[] args) {  
3         String linha = "*";  
4         for(int contador = 1; contador <= 10; contador++) {  
5             System.out.println(linha);  
6             linha += "*";  
7         }  
8     }  
9 }
```

Código Java 2.22: *Triangulo.java*

### Exercício Complementar 2.2

```
1 public class Triangulos {  
2     public static void main(String[] args) {  
3         String linha = "*";  
4         for(int contador = 1; contador <= 10; contador++) {  
5             System.out.println(linha);  
6             int resto = contador % 4;  
7             if(resto == 0) {  
8                 linha = "*";  
9             } else {  
10                 linha += "*";  
11             }  
12         }  
13     }  
14 }
```

Código Java 2.23: *Triangulos.java*

### Exercício Complementar 2.3

```
1 public class Fibonacci {  
2     public static void main(String[] args) {  
3         int penultimo = 0;  
4         int ultimo = 1;  
5  
6         System.out.println(penultimo);  
7         System.out.println(ultimo);  
8     }  
9 }
```

```
9  for(int contador = 0; contador < 28; contador++) {  
10     int proximo = penultimo + ultimo;  
11     System.out.println(proximo);  
12  
13     penultimo = ultimo;  
14     ultimo = proximo;  
15 }  
16 }  
17 }
```

Código Java 2.24: Fibonacci.java

### Exercício Complementar 3.1

```
1 public class Aluno {  
2     public String nome;  
3     public String rg;  
4     public String dataNascimento;  
5 }
```

Código Java 3.20: Aluno.java

### Exercício Complementar 3.2

```
1 public class TestaAluno {  
2     public static void main(String[] args) {  
3         Aluno a1 = new Aluno();  
4         a1.nome = "Marcelo Martins";  
5         a1.rg = "33333333-3";  
6         a1.dataNascimento = "02/04/1985";  
7  
8         Aluno a2 = new Aluno();  
9         a2.nome = "Rafael Cosentino";  
10        a2.rg = "22222222-2";  
11        a2.dataNascimento = "30/10/1984";  
12  
13        System.out.println("Dados do primeiro aluno");  
14        System.out.println("Nome: " + a1.nome);  
15        System.out.println("RG: " + a1.rg);  
16        System.out.println("Data de nascimento: " + a1.dataNascimento);  
17  
18        System.out.println("-----");  
19  
20        System.out.println("Dados do segundo aluno");  
21        System.out.println("Nome: " + a2.nome);  
22        System.out.println("RG: " + a2.rg);  
23        System.out.println("Data de nascimento: " + a2.dataNascimento);  
24    }  
25 }
```

Código Java 3.21: TestaAluno.java

### Exercício Complementar 3.3

```

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

*Código Java 3.22: Funcionario.java*

### Exercício Complementar 3.4

```

1 public class TestaFuncionario {
2     public static void main(String[] args) {
3         Funcionario f1 = new Funcionario();
4         f1.nome = "Marcelo Martins";
5         f1.salario = 1800.57;
6
7         Funcionario f2 = new Funcionario();
8         f2.nome = "Rafael Cosentino";
9         f2.salario = 2000.88;
10
11        System.out.println("Dados do primeiro funcionário");
12        System.out.println("Nome: " + f1.nome);
13        System.out.println("Salário: " + f1.salario);
14
15        System.out.println("-----");
16
17        System.out.println("Dados do segundo funcionário");
18        System.out.println("Nome: " + f2.nome);
19        System.out.println("Salário: " + f2.salario);
20    }
21 }
```

*Código Java 3.23: TestaFuncionario.java*

### Exercício Complementar 3.5

```

1 public class Turma {
2     public String periodo;
3     public int serie;
4     public String sigla;
5     public String tipoDeEnsino;
6 }
```

*Código Java 3.24: Turma.java*

### Exercício Complementar 3.6

```

1 public class TestaTurma {
2     public static void main(String[] aseries) {
3         Turma t1 = new Turma();
```

```

4     t1.periodo = "Tarde";
5     t1.serie = 8;
6     t1.sigla = "A";
7     t1.tipoDeEnsino = "Fundamental";
8
9     Turma t2 = new Turma();
10    t2.periodo = "Manha";
11    t2.serie = 5;
12    t2.sigla = "B";
13    t2.tipoDeEnsino = "Fundamental";
14
15    System.out.println("Dados da primeira turma");
16    System.out.println("Período: " + t1.periodo);
17    System.out.println("Série: " + t1.serie);
18    System.out.println("Sigla: " + t1.sigla);
19    System.out.println("Tipo de ensino: " + t1.tipoDeEnsino);
20
21    System.out.println("-----");
22
23    System.out.println("Dados da segunda turma");
24    System.out.println("Período: " + t2.periodo);
25    System.out.println("Série: " + t2.serie);
26    System.out.println("Sigla: " + t2.sigla);
27    System.out.println("Tipo de ensino: " + t2.tipoDeEnsino);
28 }
29 }
```

*Código Java 3.25: TestaTurma.java***Exercício Complementar 3.7**

```

1 public class Aluno {
2     public String nome;
3     public String rg;
4     public String dataNascimento;
5     public Turma turma;
6 }
```

*Código Java 3.33: Aluno.java***Exercício Complementar 3.8**

```

1 public class TestaAlunoTurma {
2     public static void main(String[] args) {
3         Turma t = new Turma();
4         Aluno a = new Aluno();
5
6         t.periodo = "Manha";
7         t.serie = 5;
8         t.sigla = "B";
9         t.tipoDeEnsino = "Fundamental";
10
11         a.nome = "Rafael Cosentino";
12         a.dataNascimento = "30/10/1984";
13         a.rg = "11111111";
14
15         System.out.println("Dados da turma");
16         System.out.println("Período: " + t.periodo);
```

```

17 System.out.println("Série: " + t.serie);
18 System.out.println("Sigla: " + t.sigla);
19 System.out.println("Tipo de ensino: " + t.tipoDeEnsino);
20
21 System.out.println("-----");
22
23 System.out.println("Dados do aluno");
24 System.out.println("Nome: " + a.nome);
25 System.out.println("Data de nascimento: " + a.dataNascimento);
26 System.out.println("RG: " + a.rg);
27
28 System.out.println("-----");
29
30 a.turma = t;
31
32 System.out.println("Dados da turma obtidos através do aluno");
33 System.out.println("Período: " + a.turma.periodo);
34 System.out.println("Série: " + a.turma.serie);
35 System.out.println("Sigla: " + a.turma.sigla);
36 System.out.println("Tipo de ensino: " + a.turma.tipoDeEnsino);
37 }
38 }
```

*Código Java 3.34: TesteAlunoTurma.java***Exercício Complementar 3.9**

```

1 public class Funcionario {
2     public String nome;
3     public double salario;
4
5     public void aumentaSalario(double valor) {
6         this.salario += valor;
7     }
8
9     public String consultaDados() {
10        return "Nome: " + this.nome + "\nSalário: " + this.salario;
11    }
12 }
```

*Código Java 3.42: Funcionario.java***Exercício Complementar 3.10**

```

1 public class TestaMetodosFuncionario {
2     public static void main(String[] args) {
3         Funcionario f1 = new Funcionario();
4
5         f1.nome = "Rafael Cosentino";
6         f1.salario = 1000;
7
8         System.out.println(f1.consultaDados());
9
10        System.out.println("-----");
11
12        f1.aumentaSalario(100);
13
14        System.out.println(f1.consultaDados());
```

```
15 }  
16 }
```

Código Java 3.43: TestaMetodosFuncionario.java

### Exercício Complementar 4.1

```
1 public class Media {  
2     public static void main(String[] args) {  
3         double soma = 0;  
4         for (String arg : args) {  
5             double d = Double.parseDouble(arg);  
6             soma += d;  
7         }  
8         System.out.println(soma / args.length);  
9     }  
10 }
```

Código Java 4.17: Media.java

Execute a classe `Media` e observe que a média não é exibida no Console já que nenhum parâmetro foi passado para o método `main`.

Defina os parâmetros que devem ser passados para o método `main` da classe `Media`. Para isso, digite “CTRL + 3” e pesquise por “Run Configurations”.

### Exercício Complementar 4.2

```
1 public class Maior {  
2     public static void main(String[] args) {  
3         double maior = Double.parseDouble(args[0]);  
4         for (int i = 1; i < args.length; i++) {  
5             double d = Double.parseDouble(args[i]);  
6             if (maior < d) {  
7                 maior = d;  
8             }  
9         }  
10        System.out.println(maior);  
11    }  
12 }
```

Código Java 4.18: Maior.java

Execute a classe `Maior` e observe que a média não é exibida no Console já que nenhum parâmetro foi passado para o método `main`.

Defina os parâmetros que devem ser passados para o método `main` da classe `Maior`. Para isso, digite “CTRL + 3” e pesquise por “Run Configurations”.

### Exercício Complementar 5.1

```

1 public class Funcionario {
2     public String nome;
3     public double salario;
4     public static double valeRefeicaoDiario;
5 }
```

*Código Java 5.18: Funcionario.java*

### Exercício Complementar 5.2

```

1 public class TestaValeRefeicao {
2     public static void main(String[] args) {
3         System.out.println(Funcionario.valeRefeicaoDiario);
4         Funcionario.valeRefeicaoDiario = 15;
5         System.out.println(Funcionario.valeRefeicaoDiario);
6     }
7 }
```

*Código Java 5.19: TestaValeRefeicao.java*

Execute a classe TestaValeRefeicao.

### Exercício Complementar 5.3

```

1 public class Funcionario {
2     public String nome;
3     public double salario;
4     public static double valeRefeicaoDiario;
5
6     public static void reajustaValeRefeicaoDiario(double taxa) {
7         Funcionario.valeRefeicaoDiario += Funcionario.valeRefeicaoDiario * taxa;
8     }
9 }
```

*Código Java 5.20: Funcionario.java*

### Exercício Complementar 5.4

```

1 class TestaValeRefeicao {
2     public static void main(String[] args) {
3         System.out.println(Funcionario.valeRefeicaoDiario);
4         Funcionario.valeRefeicaoDiario = 15;
5         System.out.println(Funcionario.valeRefeicaoDiario);
6
7         Funcionario.reajustaValeRefeicaoDiario(0.1);
8         System.out.println(Funcionario.valeRefeicaoDiario);
9     }
10 }
```

*Código Java 5.21: TestaValeRefeicao.java*

Execute a classe TestaValeRefeicao.

### Exercício Complementar 6.1

---

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

*Código Java 6.15: Conta.java*

### Exercício Complementar 6.2

---

```
1 public class Conta {  
2     private double numero;  
3     private double saldo;  
4     private double limite;  
5  
6     public double getNumero() {  
7         return numero;  
8     }  
9  
10    public void setNumero(double numero) {  
11        this.numero = numero;  
12    }  
13  
14    public double getSaldo() {  
15        return saldo;  
16    }  
17  
18    public void setSaldo(double saldo) {  
19        this.saldo = saldo;  
20    }  
21  
22    public double getLimite() {  
23        return limite;  
24    }  
25  
26    public void setLimite(double limite) {  
27        this.limite = limite;  
28    }  
29 }
```

*Código Java 6.16: Conta.java*

### Exercício Complementar 6.3

---

```

1 public class Testa {
2     public static void main(String[] args) {
3         Conta c = new Conta();
4
5         c.setNumero(123);
6         c.setSaldo(2000);
7         c.setLimite(1000);
8
9         System.out.println(c.getNumero());
10        System.out.println(c.getSaldo());
11        System.out.println(c.getLimite());
12    }
13 }
```

*Código Java 6.17: Testa.java***Exercício Complementar 7.1**

```

1 public 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 }
```

*Código Java 7.28: Funcionario.java***Exercício Complementar 7.2**

```

1 public 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 }
```

*Código Java 7.29: Gerente.java*

```

1 public class Telefonista extends Funcionario {
2     private int estacaoDeTrabalho;
3
4     public void mostraDados() {
5         super.mostraDados();
6         System.out.println("Estação de Trabalho " + this.estacaoDeTrabalho);
7     }
8
9     // GETTERS AND SETTERS
10 }
```

*Código Java 7.30: Telefonista*

```

1 public class Secretaria extends Funcionario {
2     private int ramal;
3
4     public void mostraDados() {
5         super.mostraDados();
6         System.out.println("Ramal " + this.ramal);
7     }
8
9     // GETTERS AND SETTERS
10 }
```

*Código Java 7.31: Secretaria.java***Exercício Complementar 7.3**

```

1 public 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 }
```

*Código Java 7.32: TestaFuncionarios.java***Exercício Complementar 8.1**

```

1 public class Funcionario {
2     private int codigo;
3
4     // GETTERS AND SETTERS
5 }
```

*Código Java 8.13: Funcionario.java*

### Exercício Complementar 8.2

```

1 public class Gerente extends Funcionario {
2     private String usuario;
3     private String senha;
4
5     // GETTERS AND SETTERS
6 }
```

*Código Java 8.14: Gerente.java*

```

1 public class Telefonista extends Funcionario {
2     private int ramal;
3
4     // GETTERS AND SETTERS
5 }
```

*Código Java 8.15: Telefonista.java*

### Exercício Complementar 8.3

```

1 import java.text.SimpleDateFormat;
2 import java.util.Date;
3
4 public class ControleDePonto {
5
6     public void registraEntrada(Funcionario f) {
7         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
8         Date agora = new Date();
9
10        System.out.println("ENTRADA: " + f.getCodigo());
11        System.out.println("DATA: " + sdf.format(agora));
12    }
13
14     public void registraSaida(Funcionario f) {
15         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
16         Date agora = new Date();
17
18        System.out.println("SAÍDA: " + f.getCodigo());
19        System.out.println("DATA: " + sdf.format(agora));
20    }
21 }
```

*Código Java 8.16: ControleDePonto.java*

**Exercício Complementar 8.4**

```
1 public class TestaControleDePonto {  
2     public static void main(String[] args) {  
3         Gerente g = new Gerente();  
4         g.setCodigo(1);  
5         g.setUsuario("rafael.cosentino");  
6         g.setSenha("12345");  
7  
8         Telefonista t = new Telefonista();  
9         t.setCodigo(2);  
10        t.setRamal(13);  
11  
12        ControleDePonto cdp = new ControleDePonto();  
13  
14        cdp.registraEntrada(g);  
15        cdp.registraEntrada(t);  
16  
17        cdp.registraSaida(g);  
18        cdp.registraSaida(t);  
19    }  
20 }
```

Código Java 8.17: *TestaControleDePonto.java*

**Exercício Complementar 10.1**

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

Código Java 10.18: *Funcionario.java*

**Exercício Complementar 10.2**

```
1 public class TestaFuncionario {  
2     public static void main(String[] args) {  
3         Funcionario f = new Funcionario();  
4  
5         f.setSalario(3000);  
6  
7         System.out.println(f.getSalario());  
8     }  
9 }
```

Código Java 10.19: *TestaFuncionario.java*

**Exercício Complementar 10.3**

```
1 public abstract class Funcionario {  
2     private double salario;  
3  
4     // GETTERS AND SETTERS  
5 }
```

*Código Java 10.20: Funcionario.java*

A classe TestaFuncionario não compila pois não é permitido criar objetos de classes abstratas.

#### Exercício Complementar 10.4

---

```
1 public class Gerente extends Funcionario {  
2     private String usuario;  
3     private String senha;  
4  
5     // GETTERS E SETTERS  
6 }
```

*Código Java 10.21: Gerente.java*

#### Exercício Complementar 10.5

---

```
1 public class TestaFuncionario {  
2     public static void main(String[] args) {  
3         Funcionario f = new Gerente();  
4  
5         f.setSalario(3000);  
6  
7         System.out.println(f.getSalario());  
8     }  
9 }
```

*Código Java 10.22: TestaFuncionario.java*

#### Exercício Complementar 10.6

---

```
1 public abstract class Funcionario {  
2     private double salario;  
3  
4     public abstract double calculaBonificacao();  
5  
6     // GETTERS AND SETTERS  
7 }
```

*Código Java 10.23: Funcionario.java*

**Exercício Complementar 10.7**

A classe Gerente não compila pois o método calculaBonificacao não foi implementado.

**Exercício Complementar 10.8**

```
1 public class Gerente extends Funcionario {  
2     private String usuario;  
3     private String senha;  
4  
5     public double calculaBonificacao() {  
6         return this.getSalario() * 0.2 + 300;  
7     }  
8  
9     // GETTERS E SETTERS  
10 }
```

Código Java 10.24: Gerente.java

**Exercício Complementar 10.9**

```
1 public class TestaFuncionario {  
2     public static void main(String[] args) {  
3         Funcionario f = new Gerente();  
4  
5         f.setSalario(3000);  
6  
7         System.out.println(f.getSalario());  
8  
9         System.out.println(f.calculaBonificacao());  
10    }  
11 }
```

Código Java 10.25: TestaFuncionario.java

**Exercício Complementar 15.1**

```
1 package br.com.k19.testes;  
2  
3 import java.io.FileInputStream;  
4 import java.io.FileOutputStream;  
5 import java.io.IOException;  
6 import java.io.InputStream;  
7 import java.io.PrintStream;  
8 import java.util.Scanner;  
9  
10 public class ArquivoParaArquivo {  
11     public static void main(String[] args) throws IOException {  
12         InputStream arquivo1 = new FileInputStream("entrada.txt");  
13         Scanner scanner = new Scanner(arquivo1);  
14     }  
15 }
```

```
15  FileOutputStream arquivo2 = new FileOutputStream("saída.txt");
16  PrintStream printStream = new PrintStream(arquivo2);
17
18  while (scanner.hasNextLine()) {
19      String linha = scanner.nextLine();
20      printStream.println(linha);
21  }
22
23  scanner.close();
24  printStream.close();
25 }
26 }
```

Código Java 15.10: ArquivoParaArquivo.java

## Exercício Complementar 15.2

```
1 package br.com.k19.testes;
2
3 import java.io.FileOutputStream;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.io.PrintStream;
7 import java.util.Scanner;
8
9 public class TecladoParaArquivo {
10    public static void main(String[] args) throws IOException {
11        InputStream teclado = System.in;
12        Scanner scanner = new Scanner(teclado);
13
14        FileOutputStream arquivo = new FileOutputStream("saída.txt");
15        PrintStream printStream = new PrintStream(arquivo);
16
17        while (scanner.hasNextLine()) {
18            String linha = scanner.nextLine();
19            printStream.println(linha);
20        }
21
22        scanner.close();
23        printStream.close();
24    }
25 }
```

Código Java 15.11: TecladoParaArquivo.java