# Introdução à Programação Orientada a Objetos

*Rogério da Silva Batista*

*Rafael Araújo de Moraes*

Curso Técnico em Informática

INSITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA PIAUÍ

UNIVERSIDADE FEDERAL DE SANTA CATARINA

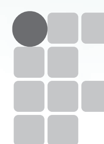
PDE | PRONATEC
PROGRAMA NACIONAL DE ACESSO AO ENSINO TÉCNICO E EMPREGO

Ministério da Educação

# Introdução à Programação Orientada a Objetos

*Rogério da Silva Batista*

*Rafael Araújo de Moraes*




**Teresina – PI**
**2013**

**Presidência da República Federativa do Brasil**

**Ministério da Educação**

**Secretaria de Educação Profissional e Tecnológica**

© Instituto Federal de Educação, Ciência e Tecnologia do Piauí
Este Caderno foi elaborado em parceria entre o Instituto Federal de Educação, Ciência e Tecnologia do Piauí e a Universidade Federal de Santa Catarina para a Rede e-Tec Brasil.

**Equipe de Elaboração**
Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI

**Coordenação Institucional**
Francieric Alves de Araujo/IFPI

**Coordenação do Curso**
Thiago Alves Elias da Silva/IFPI

**Professores-autores**
Rogério da Silva Batista/IFPI
Rafael Araújo de Moraes/IFPI

**Comissão de Acompanhamento e Validação**
Universidade Federal de Santa Catarina – UFSC

**Coordenação Institucional**
Araci Hack Catapan/UFSC

**Coordenação do Projeto**
Silvia Modesto Nassar/UFSC

**Coordenação de Design Instrucional**
Beatriz Helena Dal Molin/UNIOESTE e UFSC

**Coordenação de Design Gráfico**
Juliana Tonietto/UFSC

**Design Instrucional**
Gustavo Pereira Mateus/UFSC
Juliana Leonardi/UFSC

**Web Master**
Rafaela Lunardi Comarella/UFSC

**Web Design**
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

**Diagramação**
Liana Domeneghini Chiaradia/UFSC
Marília Ceriolli Hermoso/UFSC

**Revisão**
Júlio César Ramos/UFSC

**Projeto Gráfico**
e-Tec/MEC

**Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina**

B333i Batista, Rogério da Silva
Introdução à programação orientada a objetos
/ Rogério da Silva Batista, Rafael Araújo de Moraes. – Teresina : Instituto Federal de Educação, Ciência e Tecnologia do Piauí, 2013. 118 p. : il., tabs.

Inclui bibliografia
ISBN: 978-85-67082-01-1

1.Programação orientada a objetos (Computação).
I. Moraes, Rafael Araújo de. II. Título.

CDU: 681.31:519.689

# Apresentação e-Tec Brasil

Bem-vindo a Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira propiciando caminho de o acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (SETEC) e as instâncias promotoras de ensino técnico como os Institutos Federais, as Secretarias de Educação dos Estados, as Universidades, as Escolas e Colégios Tecnológicos e o Sistema S.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Fevereiro de 2013

Nosso contato
etecbrasil@mec.gov.br

# Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

**Atenção:** indica pontos de maior relevância no texto.

**Saiba mais:** oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

**Glossário:** indica a definição de um termo, palavra ou expressão utilizada no texto.

**Mídias integradas:** sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

**Atividades de aprendizagem:** apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

# Sumário

**Palavra do professor-autor** 9

**Apresentação da disciplina** 11

**Projeto instrucional** 13

**Aula 1 – Introdução ao Java** 15
- 1.1 Histórico 15
- 1.2 Características 16
- 1.3 A Máquina Virtual Java – *Java Virtual Machine* (JVM) 17
- 1.4 Etapas de um programa Java 18
- 1.5 *Hotspot* e JIT 19
- 1.6 Versões do JAVA 19
- 1.7 JRE e JDK 20
- 1.8 Compilando o primeiro programa Java 21
- 1.9 Executando o primeiro programa Java 22

**Aula 2 – Tipos de dados, operadores e controle de fluxo** 25
- 2.1 Tipos de dados 25
- 2.2 Declaração de variáveis 26
- 2.3 Conversões entre tipos 27
- 2.4 Operadores 30
- 2.6 Constantes 33
- 2.7 Controle de fluxo 33
- 2.8 Escopo de variáveis 40

**Aula 3 – Orientação a objetos: conceitos básicos** 45
- 3.1 Introdução 45
- 3.2 Modelos 46
- 3.3 Objetos 52
- 3.4 Classes 56
- 3.5 Troca de mensagens entre objetos 59
- 3.6 Criando classes em Java 62
- 3.7 Escopo 67

3.8 Encapsulamento 69
3.9 Criando objetos e acessando dados encapsulados 71
3.10 Criando aplicações em Java 74

**Aula 4 – Construtores, sobrecarga, atributos e métodos de classe 83**
4.1 Introdução 83
4.2 O que são construtores? 86
4.3 Sobrecarga de métodos 87
4.4 Cuidados com sobrecarga de métodos 88

**Aula 5 – Campos e métodos estáticos 91**
5.1 Introdução 91
5.2 Campos estáticos em classe 91
5.3 Métodos estáticos em classe 97
5.4 Campos e métodos estáticos em aplicações 100
5.5 Fábricas de instâncias 102

**Aula 6 – Reutilização de classes 107**
6.1 Introdução 107
6.2 Herança 107
6.3 Polimorfismo 110
6.4 Criação de um objeto de uma subclasse e o ponteiro *super* 113
6.5 Método final e classe final 115

**Referências 118**

# Palavra do professor-autor

Caro estudante,

Bem-vindo à disciplina Introdução à Programação Orientada a Objetos. Este material foi elaborado com o objetivo de contribuir para o desenvolvimento de seus estudos e para a ampliação de seus conhecimentos sobre o assunto tratado nesta disciplina.

Um grande abraço!

# Apresentação da disciplina

Este texto é destinado aos estudantes aprendizes que participam do programa Escola Técnica Aberta do Brasil (e-Tec Brasil), vinculado à Escola Técnica Aberta do Piauí (ETAPI) do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI), com apoio das prefeituras municipais dos respectivos polos: Alegrete do Piauí, Batalha, Monsenhor Gil e Valença do Piauí. O caderno é composto de seis aulas, assim distribuídas:

Na **Aula 1** faremos uma introdução à linguagem Java, abordaremos algumas características, versões, instalação, compilação e execução. A linguagem Java será a nossa linguagem de programação padrão utilizada neste curso, e é através dela que aplicaremos na prática os conceitos do paradigma orientado a objeto.

Na **Aula 2** abordaremos a linguagem Java com mais detalhes sobre suas estruturas de controle e seus comandos.

Na **Aula 3** começaremos de fato a estudar o paradigma orientado a objetos. Nessa aula vamos entender alguns conceitos fundamentais como classes, objetos, encapsulamento e vamos aplicar esses conceitos por meio de definição de classes e aplicações em Java.

Na **Aula 4** definiremos o que são e para que servem os construtores de classes e vamos aprender o conceito de sobrecarga de métodos para utilizarmos em nossas aplicações.

Na **Aula 5** conheceremos o que são campos e métodos estáticos e como utilizá-los em nosso código; vamos também aprender como diferenciar métodos estáticos de métodos não estáticos e também como declarar campos que serão compartilhados entre todas as instâncias de uma classe.

Na **Aula 6** vamos conhecer herança e polimorfismo, que são os conceitos fundamentais que diferenciam a linguagem orientada a objetos das demais linguagens. Vamos aprender a utilizar esses conceitos em nosso código e a criar objetos de uma subclasse e utilizar o método e classe final.

# Projeto instrucional

**Disciplina**: Introdução à Programação Orientada a Objetos (carga horária: 90h).

**Ementa**: Introdução ao Java: histórico, características e versões. Tipos de dados, variável, constantes, operações e estrutura condicional e de repetição. Conceitos básicos de orientação a objeto: classe, objeto, encapsulamento, polimorfismo e herança. Criação de construtores, sobrecarga, métodos estáticos.

| AULA | OBJETIVOS DE APRENDIZAGEM | MATERIAIS | CARGA HORÁRIA (horas) |
|---|---|---|---|
| 1. Introdução ao Java | Conhecer o conceito de Java, suas características e aplicações e ambiente de desenvolvimento.<br>Identificar as fases de um programa em Java.<br>Entender o processo de compilação e execução de um programa simples. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de ensino-aprendizagem (AVEA). | 15 |
| 2. Tipos de dados, operadores e controle de fluxo | Conhecer os tipos primitivos da linguagem Java bem como os operadores utilizados.<br>Aprender a trabalhar com os recursos da linguagem Java.<br>Entender o que é um escopo de uma variável. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de ensino-aprendizagem (AVEA). | 15 |
| 3. Orientação a objetos – conceitos básicos | Compreender o que é e para que serve orientação a objetos.<br>Criar modelos para representação de informações em um determinado contexto.<br>Conceituar objetos, classes, atributos e comportamentos.<br>Entender o significado da criação e uso de variáveis de referência e objetos na memória.<br>Implementar o encapsulamento através modificadores de acesso nas definições de classe.<br>Implementar classes executáveis através do método main para manipulação de classes através de seus objetos criados. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de ensino-aprendizagem (AVEA). | 15 |
| 4. Construtores, sobrecarga, atributos e métodos de classe | Conhecer o que são e para que servem os construtores.<br>Criar construtores para classes.<br>Compreender a sobrecarga de métodos e quais os principais cuidados que devemos ter com ela. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de ensino-aprendizagem (AVEA). | 15 |

Continua

| AULA | OBJETIVOS DE APRENDIZAGEM | MATERIAIS | CARGA HORÁRIA (horas) |
| --- | --- | --- | --- |
| 5. Campos e métodos estáticos | Compreender o que são campos e métodos estáticos e como utilizá-los para escrever melhor seu programa em Java.<br>Saber diferenciar método estático de métodos não estáticos. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de enssino--aprendizagem (AVEA). | 15 |
| 6. Reutilização de classe | Compreender os principais meios de reutilização de classes.<br>Compreender a utilização do conceito de herança, polimorfismo, dentre outros.<br>Aprender os dois tipos de polimorfismo e como aplicá-los em nosso código;<br>Aprender como criar objetos de uma subclasse.<br>Utilizar o método e classe final. | Caderno diadático da disciplina.<br>Material disponibilizado no ambiente virtual de enssino--aprendizagem (AVEA). | 15 |
| | | | **Conclusão** |

# Aula 1 – Introdução ao Java

## Objetivos

Conhecer o conceito de Java, suas características, aplicações e ambiente de desenvolvimento.

Entender as fases de um programa em Java.

Entender o processo de compilação e execução de um programa simples.

## 1.1 Histórico

A linguagem de programação Java foi criada em 1992 por James Gosling e iniciou como parte do projeto Green, da Sun Microsystems, com a finalidade de desenvolver inovações tecnológicas. A equipe responsável teve a ideia de criar um interpretador (veremos o que é isso mais adiante) para pequenos dispositivos, facilitando a reescrita de *software* para aparelhos eletrônicos, como videocassete, televisão e aparelhos de TV a cabo.

A ideia não deu certo. Tentaram fechar diversos contratos com grandes fabricantes de eletrônicos, como a Panasonic, mas não houve êxito, devido ao conflito de interesses. Com o advento da *web*, a Sun percebeu que poderia utilizar a ideia criada em 1992 para possibilitar rodar pequenas aplicações dentro de um navegador. A semelhança era que na internet havia uma grande quantidade de sistemas operacionais e navegadores e, com isso, seria grande vantagem poder programar numa única linguagem, independentemente da plataforma. Inicialmente a linguagem iria chamar-se **Oak**. A mudança de nome ocorreu porque já existia uma linguagem de programação com esse nome; então, a linguagem foi rebatizada para **Java**.

## 1.2 Características

O termo Java é utilizado, geralmente, quando nos referimos a:

- linguagem de programação orientada a objetos;
- ambiente de desenvolvimento composto pelo compilador, interpretador, gerador de documentação etc.;
- ambiente de execução que pode ser praticamente qualquer máquina que possua *Java Runtime Environment* (JRE) instalado.

A linguagem de programação Java é uma linguagem de alto nível, com as seguintes características:

- **Simples**: O aprendizado da linguagem de programação Java pode ser feito em um curto período de tempo, pois é muito familiar para quem programa em C/C++. Possui um pequeno numero de construções de linguagem e elimina algumas das construções mais complicadas como: desvio incondicional (goto), herança múltipla e ponteiros.
- **Orientada a objetos**: Desde o início do seu desenvolvimento, essa linguagem foi projetada para ser orientada a objetos. Para isso, atende a todos os requisitos como: oferecer mecanismos de abstração, encapsulamento e hereditariedade.
- **Robusta**: Ela foi pensada para o desenvolvimento de *softwares* confiáveis, provendo verificações tanto em tempo de execução quanto compilação. O coletor de lixo responsabiliza-se pela limpeza da memória quando houver necessidade, evitando erros de uso dessa memória.
- **Segura**: Aplicações Java são executadas em ambiente próprio *Java Runtime Environment* (JRE), o que inviabiliza a intrusão de código malicioso.
- **Portável**: Programas desenvolvidos nessa linguagem podem ser executados em praticamente qualquer máquina, desde que esta possua o JRE instalado.

## 1.3 A Máquina Virtual Java – *Java Virtual Machine* (JVM)

Em uma linguagem de programação como C e Pascal, temos o seguinte quadro quando vamos compilar um programa (Figura 1.1).

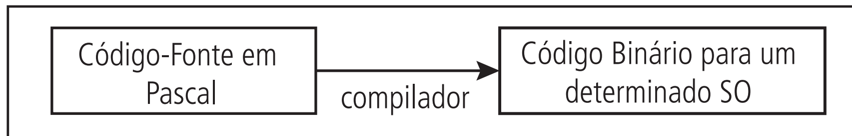



**Figura 1.1: Processo de compilação em uma linguagem de programação Pascal**
Fonte: Elaborada pelos autores

O código-fonte é compilado para uma plataforma e sistema operacional específicos. Muitas vezes, o próprio código-fonte é desenvolvido visando a uma única plataforma. Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber "conversar" com o sistema operacional em questão.

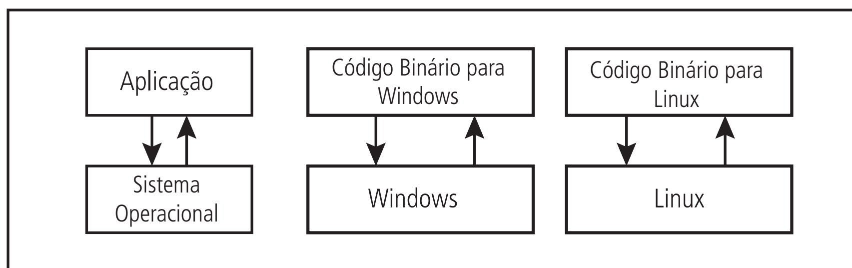



**Figura 1.2: Exemplo de código executável para cada sistema operacional**
Fonte: Elaborada pelos autores

Isto é, temos um código executável para cada sistema operacional. É necessário compilar uma vez para Windows, outra para o Linux, etc.

Como foi dito anteriormente, na maioria das vezes, a sua aplicação se utiliza das bibliotecas do sistema operacional, como, por exemplo, a de interface gráfica para desenhar as telas de um sistema. A biblioteca de interface gráfica do Windows é bem diferente da biblioteca do Linux, ou seja, o programador é obrigado a reescrever o mesmo pedaço de código (referente à interface gráfica) para diferentes sistemas operacionais, já que eles não são compatíveis.

A máquina virtual Java (JVM) é uma máquina imaginária que emula uma aplicação em uma máquina real. É a JVM que permite a portabilidade do có-

digo Java; isso ocorre porque todo código Java é compilado para um formato intermediário, o *bytecode*. Esse formato é então interpretado pela JVM. Existem diversas JVMs, cada uma delas destinada a um tipo de sistema operacional (Windows, Linux, Mac, etc.). Dessa forma, sendo o código de uma aplicação Java transformada em *bytecode* e interpretada pela JVM, podemos desenvolver uma aplicação sem nos preocuparmos onde ela será executada, pois uma vez que a JVM está instalada, nossa aplicação irá executar sem nenhum envolvimento com o sistema operacional.

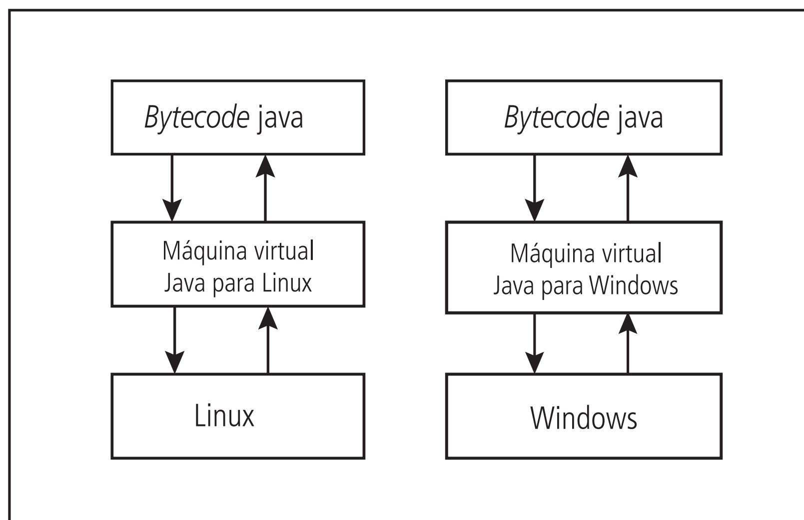



**Figura 1.3: Relação entre o *bytecode*, a máquina virtual e o sistema operacional**
Fonte: Elaborada pelos autores

## 1.4 Etapas de um programa Java

As etapas pelas quais passa um programa relacionam-se da seguinte forma:

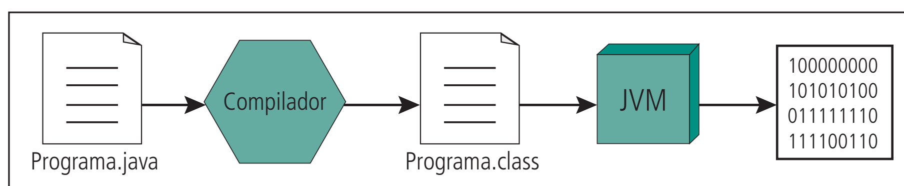



**Figura 1.4: Etapas de um programa Java**
Fonte: Curso Java Starter (www.t2ti.com)

1. criação do código-fonte (Programa.Java);

2. compilação do código-fonte e geração do *bytecode* (Programa.class);

3. interpretação do *bytecode* pela máquina virtual;

4. conversão do *bytecode* em linguagem de máquina.

## 1.5 *Hotspot* e JIT

*Hotspot* é a tecnologia que a JVM utiliza para detectar pontos quentes da sua aplicação: código que é executado, muito provavelmente, dentro de um ou mais *loops*. Quando a JVM julgar necessário, ela vai compilar aquele código para instruções nativas da plataforma, tendo em vista que isso vai provavelmente melhorar a *performance* da sua aplicação. Esse compilador é o JIT: *Just in Time Compiler*, o compilador que aparece "bem na hora" que você precisa.

Você pode pensar então: por que a JVM não compila tudo antes de executar a aplicação? É que, teoricamente, compilar dinamicamente, na medida do necessário, pode gerar uma *performance* melhor. O motivo é simples: imagine um ".exe" gerado pelo VisualBasic, pelo gcc ou pelo Delphi; ele é estático. Em princípio podemos pensar que o fato de o programa não precisar passar por uma etapa a mais, a interpretação, irá torná-lo mais eficiente, mas muitas vezes a compilação estática não consegue prever situações que irão ocorrer durante a execução do código: trechos da aplicação mais utilizados, carga do sistema, quantidade de usuários simultâneos, memória disponível, etc. Ou seja, o compilador pode ter tomado uma decisão não tão boa.

Já a JVM, por estar compilando dinamicamente durante a execução, pode perceber que um determinado código não está com *performance* adequada e otimizar mais um pouco aquele trecho, ou ainda mudar a estratégia de otimização. É por esse motivo que as JVMs mais recentes, chegam a ganhar, em muitos casos, de códigos C compilados com o GCC 3.x, se rodados durante certo tempo.

## 1.6 Versões do JAVA

Java 1.0 e 1.1 são versões muito antigas do Java. Com o Java 1.2 houve um aumento grande no tamanho da API, e foi nesse momento que trocaram a nomenclatura de Java para Java2, com o objetivo de diminuir a confusão entre Java e Javascript. Mas lembre-se: não há versão do Java 2.0, o 2 foi incorporado ao nome: Java2 1.2. Depois vieram o Java2 1.3 e 1.4, e o Java 1.5 passou a se chamar Java 5, tanto por uma questão de *marketing* quanto porque mudanças significativas na linguagem foram incluídas. É nesse momento que o "2" do nome Java desaparece. Repare que para fins de desenvolvimento, o Java 5 ainda é referido como Java 1.5. Até a versão 1.4, existia a terceira numeração (1.3.1, 1.4.1, 1.4.2, etc.), indicando *bug fixes* e melhorias. A partir do Java 5 existem apenas *updates*: Java 5 *update* 7, por exemplo.

Hoje a última versão disponível do Java é a 6.0, lançada em 2006. Da versão 1.4 para a 5.0, a linguagem sofreu muitas modificações, o que de certa forma fomentou a mudança no versionamento do Java. Já o Java 6.0 não trouxe nenhuma mudança na linguagem, mas trouxe mais recursos na API e muitas melhorias de *performance* na VM.

Existe compatibilidade para trás em todas as versões do Java. Um *class* gerado pelo Javac da versão 1.2 precisa necessariamente rodar na JVM 6.0. Por isso, recomendamos sempre usar a última versão do Java para usufruir das tecnologias mais modernas, mas sem correr o risco de quebrar aplicações antigas.

## 1.7 JRE e JDK

- **JRE**: O *Java Runtime Environment* contém tudo aquilo que um usuário comum precisa para executar uma aplicação Java (JVM e bibliotecas), como o próprio nome diz é o "Ambiente de execução Java";

- **JDK**: O *Java Development Kit* é composto pelo JRE e um conjunto de ferramentas úteis ao desenvolvedor Java.

### 1.7.1 Ferramentas do JDK

A seguir temos uma breve descrição das principais ferramentas que fazem parte do JDK:

- **javac**: compilador da linguagem Java;

- **Java**: interpretador Java;

- **jdb**: debugador Java;

- **Java-prof**: interpretador com opção para gerar estatísticas sobre o uso dos métodos;

- **javadoc**: gerador de documentação;

- **jar**: ferramenta que comprime, lista e expande;

- **appletviewer**: permite a execução e *debug* de *applets* sem *browser*;

- **javap**: permite ler a interface pública das classes;

- **extcheck**: detecta conflitos em arquivos Jar.

## 1.8 Compilando o primeiro programa Java

Vamos para o nosso primeiro código. O programa que imprime uma linha simples.

```
1  class MeuPrograma {
2      public static void main(String[] args) {
3          System.out.println("Meu primeiro programa Java!!");
4      }
5  }
```

**Figura 1.5: Nosso primeiro programa**
Fonte: Elaborada pelos autores

Todos os códigos apresentados neste caderno estão formatados com recursos visuais para auxiliar a leitura e compreensão. Quando for digitar os códigos no computador, trate-os como texto simples. A numeração das linhas não faz parte do código e não deve ser digitada; é apenas um recurso didático. O Java é *case sensitive*: tome cuidado com maiúsculas e minúsculas. O nome do arquivo fonte do programa (.Java) deverá ter sempre o mesmo nome da classe. No Exemplo acima, o arquivo deverá ser salvo com o nome: MeuPrograma.java.

Você deverá digitar o código acima em algum editor de textos de sua preferência.

No Linux, recomendamos o uso do gedit ou do kate. No Windows, você pode usar o Notepad ou Notepad++.  No Mac, TextWrangler ou TextMate.

Após digitar o código acima, grave-o como MeuPrograma.java em algum diretório. Para compilar, você deve pedir para que o compilador de Java da Sun, chamado javac, gere o *bytecode* correspondente ao seu código Java.

Sintaxe no console do Sistema Operacional:

```
/home/aluno/teste$ java MeuPrograma.java
```

**Figura 1.6: Sintaxe**
Fonte: Elaborada pelos autores

Depois de compilar, o *bytecode* foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo ".class" foi gerado, com o mesmo nome da sua classe Java.

```
/home/aluno/teste$ ls -l
-rw-r--r-- ...   440 2009-12-01 20:20  MeuPrograma.class
-rw-r--r-- ...   310 2009-12-01 20:19  MeuPrograma.java
/home/aluno/teste$
```

**Figura 1.7: Visualização de arquivos no console**
Fonte: Elaborada pelos autores

## 1.9 Executando o primeiro programa Java

Os procedimentos para executar seu programa são muito simples. O comando **javac** é o compilador Java, e o comando **java** é o responsável por invocar a máquina virtual para interpretar o seu programa. Sintaxe no console do Sistema Operacional:

```
/home/aluno/teste$ java MeuPrograma
 Meu primeiro programa Java!!
/home/aluno/teste$
```

**Figura 1.8: Executando o programa em Java**
Fonte: Elaborada pelos autores

O corpo do programa (representado pela linha 3) é o que será executado quando chamamos a máquina virtual. Por enquanto, todas as linhas anteriores, onde há a declaração de uma classe e a de um método, não importam para nós agora. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e esse ponto de entrada é um método *main*. O *main* é o método que inicia as aplicações Java; quando solicitamos ao interpretador que execute uma determinada classe compilada, ele procura o método *main*; se esse método não existir, irá ser gerada uma exceção informando que o método não foi localizado.

Ainda não sabemos o que é método, mas veremos mais adiante. Até lá, não se preocupe com essas declarações. Sempre que um exercício for feito, o código a ser executado sempre estará nesse corpo.

No caso do nosso código, a linha: "System.out.println" faz com que o conteúdo entre aspas seja escrito na tela.

## Resumo

Nesta aula abordamos conceitos iniciais sobre a linguagem Java, suas características e versões; a diferença entre a JRE e a JDK, de acordo com o uso ou o desenvolvimento de programas Java; vimos a importância da Máquina Virtual Java – *Java Virtual Machine* (JVM) como fator fundamental para a portabilidade do Java e seu objetivo junto ao processo de compilação e execução de um programa. Criamos a primeira aplicação Java como exemplo para mostrar o uso dos comandos mais importantes da JDK, que são: javac e java, que têm, respectivamente, como objetivos: a compilação de um programa fonte em Java(.java), cujo resultado é o *bytecode* gerado ".class" e a interpretação do *bytecode* gerado pela JVM.

## Atividades de aprendizagem

1. Explique qual a função da Máquina Virtual Java (JVM).

2. Qual a diferença entre JRE e JDK?

3. Crie um programa Java que imprima o seguinte texto "Terminei a primeira aula com um programa Java!".

4. Compile o programa desenvolvido no exercício anterior. A seguir apague o arquivo .class gerado e tente executar o programa. O que aconteceu?

5. Mude o nome do método "main" para "start", compile e execute. O que aconteceu?

6. Crie um programa Java para imprimir duas linhas de texto usando duas linhas de código "System.out", onde aparecerá o seu nome na primeira linha e na segunda linha aparecerá o time para o qual você torce.

7. Experimente escrever todo o programa anterior em maiúsculo, compile e execute. O que aconteceu?

8. Experimente salvar o arquivo com um nome diferente do nome da classe, compile e execute. O que aconteceu?

Poste suas respostas no AVEA.

# Aula 2 – Tipos de dados, operadores e controle de fluxo

## Objetivos

Conhecer os tipos primitivos da linguagem Java bem como os operadores utilizados.

Aprender a trabalhar com os recursos da linguagem Java.

Entender o que é um escopo de uma variável.

## 2.1 Tipos de dados

Como podemos observar, a linguagem Java oferece diversos tipos de dados com os quais podemos trabalhar. Há basicamente duas categorias em que se encaixam os tipos de dados: tipos primitivos e tipos de referências. Os tipos primitivos correspondem a dados mais simples ou escalares, enquanto os tipos de referências consistem em *arrays*, classes e interfaces. Existem oito tipos primitivos em Java. Seis deles são numéricos, um é o caráter e o outro é o booleano. Os tipos inteiros guardam valores numéricos sem parte fracionária. Valores negativos são permitidos. Vamos a uma descrição curta sobre cada um dos tipos mostrado no Quadro 2.1 a seguir.

**Quadro 2.1: Tipos de dados primitivos em Java**

| Tipo | Descrição |
|---|---|
| ***boolean*** | Pode ser contido em um *bit*, porém o seu tamanho não é precisamente definido. Assume os valores *true* ou *false*. |
| ***char*** | Caractere em notação Unicode de 16 *bits*. Serve para armazenar dados alfanuméricos. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535. |
| ***byte*** | Inteiro de oito *bits* em notação de complemento de dois. Pode assumir valores entre -27=-128 e 27-1=127. |
| ***short*** | Inteiro de 16 *bits* em notação de complemento de dois. Os valores possíveis cobrem a faixa de -2-15=-32.768 a 215-1=32.767 |
| ***int*** | Inteiro de 32 *bits* em notação de complemento de dois. Pode assumir valores entre -231=2.147.483.648 e 231-1=2.147.483.647. |
| ***long*** | Inteiro de 64 *bits* em notação de complemento de dois. Pode assumir valores entre -263 e 263-1. |
| | **Continua** |

| | |
|---|---|
| ***float*** | Representa números em notação de ponto flutuante normalizada em precisão simples de 32 *bits* em conformidade com a norma IEEE 754-1985. O menor valor positivo representável por esse tipo é 1.40239846e-46 e o maior é 3.40282347e+38. Quatro bytes de tamanho e 23 dígitos binários de precisão. |
| ***double*** | Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 *bits* em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é 4.94065645841246544e-324 e o maior é 1.7976931348623157e+308. Oito bytes de tamanho e 52 dígitos binários de precisão. |
| | **Conclusão** |

## 2.2 Declaração de variáveis

A declaração de variáveis em Java, como em várias outras linguagens, exige que o tipo da variável seja declarado. Você inicia a declaração indicando o tipo da variável e o nome desejado, como no exemplo:

```
tipoDaVariavel  nomeDaVariavel;
```

Por exemplo, é possível ter uma idade que vale um número inteiro:

```
int idade;
```

Com isso, você declara a variável idade, que passa a existir a partir deste momento. Ela é do tipo *int*, que guarda um número inteiro. A partir de agora você pode usá-la, primeiro atribuindo valores.

A linha a seguir é a tradução de "idade deve valer agora quinze".

```
idade = 15;
```

É sempre interessante a colocação de comentários em programas. Os comentários permitem que a manutenção posterior do código seja mais rápida e servem para indicar o que o programa faz. Os comentários em Java podem ser de dois tipos. Utilizam-se duas barras ( // ) em qualquer posição da linha. Tudo o que aparecer à direita das duas barras será ignorado pelo compilador. Exemplo:

```
idade=15; // atribuição do valor inteiro 15 à variável idade
```

Existem ocasiões em que várias linhas de comentário são necessárias. Nesse caso, utilizamos os sinais de /* e */ para indicar início e fim de bloco de comentários, como no exemplo:

```
/* Programa de Exemplo – Esse programa não faz nada.
Criado por Maria
Versao 1.0
*/
```

Note que todas as declarações terminam com o ponto e vírgula. Os nomes das variáveis devem ser iniciados com qualquer letra, seguidas por uma sequência de letras ou números. O tamanho do nome da variável não tem limites. É possível declarar várias variáveis em uma linha, bem como atribuir valores a elas na declaração, como nos exemplos abaixo:

```
int a,b;
int idade = 15; // Isto é uma inicialização
```

## 2.3 Conversões entre tipos

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo *double*, tentar atribuir ele a uma variável *int* não funciona porque é um código que diz: "i deve valer d", mas não se sabe se d realmente é um número inteiro ou não.

```
double d = 3.1415;
int i = d; // não compila
O mesmo ocorre no seguinte trecho:
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um *int*, o compilador não tem como saber que valor estará dentro desse *double* no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores. Já no caso a seguir, é o contrário:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um *double* pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo *int* podem ser guardados em uma variável *doublé*; então, não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado como número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja moldado (*casted*) como um número inteiro. Esse processo recebe o nome de *casting*.

```
double d3 = 3.14;
int i = (int) d3;
```

O *casting* foi feito para moldar a variável d3 como um *int*. O valor de i agora é 3. O mesmo ocorre entre valores *int* e *long*.

```
long x = 10000;
int i = x; // não compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o *casting*:

```
long x = 10000;
int i = (int) x;
Alguns castings aparecem também:
float x = 0.0;
```

O código acima não compila, pois todos os literais com ponto flutuante são considerados *double* pelo Java. E *float* não pode receber um *double* sem perda de informação; para fazer isso funcionar, podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra f, que pode ser maiúscula ou minúscula, indica que aquele literal deve ser tratado como *float*.

Outro caso, que é mais comum:

```
double d = 5;
float f = 3;
float x = f + (float) d;
```

Você precisa do *casting* porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o *double*. E uma observação: no mínimo, o Java armazena o resultado em um int, na hora de fazer as contas. Até *casting* com variáveis do tipo *char* podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o *boolean*.

As conversões de tipos em Java sem perda de informação (implícitas) são as seguintes:

- Um *byte* pode ser convertido em um *short, int, long, float* ou *double*.
- Um *short* pode ser convertido em um *int, long, float* ou *double*.
- Um *char* pode ser convertido em um *int, long, float* ou *double*.
- Um *int* pode ser convertido em um *long, float* ou *double*.
- Um *long* pode ser convertido em um *float* ou *double*.
- Um *float* pode ser convertido em um *double*.

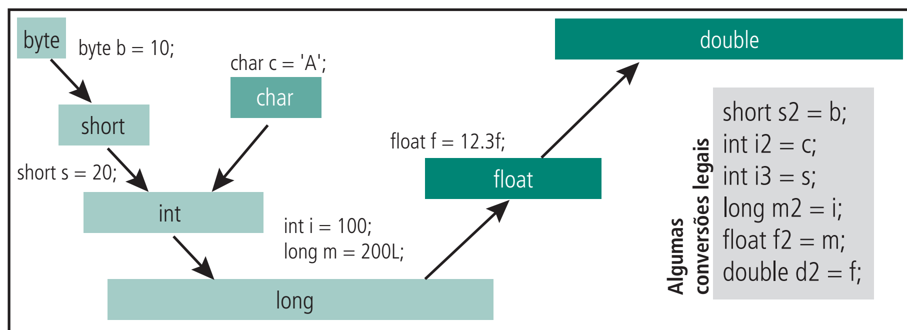



**Figura 2.1: Exemplo de conversões de tipos**
Fonte: Curso Java Starter (www.t2ti.com)

As conversões explícitas (*casting*) são permitidas em todos os tipos (exceto o *boolean*), mas o programador deve estar ciente de que poderá haver perda de informação.

## 2.4 Operadores

### 2.4.1 Operadores aritméticos

Os operadores aritméticos + - * / são utilizados para a adição, subtração, multiplicação e divisão. A divisão retorna resultado inteiro se os operadores forem inteiros, e valores de ponto flutuante em caso contrário. Se for necessário ter o valor do resto da divisão, utilizamos o % (função mod).

É possível utilizar operadores na atribuição das variáveis, como no exemplo:

```
int n = 5;
int a = 2 * n; // a = 10
```

Existe também a possibilidade de utilizar atalhos para operações (também chamados de atribuição composta):

```
x += 4;     // equivalente a x = x + 4;
x /= 10;    // equivalente a x = x/10;
```

A exponenciação é feita por uma função da biblioteca matemática. Essa biblioteca tem dezenas de operações específicas.

```
double y = Math.pow (x, b); // x é elevado a b (xb)
```

Incremento e decremento

O uso de contadores em programas é muito comum. Existem maneiras de realizar incrementos e decrementos em variáveis utilizando o sinal ++ e o --. Veja nos exemplos:

```
int a = 12;
a++ // a agora vale 13
```

O uso do incremento e do decremento depende da posição onde eles se encontram na expressão. Existem ocasiões em que se quer a expressão calculada e que o valor seja incrementado depois. Em outros casos, o valor deve ser incrementado e a expressão avaliada ao final.
Acompanhe o exemplo:

```
int m = 7;
int n = 7;
int a = 2 * ++m; // a vale 16, m vale 8
int b = 2 * n++; // b vale 14, n vale 8
```

### 2.4.2 Operadores relacionais e *booleanos*

Esses operadores servem para avaliar expressões. Para verificar a igualdade entre dois valores, usamos o sinal = = (dois sinais de igual).

O operador usado para verificar a diferença (não igual) é o !=. Temos ainda os sinais de maior (>), menor (<), maior ou igual (>=), menor ou igual (<=).

Exemplos:

```
x = = 10;
3 != 2;
10 < 10
10 > 6;
3 >= 3;
7 <= 6;
```

Existem ainda os operadores lógicos. Os principais são: AND (&&), OR (||) e NOT(!).

Exemplos:

```
(0 < 2) && ( 10 > 5)        resultado: true
( 10 >11 ) || (10 < 12 )    resultado: true
!( 4 = = 4 )                 resultado: false
```

### 2.4.3 Precedência de operadores

A precedência refere-se à ordem na qual os operadores são executados. Operadores de mesma precedência são executados de acordo com sua **associatividade**. Por exemplo, considere a seguinte linha de código :

```
int j = 3 * 10 % 7;
```

Os operadores * e % possuem a mesma precedência e associatividade da esquerda para direita. Assim, o * é executado primeiro, produzindo o resultado **2**. Se fosse colocado um **parênteses** para modificar a precedência desta forma int j = 3 * (10 % 7), o resultado final seria completamente diferente (**9**).

A Tabela 2.1 mostra os operadores Java em ordem de precedência, onde a linha 1 tem a mais alta precedência. Com exceção dos operadores unários, condicionais e o operador de atribuição, os quais são associativos da direita para esquerda, todos os outros operadores com a mesma precedência são executados da esquerda para direita.

**Tabela 2.1: Operadores**

| Operador | Precedência | Comentários | Associatividade |
|---|---|---|---|
| **++, --, !** | **1** | **Operadores unários** | **Direita** |
| *, /, % | 2 | Multiplicação, divisão, resto | Esquerda |
| +, - | 3 | Adição, subtração | Esquerda |
| < > <= >= | 4 | Operadores relacionais | Esquerda |
| = =, != | 5 | Operadores de igualdade | Esquerda |
| && | 6 | AND | Esquerda |
| \|\| | 7 | OR | Esquerda |
| = | 8 | Atribuição | Direita |

## 2.5 *Strings*

Todos os outros valores que utilizamos em Java, com exceção dos tipos explicados acima (ditos primitivos), são objetos. Um dos objetos mais utilizados é o *String* (com S maiúsculo); O *String* é uma sequência de caracteres.

```
String e = “”; // String vazia. Note as aspas duplas.
String oi = “Bom dia”;
```

As *strings* podem ser concatenadas, utilizando o sinal de +, como no exemplo:

```
String um = “Curso”;
String dois = “Java”;
String result = um + dois;
```

Nota: Uma *String* não deve ser comparada com outra usando o sinal ==, pois elas são objetos. Existe um método especial para comparar objetos, utilizando o *equals*. Assim, a comparação da *String* a com uma *String* b seria:

```
a.equals(b);
```

## 2.6 Constantes

Uma constante pode ser definida tanto como um atributo de classe como uma variável local. Uma vez que foi declarado e atribuído um valor para ela, não é permitido que outra atribuição seja efetuada, ocasionando um erro de compilação caso isso ocorra. A atribuição do valor inicial pode ser feita no momento da declaração, ou posteriormente em outra parte do programa. Para declarar uma variável do tipo constante, devemos utilizar o modificador **final** desta forma:

```
final <tipo> <identificador> [= valor];
```

Declare constantes com nomes descritivos, escritos com todas as letras maiúsculas. Separe nomes internos com *underscores* ( _ ). Exemplos:

```
final double MIN_VALOR = 100.0;
final long NUMERO_MAXIMO_DE_VEZES;
```

## 2.7 Controle de fluxo

Nesta seção iremos abordar os comandos que nos permitem controlar o fluxo do programa e expressões condicionais em Java. Mas, antes, temos que aprender a delimitar blocos e conceituar o escopo.

Um bloco nada mais é do que uma série de linhas de código situadas entre um abre e fecha chaves ( { } ). Podemos criar blocos dentro de blocos. Dentro de um bloco temos um determinado escopo, que determina a visibilidade e tempo de vida de variáveis e nomes. Por exemplo:

```
1  {
2  int x = 10;
3  // aqui eu tenho acesso ao x
4    {
5        int z = 20;
6        // aqui eu tenho acesso ao x e ao z
7    }
8  // aqui eu tenho acesso ao x; o z esta fora do escopo
9  }
```

**Figura 2.2:Visibilidade e tempo de vida de uma variável**
Fonte: Elaborada pelos autores

Assim, é permitida a definição de variáveis com mesmo nome, desde que elas não estejam compartilhando o mesmo escopo. A definição dos blocos ajuda a tornar o programa mais legível e a utilizar menos memória, além de indicar quais os comandos a serem executados pelas instruções condicionais e os *loop*, que veremos mais adiante.

Expressão *booleana* formada pelos operadores lógicos e relacionais. Se a expressão *boolena* for verdadeira (*true*), executa comando1 ou bloco1; caso seja falsa (*false*), executa comando2 ou bloco2.

## 2.7.1 O comando *if – else*

**Sintaxe: if (expressão)**

```
 comando1 ou { bloco1 }
else // opcional
 comando2 ou { bloco2 } // opcional
```

Exemplo:

```
if  ( fim = = true )
   System.out.println("Término!");
else
   System.out.println("Continuando...");
```

Para mais de um comando ser executado depois da declaração, utiliza-se o conceito de blocos, delimitados por **{ }**. Você pode concatenar expressões *booleanas* através dos operadores lógicos "E" (&&) e "OU" (||).

Exemplo:

```
1  if  ( fim = = false ) && (cont > 0){
2       cont ++;
3       System.out.println("Continuando...");
4  }
5  else {
6       cont = 0;
7       System.out.println("Término!");
8  }
```

**Figura 2.3: Concatenação de expressões *booleanas***
Fonte: Elaborada pelos autores

É recomendado **sempre** usar blocos de comandos quando houver mais de um comando, para facilitar a leitura e compreensão do código. É possível também criar comandos *if* aninhados.

Exemplo :

```
1  if ( fim = = false) && (cont < 0){
2      cont++;
3  }
4  else if (fim = = false) && (cont > 0){
5          cont--;
6        }
7        else {
8            cont = 0;
9            System.out.println("Término!!");
10       }
```

**Figura 2.4: *If* aninhados**
Fonte: Elaborada pelos autores

## 2.7.2 O comando *switch*

Utilizamos o comando *switch* para avaliar uma expressão contra vários resultados possíveis. Aceita **somente** um *char*, *byte*, *short* ou *int* como sua expressão de comparação. O valor é procurado nas declarações **case** que vêm depois da declaração *switch* e o código adequado é executado. O *break* no final de cada comando serve para evitar comparações inúteis depois de encontrado o valor correto. Se for preciso mais de um comando, é necessário colocar o bloco das instruções entre { }.

A variável deve ser um literal ou uma constante.

Quando encontra uma opção válida, os comandos são executados até um *break* ser encontrado, o qual transfere o controle para o comando seguinte ao comando *switch*. Ou seja, se um *break* não é especificado ao término de um *case*, todas as diretivas dos *cases* subsequentes serão executadas até que um *break* seja encontrado ou o comando *switch* termine.

Sintaxe:

**switch (variável)**
**{**
**case (valor1): comando ou { bloco } break;**
**case (valor2): comando2 ou { bloco2 } break;**

**default: comando_final ou { bloco final }**
**}**

Exemplo:

```
1  switch ( mes ){
2  case 1: System.out.println("Janeiro");
3          break;
4  case 2: System.out.println("Fevereiro");
5          break;
6  case 3: System.out.println("Março");
7          break;
8  case 4: System.out.println("Abril");
9          break;
10 case 5: System.out.println("Maio");
11         break;
12 case 6: System.out.println("Junho");
13         break;
14 case 7: System.out.println("Julho");
15         break;
16 case 8: System.out.println("Agosto");
17         break;
18 case 9: System.out.println("Setembro");
19         break;
20 case 10: System.out.println("Outubro");
21         break;
22 case 11: System.out.println("Novembro");
23         break;
24 case 12: System.out.println("Dezembro");
25         break;
26 default: System.out.println("Mes invalido!");
27 }
```

**Figura 2.5: Exemplo de comando *switch***
Fonte: Elaborada pelos autores

## 2.7.3 O comando *while*

Implementa um *loop* para executar um bloco de comandos sucessivas vezes. A expressão de comparação é avaliada antes que o laço seja executado. O objetivo é que esse bloco de comandos seja repetido enquanto a expressão de comparação permanecer verdadeira. É importante lembrar que o bloco de comandos deve estar entre chaves ({}).

Sintaxe :

```
while ( expressão_booleana ) {
  comandos;
}
```

Enquanto a expressão for verdadeira, o laço será executado

Exemplo:

```
int i = 0;
while(i < 10) {
  System.out.println(i);
  i = i + 1;
}
```

## 2.7.4 O comando *do-while*

Utilizado quando se quer que o corpo do laço seja necessariamente executado pelo menos uma vez. A expressão de comparação é avaliada depois que o laço for executado. A única diferença entre o *while* e o *do-while* é que no segundo, o comando é sempre executado pelo menos uma vez.

Sintaxe:

```
do {
  comandos;
} while ( expressao_booleana );
```

Exemplo:

```
int i = 1;
do {
  System.out.println(i);
  i = i + 1;
} while(i >= 10);
```

## 2.7.5 O comando *for*

Outro comando de *loop* extremamente utilizado é o ***for***. A ideia é a mesma do ***while***: fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas, além disso, o *for* isola também um espaço para

inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis as variáveis que são relacionadas ao *loop*. Comumente, um *loop* ***for*** possui uma parte inicial com a inicialização das variáveis, seguida por uma expressão de comparação e depois a parte final com o incremento ou decremento das variáveis do laço.

Sintaxe:

```
for ( inicialização; condição; iteração ) {
comandos;
}
```

Neste caso, como só temos um comando, não foi necessário o uso das chaves ({}).

Exemplo:

```
int i;
for (i = 0; i < 10; i ++)
System.out.println(i);
```

Pode-se declarar variáveis na inicialização do *for*.

Exemplo:

```
for (int i = 0 ; i < 10 ; i++)
  comando;
```

O único elemento realmente imprescindível é a condição. É possível construir laços *for* sem a inicialização e/ou o incremento. Ex.:

```
int i = 2;
for ( ; i<10 ; ){
  comandos;
  i++;
}
```

A inicialização e incremento podem ser feitos com mais de uma variável. Nesse caso devem estar separados por vírgula. Ex.:

```
for ( i = 0, j = 10 ; i < 8 && j > 2 ; i ++, j-- )
  comando;
```

## 2.7.6 Controlando *loops*

Apesar de termos condições booleanas nos nossos laços, em algum momento podemos decidir parar o *loop* por algum motivo especial, sem que o resto do laço seja executado.

O comando **break** é utilizado para transferir o controle para o final de uma construção de laço (*for, do-while, while*) ou de um *switch*. O laço vai encerrar independentemente de seu valor de comparação, e o comando após ele será executado. Exemplo:

O código acima irá imprimir os números sequenciais começando do numero 0 (zero) e irá parar quando a variável i for maior que 10, ou seja, quando a variável i atingir o valor 10.

O código acima vai percorrer os números de x a y e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra-chave *break*.

```
1  int i = 0;
2  while (true) {
3      System.out.println(i);
4      i++;
5  if ( i > 10 )
6      break;
7  }
```

**Figura 2.6: Comando *break***
Fonte: Elaborada pelos autores

```
1  for (int i = x; i < y; i++) {
2      if (i % 19 == 0) {
3          System.out.println("Achei um número divisível
4                              por 19 entre x e y");
5          break;
6      }
7  }
```

**Figura 2.7: Exemplo  de comando *for***
Fonte: Elaborada pelos autores

O comando ***continue*** interrompe a execução de um *loop* naquele ponto sem executar o resto dos comandos, e volta para o início do *loop*para uma nova iteração. Exemplo:

```
1  for (int i = 0; i < 100; i++) {
2    if(i > 50 && i < 60) {
3      continue;
4    }
5  System.out.println(i);
6  }
```

**Figura 2.8: O comando *continue***
Fonte: Elaborada pelos autores

Este código irá inicialmente imprimir todos os números de 0 a 99, porém existe uma condição dentro do laço que impede que os números compreendidos entre 50 e 60 sejam impressos.

## 2.8 Escopo de variáveis

O escopo da variável é o nome dado ao trecho de código em que aquela variável existe e que é possível acessá-la.

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro. Ex.:

```
//aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro só valem até o fim daquele bloco. Ex.:

```
1  //aqui a variável i não existe
2  int i = 5;
3  // a partir daqui ela existe
4  while (condicao) {
5  // o i ainda vale aqui
6      int j = 7;
7  // o j passa a existir
8  }
9  // aqui o j não existe mais, mas o i continua a valer
```

**Figura 2.9: Validade de variáveis**
Fonte: Elaborada pelos autores

No bloco acima, a variável j para de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

O mesmo vale para um *if*:

```
1  if (algumBooleano) {
2      int i = 5;
3  }
4  else {
5      int i = 10;
6  }
7  System.out.println(i); // cuidado!
```

**Figura 2.10: Acesso à variável fora do seu escopo**
Fonte: Elaborada pelos autores

Aqui a variável i não existe fora do *if* e do *else*! Se você declarar a variável antes do *if*, vai haver outro erro de compilação: dentro do *if* e do *else* a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
1  int i;
2  if (algumBooleano) {
3      i = 5;
4  }
5  else {
6      i = 10;
7  }
8  System.out.println(i);
```

**Figura 2.11: Uso de variáveis dentro de *if***
Fonte: Elaborada pelos autores

Uma situação parecida pode ocorrer com o *for*:

```
for (int i = 0; i < 10; i++) {
  System.out.println("olá!");
}
System.out.println(i); // cuidado!
```

Nesse ***for*** a variável i morre ao seu término, não podendo ser acessada de fora dele, gerando um erro de compilação. Se você realmente quer acessar o contador depois do *loop* terminar, precisa de algo como:

```
int i;
for (i = 0; i < 10; i++) {
   System.out.println("olá!");
}
System.out.println(i);
```

## Resumo

Nesta aula fizemos uma abordagem da linguagem Java e exploramos suas principais características. Vimos os tipos primitivos e também as formas de conversão (*casting*) implícita e explícita. Os operadores em Java estão classificados em: Relacionais, Aritméticos e Lógicos. A precedência de operadores deve ser lavada em consideração na hora de resolver uma expressão. Abordamos os comandos que nos permitem controlar o fluxo de um programa em Java.

Os comandos vistos foram: ***if-else<***, ***switch***, ***while***, ***do-while*** e ***for***. Os comandos *if-else* e *switch* são condicionais (decisão), enquanto os comandos: *while*, *do-while* e *for* são de repetição (controle de laços). Foram explicados os objetivos de cada comando bem como sua sintaxe e suas diferenças. Por último, explicamos um conceito de fundamental importância para qualquer programador, o escopo de variável, que é o bloco de código dentro do qual ela é acessível e determina quando a variável é criada e destruída.

## Atividades de aprendizagem

**1.** Analise as seguintes declarações de variáveis e assinale (V) para Verdadeiro e (F) para Falso

**a)** ( ) byte x = 300;

**b)** ( ) short $mom = 43;

**c)** ( ) int num = 2.5;

**d)** ( ) double d = 20;

**e)** ( ) char c = 25;

**f)** ( ) boolean epar = (4%2=0)

**g)** ( ) char nome = “Maria”;

**h)** ( ) String b = “1”

**i)** ( ) float x = 2.7;

**j)** ( ) int a = false;

**2.** Escreva um programa que declare, inicialize e imprima as seguintes variáveis:

- Inteiro i de 32 *bits* com valor 1
- Inteiro j de 64 *bits* com valor 2
- Ponto flutuante p de 32 *bits* com valor 20.0
- Ponto flutuante q de 64 *bits* com valor 30.0
- *Boolean* b com valor verdadeiro
- Caracter c com valor 'k'

**3.** Crie um programa seguindo as orientações abaixo. Faça tudo que pede dentro do método ***main***.

**a)** Declare duas variáveis para armazenar os valores de dois itens de venda. Os valores dos dois itens devem ser 2.95 e 3.50. Pense em nomes significativos para as variáveis e também considere qual deve ser seu tipo.

**b)** Use o System.out.println() para mostrar o conteúdo de suas variáveis. Execute o programa e veja a saída. Apresente uma mensagem significativa como "O Item 1 custa 2.95 e ..." (Dica : utilize o operador + para concatenar o texto com o valor da variável).

**c)** Declare uma outra variável para armazenar o custo total dos itens. Utilize o operador de adição para realizar o cálculo e imprima o resultado.

**d)** Crie uma constante para armazenar a taxa de 8.25 % que deve ser cobrada sobre o valor total. Armazene o cálculo numa variável chamada taxaCalculada e imprima o resultado.

**e)** Adicione ao valor de cada item o valor da taxa calculado. Use a menor linha de código possível para fazer isso. Some novamente os dois valores e atribua o resultado a uma variável chamada novoCusto.

**f)** Crie uma variável *booleana* chamada muitoCaro. Escreva uma assertiva lógica para setar essa variável para *true* se novoCusto for maior que 10 e para *false*, caso contrário. Imprima o valor obtido em muitoCaro. (Não utilize *if* para escrever a assertiva.)

4. Declare uma variável chamada long1 do tipo *long* e a inicialize com 100. Agora declare duas variáveis do tipo int, int1 e int2, e inicialize int1 com 200. Agora faça int2 receber int1 mais long1 e explique os resultados. Arrume o código para que não ocorra o erro anterior.

5. Implemente um programa que recebe um número de 1 a 7 e imprime o dia da semana correspondente.

6. Altere o programa do exercício anterior para ficar recebendo o número dentro de um laço enquanto for diferente de 0 (zero).

7. Escreva um programa que, dada uma variável x (com valor 180, por exemplo), temos y de acordo com a seguinte regra:

   se x é par, y = x / 2
   se x é impar, y = 3 * x + 1
   imprime y
   O programa deve então jogar o valor de y em x e continuar até que y tenha o valor final de 1.
   Por exemplo, para x = 13, a saída será:
   40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

8. Escreva um programa que imprima todos os múltiplos de 3, entre 1 e 100.

9. Escreva um programa que implemente um laço *while* que execute 20 vezes, imprimindo o valor da variável x que inicialmente está com valor 10.

10. Reescreva o programa anterior utilizando o comando *do-while*.

Poste suas soluções no AVEA.

# Aula 3 – Orientação a objetos: conceitos básicos

## Objetivos

Compreender o que é e para que serve orientação a objetos.

Criar modelos para representação de informações em um determinado contexto.

Conceituar objetos, classes, atributos e comportamentos.

Entender o significado da criação e uso de variáveis de referência e objetos na memória.

Implementar o encapsulamento através modificadores de acesso nas definições de classe.

Implementar classes executáveis através do método *main* para manipulação de classes através de objetos criados.

## 3.1 Introdução

Iniciaremos nossa aula perguntando: Por que Orientação a Objetos? Qual a vantagem da "Programação Orientada à Objetos" em relação à "Programação Estruturada"?

Um paradigma de programação, seja ele estruturado ou orientado a objetos é a forma como a solução para um determinado problema é desenvolvida. Por exemplo, em Orientação a Objetos os problemas são resolvidos pensando-se em interações entre diferentes objetos; já no paradigma Estruturado, procura-se resolver os problemas decompondo-os em funções e dados que somados formarão um programa.

Durante muito tempo, a programação Estruturada foi o paradigma mais difundido; porém, à medida que os programas foram tornando-se mais complexos, surgiu a necessidade de resolver os problemas de uma maneira diferente. Nesse contexto surge o paradigma da Programação Orientada a

Objetos. Nesse paradigma os dados e as operações que serão realizadas sobre estes formam um conjunto único (objeto), e a resolução de um problema é dada em termos de interações realizadas entre esses objetos. Dizemos que um objeto encapsula a lógica de negócios concentrando a responsabilidade em pontos únicos do sistema, viabilizando um maior reuso do código pela modularidade dessa abordagem.

Benefícios da abordagem orientada a objetos:

- modularidade: uma vez criado, um objeto pode ser passado por todo o sistema;

- encapsulamento: detalhes de implementação ficam ocultos externamente ao objeto;

- reuso: uma vez criado, um objeto pode ser utilizado em outros programas;

- manutenibilidade: manutenção é realizada em pontos específicos do programa (objetos).

## 3.2 Modelos

Modelos são representações simplificadas de objetos, pessoas, itens, tarefas, processos, conceitos, ideias, etc., usados comumente por pessoas no seu dia a dia, independentemente do uso de computadores.

Um modelo comumente possui dados (informações) e operações (procedimentos) associados a ele.

Para exemplificar o uso de modelos, consideremos uma lâmpada incandescente comum e um modelo para representá-la. Essa lâmpada tem um dado básico, que é seu estado ("ligada" ou "desligada"). As operações que podemos efetuar nessa lâmpada também são simples: podemos ligá-la ou desligá-la. O ato de ligar a lâmpada equivale a modificar seu estado para "ligada", enquanto que desligar a lâmpada significa modificar seu estado para "desligada". Para saber se uma lâmpada está ligada ou desligada, podemos pedir que uma operação mostre o valor do estado. Nota-se neste exemplo que a modelagem de entidades do mundo real pode ser muito diferente da que vai ser usada em programas de computador: no mundo real,

para saber se a lâmpada está ligada ou não, basta observá-la; enquanto que na modelagem para uso em um computador, precisamos de uma operação para saber se ela está ou não ligada. Vale ressaltar que os dados contidos no modelo são somente os relevantes à abstração do mundo real que está sendo feita. Por exemplo, alguns dados como o preço da lâmpada são irrelevantes e não devem ser representadas pelo modelo em questão. A Figura 3.1 mostra o modelo Lampada usando uma variante do diagrama de classes da Linguagem Unificada de Modelagem (*Unified Modeling Language*, UML). Nesse diagrama, o retângulo superior mostra o nome do modelo, o retângulo central mostra dados do modelo a ser manipulados e o retângulo inferior mostra que operações do modelo podem ser usadas. Os nomes das operações ou métodos são seguidos de parênteses, e quando argumentos ou parâmetros devem ser especificados para as operações, eles são listados dentro dos parênteses.

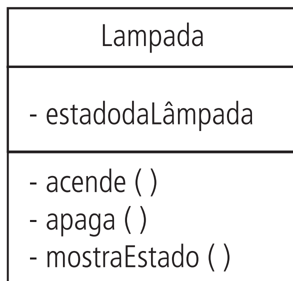



**Figura 3.1: Representação do modelo Lampada, seus dados e operações**
Fonte: Elaborada pelos autores

A listagem abaixo mostra como o modelo **Lampada** poderia ser descrito em pseudocódigo.

```
 1  modelo Lampada          // representa uma lâmpada em uso
 2  início do modelo
 3      dado estadoDaLâmpada; // indica se está ligada ou não
 4    operação acende()      // acende a lâmpada
 5      início
 6         estadoDaLâmpada = aceso;
 7      fim
 8    operação apaga()       // apaga a lâmpada
 9     início
10      estadoDaLâmpada = apagado;
11     fim
12    operação mostraEstado() // mostra o estado da lâmpada
13     início
14       se (estadoDaLâmpada == aceso)
15          imprime "A lâmpada está acesa";
16       senão
17          imprime "A lâmpada está apagada";
18      fim
19   fim do modelo
```

**Figura 3.2: Modelo Lampada escrito em pseudocódigo**
Fonte: Elaborada pelos autores

Outro exemplo: Um modelo de uma conta bancária simplificada.

O modelo deste exemplo representa uma conta bancária simplificada, que somente representa o nome do correntista, o saldo da conta e se a conta é especial ou não. Se a conta for especial, o correntista terá o direito de retirar mais dinheiro do que tem no saldo (ficar com o saldo negativo). Aspectos práticos encontrados em contas de banco reais (senhas, taxas, impostos, etc.) foram deixados de lado em nome da simplicidade. A Figura 3.3 mostra os dados e operações deste modelo, e a listagem mostra o modelo descrito em pseudocódigo.

| ContaBancariaSimplificada |
|---|
| - nomeDoCorrentista<br>- saldo<br>- contaÉEspecial |
| - abreConta (nome, depósito, éEspecial<br>- abreContaSimples (nome)<br>- deposita (valor)<br>- retira (valor)<br>- mostraDados ( ) |

**Figura 3.3: Representação do modelo ContaBancariaSimplificada, seus dados e operações**
Fonte: Elaborada pelos autores

A Figura 3.4 mostra como o modelo **ContaBancariaSimplificada**poderia ser descrito em pseudocódigo.

```
1   modelo ContaBancariaSimplificada
2    início do modelo
3    dado nomeDoCorrentista,saldo,contaÉEspecial; // dados da conta
4      // Inicializa simultaneamente todos os dados do modelo
5    operação abreConta(nome,depósito,especial)
6    início
7       nomeDoCorrentista = nome;
8       saldo = depósito;
9       contaÉEspecial = especial;
10   fim
11  operação abreContaSimples(nome) // argumento para esta operação
12   início
13       nomeDoCorrentista = nome;
14       saldo = 0.00;
15       contaÉEspecial = falso;
16   fim
17   operação deposita(valor) // Deposita um valor na conta
18   início
19      saldo = saldo + valor;
20   fim
21
22  operação retira(valor)
23   início
24      se (contaÉEspecial == falso) // A conta não é especial !
25      início
26        se (valor <= saldo) // se existe saldo suficiente...
27           saldo = saldo - valor; // faz a retirada.
28      fim
29      senão // A conta é especial, pode retirar à vontade !
30        saldo = saldo - valor;
31   fim
```

**Figura 3.4: Conta bancária simplificada escrita em pseudocódigo**
Fonte: Elaborada pelos autores

```
1   operação mostraDados() // mostra os dados da conta
2   início
3      imprime "O nome do correntista é ";
4      imprime nomeDoCorrentista;
5      imprime "O saldo é ";
6      imprime saldo;
7      se (contaÉEspecial)
8         imprime "A conta é especial.";
9      senão
10        imprime "A conta é comum.";
11  fim
12  fim do modelo
```

**Figura 3.5: Operação mostraDados em pseudocódigo**
Elaborada pelos autores

Alguns pontos interessantes da listagem acima são:

- Existem duas operações que podem ser usadas para a abertura de contas: uma para a qual temos que passar todos os dados que serão usados (abreConta) e outra para a qual somente precisaremos passar o nome do correntista (abreContaSimples). É esperado que somente uma dessas operações seja usada por quem for utilizar o modelo, mas é comum oferecer várias opções .

- O conceito de que operações manipulam os dados do modelo fica mais claro ainda com as operações **deposita** e **retira**, que são usadas para modificar o saldo da conta.

- O uso de blocos é demonstrado de novo na operação **retira**, na qual um bloco delimita comandos que só serão executados se a conta não for especial.

Também é possível a criação de modelos que contenham somente dados ou somente operações. Os primeiros são pouco usados. Quando criamos modelos para representação de dados é interessante e útil adicionar algumas operações para manipulação desses dados. Modelos que contenham somente operações podem ser considerados bibliotecas de operações. São exemplos desses modelos os grupos de funções matemáticas e de processamento de dados, que não precisem ser armazenados. A criação e o uso de bibliotecas de operações em Java serão vistos mais adiante.

Modelos podem conter submodelos e ser parte de outros modelos. Por exemplo, um modelo de casa poderia ser composto por diversos modelos de lâmpada. Um modelo de lâmpada pode fazer parte tanto de um modelo de casa quanto de um modelo de carro, etc.

A simplificação inerente aos modelos é, em muitos casos, necessária: dependendo do contexto, algumas informações devem ser ocultas ou ignoradas. Por exemplo, a representação das informações sobre uma pessoa pode ser feita de maneira diferente, dependendo do contexto, como nos três exemplos mostrados abaixo:

- **Pessoa como Empregado de Empresa:** Para uma pessoa como empregada de uma empresa, para fins de processamento de folha de pagamento, seria necessária a representação do *nome*, *cargo*, *salário* e *horasExtrasTrabalhadas*, entre outros dados. Esse modelo poderia conter as operações *calculaSalario* e *aumentaSalario*.

- **Pessoa como Paciente de uma Clínica Médica:** Para um modelo de paciente seria necessário representar o *nome*, o *sexo*, a *idade*, a *altura*, o *peso* e o *historicoDeConsultas* do paciente. Esse modelo poderia conter as operações *verificaObesidade*, que usaria os dados *sexo*, *altura* e *peso* para verificar se aquela pessoa está com o peso ideal para sua categoria e *adicionaInformacaoAoHistorico*, que adicionaria informações recentes ao histórico do paciente.

- **Pessoa como Contato Comercial:** Para esse modelo, seria necessário representar o *nome*, o *telefone*, o *cargo* e *empresa* da pessoa. Algumas operações úteis para esse modelo seriam *mostraTelefone*, que retornaria o telefone de uma certa pessoa, e *trabalhaEmEmpresa*, que informaria se a pessoa trabalha em uma certa empresa.

As três maneiras de representarmos os dados de uma pessoa e operações nesses dados (ou os três modelos de representação de pessoas) são dependentes de contexto: alguns dados e operações que podem ser úteis para um modelo de representação de pessoas podem ser irrelevantes para outro – por exemplo, não faria sentido representar o *salário* de uma pessoa para fins de registros no banco de dados de pacientes de uma clínica ou a operação *verificaObesidade* de uma pessoa que não seja paciente de uma clínica médica. Por essa razão, e apesar dos três exemplos acima representarem pessoas de forma genérica, é difícil, se não impossível, elaborar um "supermodelo" capaz de representar todos os dados e operações relativos a uma pessoa, independentemente do contexto, e usar esse modelo no lugar de todos os outros mais especializados ou dependentes de contexto.

Modelos podem ser reutilizados para representar diferentes objetos, pessoas ou itens: o mesmo modelo *PacienteDeClinica* poderia ser utilizado para representar cada um dos pacientes de uma clínica – os pacientes podem ser representados pelo mesmo modelo, mas os dados individuais de cada um podem ser diferentes, conforme a necessidade dos diversos exemplos de cada modelo – assim, *João*, *Pedro* e *Maria* seriam exemplos do modelo *Paciente*, cada um contendo dados de um paciente, provavelmente diferentes entre si. Vemos que não é necessária a criação de um modelo para cada item, pessoa ou objeto do mundo real.

A criação e uso de modelos é uma tarefa natural, e a extensão dessa abordagem à programação deu origem ao paradigma Orientação a Objetos.

## 3.3 Objetos

A resolução de um problema passa pela análise de uma determinada situação real, tendo-se por objetivo a construção de um modelo que represente essa situação. Isso caracteriza o processo de abstração. Esse modelo deverá considerar os objetos (entidades) que integram o problema. Mas o que seria mesmo um objeto?

**Objetos** são um conjunto de dados e procedimentos que representam a estrutura e o comportamento inerentes às entidades, concretas ou abstratas, do mundo real. Em outras palavras, são qualquer coisa que tenha algum significado dentro do contexto do problema, seja ela concreta ou abstrata. Alguns exemplos de objetos:

- Um livro.
- Uma data.
- Uma festa.
- Um carro.
- Um pedido de compra de material.

Olhando à sua volta é possível perceber muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta. Esses objetos têm duas características básicas, que são o **estado** e o **comportamento**. Alguns exemplos:

| Objeto | Estado | Comportamento |
|---|---|---|
| Pessoa | nome, idade, RG | falar, andar, sorrir, chorar |
| Cachorro | nome, raça | latir, correr |
| Carro | cor, marca, modelo | acelerar, frear, ligar |
| Televisão | marca, tamanho, tipo | ligar, sintonizar, desligar |

Cada objeto consiste em uma entidade com identidade própria. Assim, mesmo que a partir de uma observação simples se diga que dois objetos são iguais, tem-se dois objetos distintos. Se analisarmos dois exemplares de um livro, por exemplo, veremos que se constituem em dois objetos distintos, apesar de terem a mesma quantidade de páginas, mesmo conteúdo, etc.

Em termos de programação, podemos definir um objeto como a abstração de uma entidade do mundo real, que apresenta sua própria existência, identificação, características de composição e que tem alguma utilidade, isto é, pode executar determinados serviços quando solicitado.

Todo objeto deve ter uma identificação. A identificação de um objeto refere-se ao nome que apresenta, o qual será definido pelo projetista do modelo. As características de composição de um objeto estão associadas à sua constituição e definem a sua estrutura. Essas características são denominadas Atributos. Assim, por exemplo, um objeto que seja a abstração de um celular, digamos o objeto identificado pelo nome meuCelular, pode ter como atributos, dependendo da abstração feita, a sua cor, a sua marca, o seu peso, seu comprimento, seu preço, a bateria etc.

Os atributos de um objeto definem a composição deste e podem ser modelados, dependendo do contexto, como simples valores ou como outros objetos. Pode-se tratar o atributo de um objeto como um simples valor quando este constitui-se em uma informação que pode ser expressa por um único valor. No caso do objeto umCelular, o atributo que define o comprimento do Celular pode ser expresso por um simples valor, por exemplo, 8.0. Já o atributo que representa a bateria, dadas as suas características, deve ser modelado como outro objeto. Os atributos de um objeto definem a sua estrutura e podem ser modificados durante a vida útil do objeto. Assim, por exemplo, no caso do objeto umCelular, o valor do atributo que especifica o preço muda durante a sua vida útil.

Além das características de composição, ou seja, os atributos, um objeto apresenta também as características de utilidade. Essas características dizem respeito às ações ou serviços que o objeto pode executar. Denominamos Comportamento o conjunto de ações ou serviços que um objeto pode prestar. Conforme ocorre no mundo real, para que um objeto execute determinado serviço, é necessário que haja uma solicitação. A denominação de comportamento para o conjunto de serviços que o objeto pode executar se justifica, uma vez que que esses serviços definem a forma como ele reage às solicitações.

Um objeto é uma entidade que tem uma identificação e que apresenta atributos e/ou comportamento. No caso do objeto umCelular, a abstração pode considerar um comportamento formado por serviços, tais como ligar, desligar, fazer uma chamada,  etc. Veja a seguir alguns exemplos de objetos com suas respectivas abstrações. Lembre-se que uma abstração é dependente de quem a executa, ou seja, é dependente do problema que se quer resolver.

Em termos de programação, um objeto é a abstração de uma entidade presente no problema a ser resolvido.

**Exemplo 1**

Seja o objeto resultante de uma abstração da porta de determinada sala, digamos o objeto *umaPorta*. Pode-se nessa abstração identificar, por exemplo, os seguintes atributos e serviços:

Atributos (Estado):

Cor. (pode ser expresso por um valor);

Altura. (pode ser expresso por um valor);

Largura (pode ser expresso por um valor);

A fechadura (expresso como outro objeto).

Serviços (Comportamento)

Abrir; Fechar.

Veja que nesse exemplo os atributos cor, altura e largura podem ser expressos como valores. Assim, o objeto identificado por *umaPorta* pode ter, por exemplo, o valor 2.1 para o atributo altura ou o valor "cinza" para o atributo cor. Já o atributo que representa a fechadura da porta constitui-se em outro objeto que, obviamente, apresenta características diferentes de uma porta.

**Exemplo 2**

Considere a abstração de um relógio, digamos o objeto *meuRelogio*. Fazendo-se uma abstração do ponto de vista do usuário do relógio, podemos definir, por exemplo, os seguintes atributos e serviços:

Atributos(Estado)

Peso (30 gramas, por exemplo)

Custo (R$ 200,00 por exemplo)

A Pulseira (outro objeto)

Os Ponteiros (outro objeto)

A Pilha (outro objeto)

Serviços (Comportamento)

Iniciar marcação de horas (funcionar);

Informar horas;

Informar data;

Corrigir horas;

Trocar pilha.

Na definição do comportamento de um objeto estaremos identificando quais os serviços que ele pode executar. Nesse processo, devemos imaginar o objeto como algo que tenha vida e que, portanto, tenha a capacidade de realização de serviços, mesmo quando esse objeto tratar-se da abstração de algo inanimado ou algo abstrato. Veja que, no caso do objeto *meuRelogio*, a ação trocar pilha não se constitui em objetivo específico de um relógio; entretanto, necessita ser realizada em algum momento e, assim, consideramos em nossa abstração que o objeto tem a capacidade de realizar esse serviço.

**Exemplo 3**

Considere a abstração de determinado aluno de uma disciplina. Digamos que esse objeto seja identificado por *umAluno*. Considerando que o sistema de avaliação na disciplina seja constituído por uma prova escrita, poderíamos identificar os seguintes atributos e serviços:

**Atributos (Estado)**

Nome do aluno;

Número de matrícula;

Nota da prova escrita;

**Serviços (Comportamento)**

Estudar;

Assistir a aula;

Realizar prova.

## 3.4 Classes

Se dois ou mais objetos apresentam as mesmas características, diz-se que são pertencentes a uma mesma classe. O relógio do José e o relógio da Maria são dois objetos distintos, mas ambos apresentam os mesmos atributos e podem executar os mesmos serviços. Isso implica que ambos pertencem à mesma classe, por exemplo, à classe Relógio. Quando dizemos que dois ou mais objetos apresentam os mesmos atributos, estamos nos referindo à existência do atributo e não ao seu conteúdo ou valor. Assim, o relógio do José tem o atributo peso, cujo valor pode ser, por exemplo, 30 gramas. O relógio da Maria também tem o atributo peso, mas seu valor pode ser diferente, por exemplo, 20 gramas. Da mesma forma, o relógio do José tem como atributo a sua pulseira, o qual representa a pulseira desse relógio, que em nossa modelagem exemplo consideramos como outro objeto. Também o relógio da Maria tem como atributo a sua pulseira, a qual é um objeto distinto da pulseira do relógio do José.

Uma classe define as características de um grupo de objetos, isto é, define como serão as instâncias pertencentes a ela. Uma instância é um objeto gerado por uma classe. A classe é apenas a representação de um objeto, enquanto a instância de uma classe é o objeto propriamente dito, com tempo e espaço de existência.

Assim, uma classe especifica quais serão os atributos de todo objeto pertencente a ela bem como quais serviços esse objeto poderá executar. Um serviço também pode ser denominado de Operação. Quando pensamos em uma classe, estamos pensando em como serão os seus objetos. É importante que não se confunda classe com objeto. O objeto pode ser visualizado como o caso "concreto" (ocorrência), enquanto a classe pode ser visualizada como a "ideia" que se tem do objeto.

Quando realizamos uma abstração tendo por objetivo a definição do modelo para a resolução de um problema, procuramos identificar quais objetos integram o problema e como eles interagem no sentido da sua resolução. A identificação, juntamente com a abstração desses objetos, implica definir como estes serão, isto é, que atributos terão tais objetos e que serviços eles poderão executar. Assim, ao identificarmos objetos, estaremos pensando em suas respectivas classes. Veja que será a classe que definirá como serão seus objetos. Como todo objeto é uma instância de alguma classe, isto é, todo objeto pertence a uma classe, a classe pode ser considerada como o tipo do objeto.

Uma classe consiste em uma entidade que descreve quais atributos terão todos os seus objetos bem como quais serviços estes poderão executar. Em termos de programação, quando dizemos que um objeto executa um serviço, significa dizer que o computador executa determinada sequência de instruções. À sequência de instruções que devem ser executadas pelo computador, representando a execução do serviço pelo objeto, denominaremos Método. Uma classe, portanto, é uma descrição dos atributos e dos métodos de determinado tipo de objeto. Em termos de programação, definir classes significa formalizar um novo tipo de dado e todas as operações associadas a esse tipo, enquanto declarar objetos significa criar variáveis do tipo definido.

Considere um programa para um banco: é bem fácil perceber que uma entidade extremamente importante para esse sistema é a *conta*. Nossa ideia aqui é generalizar alguma informação, juntamente com funcionalidades que toda conta deve ter.

O que toda conta tem e é importante para nós?

- número da conta;
- nome do cliente;

- saldo;
- limite.

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir à conta".

- saca uma quantidade x;
- deposita uma quantidade x;
- imprime o nome do dono da conta;
- devolve o saldo atual;
- transfere uma quantidade x para uma outra conta y;
- devolve o tipo de conta.

Com isso, temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o projeto. Antes, precisamos construir uma *conta*, para poder acessar o que ela tem, e pedir a ela que faça alguma coisa.

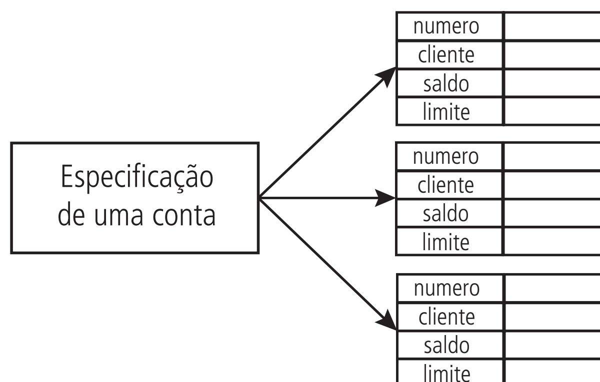



**Figura 3.6: Abstração da classe Conta e seus objetos**
Fonte: Elaborada pelos autores

Repare na Figura 3.6: apesar do papel do lado esquerdo especificar uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, nas quais possamos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como à esquerda na figura), é nas instâncias desse projeto que realmente há espaço para armazenar esses valores. Ao projeto da *conta*, isto é, à definição da *conta*, damos o nome de **classe**. Ao que podemos construir a partir desse projeto, às contas de verdade, damos o nome de **objetos**.

Um outro exemplo: uma receita de bolo. A pergunta é certeira: você come uma receita de bolo? Não. Precisamos instanciá-la, criar um objeto bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são objetos diferentes.

Podemos fazer milhares de analogias. A planta de uma casa é uma casa? Definitivamente não. Não podemos morar dentro da planta de uma casa, nem podemos abrir sua porta ou pintar suas paredes. Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.

Pode parecer trivial, mas a dificuldade inicial do paradigma da orientação a objetos é justamente saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar de forma errada essas duas palavras como sinônimos.

## 3.5 Troca de mensagens entre objetos

Um objeto só executará determinado serviço se receber uma solicitação. Essa solicitação é, em muitos casos, encaminhada por outro objeto. Quando algum objeto necessitar que um outro objeto faça algum serviço, o primeiro deve comunicar-se com o segundo, encaminhando uma mensagem (fazer uma solicitação). Objetos comunicam-se através do envio de mensagens.

Quando se tem a comunicação entre objetos, o envio de uma mensagem envolverá três elementos: o objeto emissor, isto é, aquele que encaminha a mensagem, o objeto receptor, isto é, aquele que recebe a mensagem, e a especificação do serviço a ser executado. Lembre-se que um objeto faz um serviço executando o método correspondente de sua classe. Aqui, também denominaremos o emissor de Objeto Cliente e, por conseguinte, sua classe será denominada Classe Cliente. Ao objeto receptor da mensagem denominaremos Objeto Executor.

No envio de uma mensagem devem ser especificados o nome do objeto executor e qual o método a ser executado (também denominado Seletor). Na especificação do método devem também ser especificados os recursos necessários para a sua execução. Conforme veremos mais adiante, em muitas situações, quando um objeto solicita um serviço a outro objeto, é possível que o segundo tenha que devolver ao primeiro alguma informação resultante da execução do serviço solicitado. Veja a Figura 3.7 a seguir.

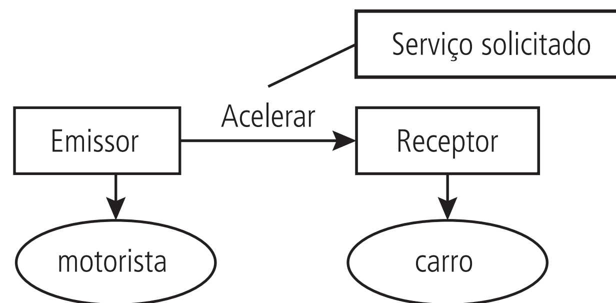



**Figura 3.7: Comunicação entre objetos por envio de mensagens**
Fonte: Elaborada pelos autoreses

A comunicação entre dois objetos através do mecanismo de envio de mensagem deve sempre considerar os serviços que o objeto sabe executar. Veja os exemplos de envio de mensagem a seguir.

**Exemplo 1**

Suponha que, em determinado problema, o objeto identificado por *umProfessor* deva solicitar que o objeto *umAluno* estude para a prova. Nesse caso, o objeto *umProfessor* (o emissor) encaminhará uma solicitação (enviará uma mensagem) ao objeto *umAluno* (receptor):

```
umAluno.estude();
```

Veja que só é possível solicitar que o objeto *umAluno* execute o serviço estude considerando que esse objeto saiba executar tal serviço. Assim, na classe do objeto umAluno, digamos, por exemplo, a classe Aluno, deve estar definido o correspondente método estude( ).

Conforme vimos anteriormente, em termos concretos, escrever um programa de computador consiste basicamente em projetar e escrever classes. Escrever uma classe consiste em elaborar um texto, expresso em determinada linguagem, que definirá quais atributos terão as instâncias da classe, quais

serviços estas poderão executar e como serão executados esses serviços. O "como" será executado um serviço consiste no método, o qual é formado por uma sequência de comandos – ordens – que o computador deve executar, representando a execução do serviço pelo objeto. Assim, a execução do comando umAluno.estude() fará com que o computador vá executar o conjunto de instruções relativo ao método *estude*, representando a execução do serviço pelo objeto *umAluno*.

**Exemplo 2**

Suponha um programa que auxilie a administração de um hotel. Imagine que os objetos umHotel e umCliente sejam, respectivamente, a abstração do hotel e de um cliente. Se, em determinado ponto desse programa, for necessário fazer com que o objeto umHotel determine a conta de um cliente, pode-se escrever:

```
valorConta = umHotel.fornecaValorConta(umCliente);
```

Para que a conta de um cliente seja determinada, é necessário que se conheçam informações relativas ao cliente. Assim, para que a execução do serviço que determinará a conta de um cliente (método fornecaValorConta) seja possível, é necessário o conhecimento do objeto *umCliente*. Por outro lado, se algum objeto encaminha uma mensagem para o objeto *umHotel*, solicitando que ele determine a conta de um cliente, esse objeto espera que seja devolvido pelo objeto *umHotel*, como resultado da execução do serviço solicitado, o valor dessa conta.

Por isso, o resultado da execução do método, isto é, o que retoma com a execução do método, é colocado em valorConta. Assim, valorConta conterá, após a execução do método, o valor da conta do objeto um Cliente.

Os elementos especificados entre parênteses, quando da solicitação de execução de um método, são denominados Argumentos.

O funcionamento do mecanismo da comunicação por envio de mensagem, juntamente com a necessidade ou não da especificação de argumentos, será visto em maiores detalhes mais adiante. Por enquanto, é importante entender que a execução de um método pode necessitar de determinados recursos, os quais deverão ser fornecidos junto com a solicitação.

Resumindo, podemos dizer que o processo de envio de mensagem de um objeto para outro envolverá os seguintes elementos:

- o objeto que encaminha a mensagem – objeto cliente (sua classe é denominada Classe Cliente);
- o objeto que recebe a mensagem – objeto receptor ou executor;
- o método a ser executado (também chamado de Seletor).

Quando da especificação do método, devem ser especificados os argumentos, caso seja necessário.

## 3.6 Criando classes em Java

### 3.6.1 Regras básicas de sintaxe

Uma classe em Java é sempre declarada com a palavra-chave *class* seguida do nome da classe. O nome da classe não pode conter espaços, deve sempre ser iniciado por uma letra. Para nomes de classes, métodos e campos em Java, o caractere sublinhado (_) e o sinal $ são considerados letras. O nome da classe não deve conter acentos e pode conter números, contanto que estes apareçam depois de uma letra. Nomes de classes não podem ser exatamente iguais às palavras reservadas de Java (mostradas no Quadro 3.1).

**Quadro 3.1: Palavras reservadas da linguagem Java**

| | | | |
|---|---|---|---|
| abstract | boolean | break | Byte |
| char | class | const | Continue |
| default | do | case | Catch |
| double | else | extends | False |
| final | finally | float | For |
| goto | if | implements | Import |
| instanceof | int | interface | Long |
| native | new | null | Package |
| private | protected | public | Return |
| short | static | strictfp | Super |
| switch | synchronized | this | Throw |
| throws | transient | true | Try |
| void | volatile | while | |

Fonte: Elaborada pelos autores

Em Java uma classe pode ser escrita (declarada) de acordo com a seguinte forma geral:

```
public class Nome_da_Classe
{
   Corpo da classe (especificação dos atributos e métodos)
}
```

Veja como o corpo da classe sempre deve iniciar com "abre chaves" ({) e finalizar com "fecha chaves" (}).

A menor classe válida em Java é a classe vazia, ou seja, é uma classe que não possui atributos ou métodos. Exemplo:

```
class Vazia
 {


 }
```

Tradicionalmente os nomes de classes começam com caracteres maiúsculos e alternam entre palavras. Exemplos de nomes de classes que seguem esta convenção são: RegistroAcademico, ContaDeLuz e CadastroDeFuncionariosDeSupermercado.

É aconselhável que os arquivos criados em editores de texto contenham somente uma classe, e os nomes dos arquivos devem ser compostos dos nomes das classes com a extensão *java*. Dessa forma, a classe Vazia deve ser criada em um arquivo *Vazia.java*.

### 3.6.2 Declarando campos (atributos) de classes em Java

A declaração dos campos nas classes em Java é simples: basta declarar o tipo de dado, seguido dos nomes dos campos que serão daquele tipo.

A Figura 3.8 mostra alguns exemplos de declaração de campos. Essa listagem corresponde ao modelo mostrado na seção 3.2, mas considerando somente os dados do modelo, e não suas operações.

```
1   class ContaBancariaSimplificada
2   {
3   /**
4   * Declaração dos campos da classe
5   */
6   //uma cadeia de caracteres para representar o nome
7
8   String nomeDoCorrentista;
9
10  //um float para representar o saldo
11
12  float saldo;
13
14  // valor booleano (true para conta especial e false para
15  // conta não especial
16
17  boolean contaÉEspecial;
```

**Figura 3.8: Conta bancária simplificada**
Fonte: Elaborada pelos autores

Algumas regras e informações sobre declaração de campos em classes em Java são mostradas a seguir:

- Nomes de campos seguem quase todas as mesmas regras de nomes de classes: devem ser iniciados por uma letra (incluindo _ e $), devem ser compostos de uma única palavra (sem espaços, vírgulas, etc.) e podem ser compostos de letras e números. Ao contrário de nomes de classes, nomes de campos podem conter acentos. Nomes de campos não podem ser iguais a nenhuma das palavras reservadas mostradas na tabela de palavras reservadas.

- Tradicionalmente nomes de campos e variáveis começam com caracteres minúsculos, alternando com maiúsculos entre palavras. Alguns exemplos de campos que seguem esta convenção são dia, nomeDoAluno, rotacoesPorMinuto e estadoDaLampada.

- Campos podem ser declarados com a forma *modificador-de-acesso tipo-ou-classe* nome-do-campo; (terminando com ponto e vírgula, como mostra o código acima). Modificadores de acesso serão vistos mais adiante, mas nos exemplos a sua declaração foi omitida propositadamente. Se vários campos forem do mesmo tipo, podemos declará-los simultaneamente com a forma *modificador-de-acesso tipo-ou-classe* nome-do-

-campo1, nome-do-campo2, nome-do-campo3; (nomes separados por vírgulas, declaração terminada com ponto e vírgula). É aconselhável evitar a declaração de muitos campos em uma mesma linha de programa para não comprometer a clareza do código.

- Dois campos da mesma classe não podem ter exatamente o mesmo nome. Por exemplo, não podemos declarar outro campo chamado saldo na classe da listagem acima, mesmo se o tipo de dado for diferente.

### 3.6.3 Métodos de classes em Java

Até agora foi mostrado como podemos criar classes e representar campos nessas classes. A maioria das classes representa modelos que têm dados e operações que manipulam estes dados. As operações dos modelos, que em Programação Orientada a Objetos são conhecidas como métodos, também seguem regras rígidas para sua criação, descritas a seguir:

- Nomes de métodos seguem as mesmas regras de nomes de campos: devem ser iniciados por uma letra (incluindo _ e $), devem ser compostos de uma única palavra (sem espaços, vírgulas, etc.), podem ser compostos de letras e números e podem conter acentos. Nomes de métodos não podem ser iguais a nenhuma das palavras reservadas mostradas anteriormente.

- Nomes de métodos refletem ações que são efetuadas nos campos da classe e/ou valores passados como argumentos para esses métodos. Embora esta não seja uma regra obrigatória, se ela for seguida os programas e classes poderão ser lidos e compreendidos mais facilmente por programadores.

- Tradicionalmente, assim como nos atributos, os nomes de métodos começam com caracteres minúsculos e alternam para maiúsculos entre palavras. Alguns exemplos de nomes de métodos que seguem esta regra são *mostraDados*, *acende* e *inicializaData*.

- Métodos não podem ser criados dentro de outros métodos, nem fora da classe a qual pertencem; ou seja, não podemos ter métodos isolados, fora de todas as classes.

Para exemplificar as regras de criação de métodos, veremos a classe *ContaBancariaSimplificada*, com campos e métodos básicos, mostrada na Figura 3.9 a seguir.

```
1   // Inicializa simultaneamente todos os dados do modelo, usando o
2   void abreContaSimples(String nome) {// nome passado como argumento e
3      nomeDoCorrentista = nome;//os outros valores com valores
4      saldo = 0.00; // default
5      contaÉEspecial = false;
6   }
7   void deposita(float valor) { // Deposita um valor na conta
8      saldo = saldo + valor;
9   }
10  boolean retira(float valor) {// Tenta Retirar um valor da conta retornando um valor booleano
11     if (contaÉEspecial == false) // informando se a operação foi realizada ou não.
12     {
13        if (valor <= saldo){ // se existe saldo suficiente...
14           saldo = saldo - valor; // faz a retirada.
15           return true; }
16        else
17           return false;
18     }
19     else  { // A conta é especial, pode retirar à vontade !
20        saldo = saldo - valor;
21        return true;
22     }
23  }
24  void mostraDados()  {// mostra os dados da conta, imprimindo os seus valores
25    System.out.print("O nome do correntista é ");
26    System.out.print(nomeDoCorrentista);
27    System.out.print("O saldo é ");
28    System.out.print(saldo);
29    if (contaÉEspecial)
30       System.out.print("A conta é especial.");
31    else
32       System.out.print("A conta é comum.");
33  }
```

**Figura 3.9: Criação de métodos da classe Conta Bancária Simplificada**
Fonte: Elaborada pelos autores

Usando a listagem acima como exemplo, podemos ver outras regras sobre declaração de métodos em classes em Java:

- O formato genérico usado para a declaração de métodos é *modificador--de-acesso tipo-ou-classe-de-retorno nome-do-método*(lista-de-argumentos). A declaração de métodos **não** é terminada com ponto e vírgula. Modificadores de acesso serão vistos mais adiante. Na listagem acima a declaração desses modificadores foi omitida propositadamente.

- Cada método deve ter, na sua declaração, um tipo ou classe de retorno, correspondente ao valor que o método deve retornar. Caso o método não retorne nada, isto é, caso ele somente execute alguma operação sem precisar retornar valores, o valor de retorno deverá ser ***void***, que nada mais é do que um tipo de retorno especial que indica que o retorno deve ser desconsiderado. Na classe acima, os métodos ***abreContaSimples*** e ***deposita*** retornam ***void***, enquanto o método ***retira*** retorna um valor do tipo *boolean*. Métodos podem retornar também instâncias de classes, mas somente um valor pode ser retornado de cada vez.

- Métodos que retornam algum valor diferente de ***void*** devem ter, em seu corpo, a palavra chave *return* seguida de uma constante ou variá-

vel do tipo ou classe que foi declarada como a de retorno do método. No Exemplo acima (classe ContaBancariaSimplificada), o método ***retira*** declara que deverá retornar um valor do tipo *booleano*, o que será feito através da palavra-chave *return*, dependendo da expressão condicional. Métodos que retornam *void* não precisam ter a palavra-chave *return* no seu corpo; e se tiverem, esta não deve ser seguida de nenhuma constante ou variável.

- Métodos podem ter listas de argumentos, ou seja, variáveis contendo valores que podem ser usados pelos métodos para efetuar suas operações. Na listagem, o método ***deposita*** recebe um argumento. Cada argumento passado para um método deve ter uma referência ou nome de variável correspondente, e cada argumento deve ser precedido de seu tipo ou classe. Opcionalmente, métodos podem não receber argumentos, como é o caso do método ***mostraDados***.

## 3.7 Escopo

O *escopo* dos campos e variáveis dentro de uma classe determina a sua visibilidade (isto é, se as variáveis ou campos podem ser acessados ou modificados em todos os métodos da classe, somente um determinado método ou mesmo somente em parte de um determinado método). A classe *Triangulo*, mostrada na Figura 3.10, será usada para exemplificar alguns pontos sobre escopo de campos e variáveis.

```
1  /**
2   * A classe Triangulo, que representa os três lados de um triângulo qualquer.
3   */
4   class Triangulo // declaração da classe
5   {
6   /**
7   * Declaração de um dos campos da classe
8   */
9   float lado1;
10
11 boolean éEquilátero()
12 {
13     boolean igualdade12,resultado;
14     igualdade12 = (lado1 == lado2); // o lado 1 é igual ao lado 2 ?
15     boolean igualdade23;
```

**Figura 3.10: Classe Triangulo**
Fonte: Elaborada pelos autores

```
1   /**
2    * O método éEquilátero verifica se o triângulo é equilátero ou não.
3    * @return true se o triângulo é equilátero, false se não for
4    */
5   boolean éEquilátero()
6    {
7     boolean igualdade12,resultado;
8     igualdade12 = (lado1 == lado2); // o lado 1 é igual ao lado 2 ?
9     boolean igualdade23;
10    igualdade23 = (lado2 == lado3); // o lado 2 é igual ao lado 3 ?
11    if (igualdade12 && igualdade23) // os três lados são iguais ?
12       resultado = true;
13    else
14       resultado = false;
15    return resultado;
16   } // fim do método éEquilátero
17   /**
18   * O método calculaPerímetro calcula o perímetro do triângulo usando seus lados.
19   * @return o perímetro do triângulo representado por esta classe
20   */
21   float calculaPerímetro()
22  {
23     float resultado = lado1+lado2+lado3;
24     return resultado;
25   } // fim do método perímetro
26
27   /**
28   * Declaração dos outros campos da classe
29   */
30   float lado2,lado3;
31   } // fim da classe Triangulo
```

**Figura 3.11: Métodos da classe Triangulo**
Fonte: Elaborada pelos autores

Algumas regras simples de escopo são mostradas a seguir:

Campos declarados em uma classe são válidos por toda a classe, isto é, o escopo de campos de uma classe é toda a classe, mesmo que os campos estejam declarados depois dos métodos que os usam, como mostrado na listagem acima, onde os campos *lado1*, *lado2* e *lado3* podem ser usados nos métodos *eEquilátero* e *calculaPerimetro* mesmo "antes" (no sentido de leitura da listagem, de cima para baixo) de serem declarados. Vale a pena notar que para o compilador Java, a ordem em que as declarações de métodos e campos aparecem é irrelevante.

Variáveis e instâncias declaradas dentro de métodos só serão válidas dentro desses métodos. Dessa forma, as variáveis *igualdade12* e *igualdade23*, declaradas no método *eEquilatero*, somente serão válidas dentro desse método. Outras variáveis podem ser declaradas em outros métodos, mas mesmo que tenham o mesmo nome serão consideradas variáveis diferentes e não relacionadas, como por exemplo, a variável *resultado*, que foi declarada como sendo do tipo *boolean* no método *eEquilatero* e como *float* no método *calculaPerimetro*.

Dentro de métodos e blocos de comandos, a ordem de declaração de variáveis e referências a instâncias é considerada: se as linhas "igualdade23 = (lado2 == lado3);" e "if (igualdade12 && igualdade23)" da listagem acima fossem trocadas, um erro de compilação ocorreria, pois o compilador não aceitaria que a variável *igualdade23* recebesse um valor antes de ser declarada. Em resumo, variáveis dentro de métodos só podem ser usadas depois de declaradas.

Variáveis passadas como argumentos para os métodos só são válidas dentro dos métodos.

## 3.8 Encapsulamento

Diferentemente da abordagem estruturada, em que dados e procedimentos são definidos de forma separada no código, na programação orientada a objeto os dados e procedimentos que manipulam esses dados são definidos numa unidade única: o objeto. Isso possibilita uma melhor modularidade do código; porém, a ideia principal é poder utilizar os objetos sem ter que se conhecer sua implementação interna, que deve ficar escondida do usuário do objeto, que irá interagir com este apenas através de sua **interface**.

À propriedade de se implementar dados e procedimentos correlacionados em uma mesma entidade e de se proteger sua estrutura interna escondendo-a de observadores externos dá-se o nome de **encapsulamento**.

O objetivo do encapsulamento é separar o usuário do objeto do programador do objeto. Seus principais benefícios são:

- possibilidade de alterar a implementação de um método ou a estrutura de dados escondidos de um objeto sem afetar as aplicações que dele se utilizam;

- criação de programas mais modulares e organizados, o que possibilita um melhor reaproveitamento do código e melhor manutenibilidade da aplicação.

Em geral, as variáveis de instância declaradas em uma definição de classe, bem como os métodos que executam operações internas sobre estas variáveis, se houver, devem ser escondidos na definição da classe. Isso é feito geralmente através de construções nas linguagens de programação conhecidas como **modificadores de acesso**.

Os modificadores de acesso são: *public*, *private*, *protected* e *package* (também conhecido como *friendly* ou *default*). O modificador *public* garante que o campo ou método da classe declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra classe. Campos e métodos que devam ser acessados (e modificados, no caso de campos) devem ser declarados com o modificador *public*.

Campos e métodos declarados com o modificador *private* só podem ser acessados, modificados ou executados por métodos da mesma classe, sendo completamente ocultos para o programador usuário que for usar instâncias dessa classe ou criar classes herdeiras ou derivadas (que serão vistas mais adiante). Campos ou métodos que devam ser ocultos totalmente de usuários da classe devem ser declarados com esse modificador.

O modificador *protected* funciona como o modificador *private*, exceto que classes herdeiras ou derivadas também terão acesso ao campo ou método marcado com esse modificador. O comportamento desse modificador será visto mais adiante.

Finalmente, campos e métodos podem ser declarados sem modificadores. Nesse caso, eles serão considerados da categoria package ou *friendly*, significando que seus campos e métodos serão visíveis (podendo ser acessados) para todas as classes pertencentes a um mesmo pacote ou *package*.

Quando definimos uma classe, é recomendado (para alguns é uma regra sagrada) que declaremos públicos apenas os métodos da sua interface. É na **interface** (ou protocolo) da classe que definimos quais mensagens podemos enviar às instâncias de uma classe, ou seja, quais são as operações que podemos solicitar que os objetos realizem. Por exemplo, na classe Triangulo acima, poderíamos reescrevê-la da seguinte forma:

```
class Triangulo // declaração da classe
{
  private float lado1;
  private float lado2;
  private float lado3;
public void inicializa(float valor1, float valor2, float valor3) {
      lado1 = valor1;
      lado2 = valor2;
      lado3 = valor3;
}
public String mostra() {
     return "Os lados do triangulo são:"+lado1+","+lado2+","+lado3;
 }
public boolean éEquilátero()
 {
  boolean igualdade12,resultado;
  igualdade12 = (lado1 == lado2); // o lado 1 é igual ao lado 2 ?
  boolean igualdade23;
  igualdade23 = (lado2 == lado3); // o lado 2 é igual ao lado 3 ?
  if (igualdade12 && igualdade23) // os três lados são iguais ?
     resultado = true;
  else
     resultado = false;
  return resultado;
 } // fim do método éEquilátero
public float calculaPerímetro()
{
   float resultado = lado1+lado2+lado3;
   return resultado;
 } // fim do método perímetro
```

**Figura 3.12: Reescrita da classe Triangulo**
Fonte: Elaborado pelos autores

## 3.9 Criando objetos e acessando dados encapsulados

Criamos objetos em Java de forma muito similar à criação de variáveis de tipos primitivos. Se uma aplicação quisesse usar a classe Triangulo, poderia declarar uma variável desse tipo da seguinte forma:

Triangulo t;

Como Java é *case sensitive* pode-se declarar dessa forma.

Esta sentença cria uma variável **t** do tipo Triangulo. Podemos dizer que t é apenas uma referência para a classe Triangulo.

Entretanto, isso não é suficiente para acessarmos os métodos e atributos públicos da classe. A sentença acima somente declara uma variável (referência) mas não cria um objeto da classe especificada (instância). Em Java, objetos são criados usando o operador **new** da seguinte forma:

```
Triangulo t1;
t1 = new Triangulo();
Triangulo t2;
t2 = new Triangulo;
```

O operador **new** cria uma instância da classe e retorna a referência do novo objeto. Como vimos, todos os objetos em Java são tratados através da referência ao endereço de memória onde o objeto está armazenado. O operador **new** realiza três tarefas:

1. Aloca memória para o novo objeto;

2. Chama um método especial de inicialização da classe denominado **construtor**;

3. Retorna a referência para o novo objeto.

É importante compreender o que ocorre na declaração e na inicialização da variável. Quando fazemos

Triangulo t1; ⟶null

O *null* é um valor válido em Java e permite comparações como: if (t == null)

É reservada uma porção da memória principal do Java (*stack*) para armazenar o endereço na memória auxiliar (*heap*) onde o objeto será armazenado. Como apenas com a declaração da variável o objeto ainda não existe, o conteúdo inicial dela será o valor nulo (***null***), indicando que ela ainda não se refere a nenhum objeto. Apenas após a inicialização

**t1 = new Triangulo();**

É que uma variável de um tipo não primitivo estará valendo algo e através dela será possível acessar os dados e operações do objeto em questão. O correto aqui é dizer que t1 se refere a um objeto. Não é correto dizer que t1 é um objeto, pois t1 é uma variável referência, apesar de que, depois de um tempo, os programadores Java falem "Tenho um objeto t1 do tipo Triangulo"; seria mais correto dizer: "Tenho uma referência t1 a um objeto do tipo Triangulo".

A Figura 3.13 mostra como seria a representação dessas instâncias na memória.

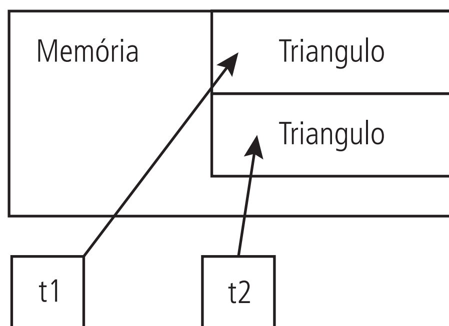



**Figura 3.13: Representação das instâncias na memória**
Fonte: Elaborada pelos autores

Podemos afirmar então que os comandos: "t1 = new Triangulo()" e "t2 = new Triangulo()" criaram dois objetos da classe Triangulo na memória, os quais são apontados respectivamente pelas variáveis de referência *t1* e *t2*. Isso quer dizer que a expressão: "if (t1==t2)" retornaria um valor falso, pois *t1* e *t2* apontam para objetos localizados em posições de memória diferentes. O operador == compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, mas o endereço em que ele se encontra; portanto, dois objetos com mesmo conteúdo são diferentes, pois estão em diferentes endereços de memória.

Agora observe a seguinte situação:

```
Triangulo t3 = t1;
```

Qual o resultado do comando acima?  O que acontece?

O operador = copia o valor de uma variável. Mas qual é o valor da variável t1? É o objeto? Não. Na verdade, o valor guardado é a referência (endereço) de onde o objeto se encontra na memória  principal. A Figura 3.14 mostra o que acontece neste caso.

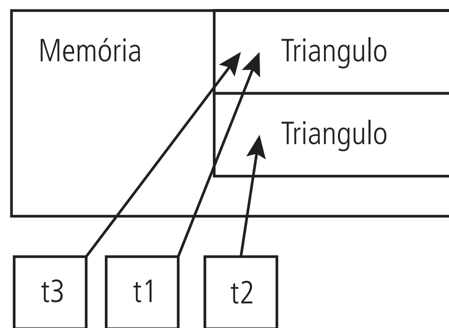



**Figura 3.14: Referências t3 e t1 apontando para o mesmo objeto**
Fonte: Elaborada pelos autores

Quando fizemos t3 = t1, t3 passa a fazer referência ao mesmo objeto que t1 referencia nesse instante. Então, nesse código em específico, quando utilizamos t1 ou t3 estamos nos referindo exatamente ao mesmo objeto! Elas são duas referências distintas, porém apontam para o mesmo objeto! Compará-las com "=" irá nos retornar true, pois o valor que elas carregam é o mesmo!

Uma vez um objeto tendo sido criado, seus métodos e atributos públicos podem ser acessados utilizando o identificador do objeto (variável que armazena sua referência) através do operador **ponto**:

<identificador>**.**<atributo>

<identificador>**.**<método>

A aplicação que criou o objeto t acima pode solicitar que ele inicialize os lados do triangulo:

```
t.inicializa(5,6,7);
```

## 3.10 Criando aplicações em Java

Em Java, podemos criar em uma classe um método especial que será considerado o ponto de entrada de um programa. A presença desse método na classe fará com que ela se torne executável, e dentro desse método poderemos ter a criação e manipulação de dados e instâncias de classes.

Esse método especial em Java é chamado ***main***, similarmente à linguagens como C e C++. Para que a classe possa ser executada pela máquina virtual Java, não basta a existência de um método chamado ***main*** – esse método

deve ter obrigatoriamente os modificadores ***public static***(nesta ordem), devendo retornar ***void*** e receber como argumento um *array* (agrupamento, conjunto ou lista) de instâncias da classe *String*; ou seja, deve ser declarado como ***public static void main(String[]* argumentos*)***. Vale a pena lembrar que a declaração de métodos **não** deve ser terminada com ponto e vírgula. Segue abaixo um exemplo de uma aplicação em Java que cria e manipula objetos da classe Triangulo.

```
1  class TestaTriangulo {
2
3  public static void main(String []argumentos) {
4
5  Triangulo t1;
6  t1 = new Triangulo();
7  Triangulo t2 = new Triangulo();
8  Triangulo t3;
9  t1.inicializa(4,3,5);
10
11 t2.inicializa(4,4,4);
12
13 Triangulo t4 = t2;
14 t3.inicializa(9,9,8);   // atenção!!!
15 t4.lado3 = 7;           // atenção!!!
16 System.out.println(t4.mostra());
17 System.out.println(t1.calculaPerímetro());
18
19 if (t2.éEquilátero())
20    System.out.println("Este triangulo é eqüilátero!");
21 else
22    System.out.println(t2.calculaPerímetro());
23
24 }
25 }
```

**Figura 3.15: Classe TestaTriangulo**
Fonte: Elaborada pelos autores

Na Figura 3.15 vimos que o modificador *public* faz com que o método seja visível de qualquer outra classe, o que será uma exigência da máquina virtual Java para executá-lo. Mais adiante veremos o papel do modificador *static*– por enquanto fiquemos com a explicação simples de que um método declarado como *static* dispensa a criação de uma instância de sua classe para que possamos chamá-lo. O tipo de retorno *void* indica que o método *main* não deverá retornar nada. O nome do método deve ser exatamente *main*, não podendo ter variações como maiúsculas, sublinhados, etc. O método deve receber como argumento um *array* de *Strings*, que deverá ter um nome (re-

ferência). O nome pode ser qualquer um que siga as regras de uso de nomes de campos em Java, mas o *array* de *Strings* deve ser declarado como String[]. *Arrays* serão discutidos mais adiante. Esse *array* de *Strings*conterá, para o método *main*, os argumentos passados para a classe executável via linha de comando do terminal do computador.

Em princípio, o método *main* poderia pertencer a qualquer classe, até mesmo a uma classe que representa um modelo qualquer. Em vez de usar esse estilo de programação, vamos tentar fazer com que o método *main*, sempre que possível, seja o único método de uma classe, estando separado da representação dos modelos. Além de tornar os programas executáveis mais claros, esta prática serve para melhorar a distinção entre os diferentes papéis de programadores que usam linguagens orientadas a objetos, discutida na seção.

Na figura acima, percebemos na linha 8 a declaração de uma variável de referência “t3” e, curiosamente, esta não está associada a nenhuma instância; portanto, não há como acessar nenhum atributo ou método da classe, pois ela não aponta para nenhum objeto da classe. Nesse caso o compilador Java apontará um erro na linha 14, pois há uma tentativa de acesso ao método da classe por essa variável de referência. Outro ponto curioso está na linha 13, onde uma variável de referência “t4” é criada e associada ao objeto apontado pela variável “t2”. Nesse caso a variável de referência “t4” existe de fato e pode fazer os mesmos acessos que a variável “t2”. Como essas variáveis apontam para a mesma instância, qualquer operação feita através da variável “t2” terá efeito na variável “t4” e vice-versa. No entanto, na linha 15, ocorre uma tentativa da variável “t4” de modificar o valor do atributo *lado3* da classe Triangulo. Como este atributo está encapsulado na classe através do modificador de acesso ***private***, o compilador Java acusará um erro de permissão. Isso ocorre porque o modificador ***private*** impede o acesso direto do atributo fora da classe onde este foi definido.

Outro exemplo é mostrado na Figura 3.16 a seguir.

```
1  Class TestaContaBancáriaSimplificada {
2  public static void main(String []args) {
3
4  ContaBancariaSimplificada c1;
5  c1 = new ContaBancariaSimplificada();
6  c1.abreContaSimples("Maria");
7  ContaBancariaSimplificada c2 = new ContaBancariaSimplificada();
8  c2.abreContaSimples("João");
9  c1.deposita(200);
10 c2.saldo = 1000; //atenção!!!
11 if (c1.retira(300))
12     System.out.println("Saque realizado com sucesso!")
13 else
14     System.out.println("saldo insuficiente!");
15 c1.mostradados();
16 c2.mostradados();
17 }
18 }
```

**Figura 3.16: Classe TestaContaBancariaSimplificada**
Fonte: Elaborada pelos autores

Na listagem acima temos uma aplicação Java que manipula a classe ContaBancariaSimplificada, criada na seção 3.6.3. Declaramos duas variáveis de referência c1 e c2 e associamos a duas instâncias de ContaBancariaSimplificada. Depois foram feitas várias operações nessas instâncias, como: abertura de conta, depósito, saque, etc. Observe que os métodos são invocados de acordo com sua definição na classe ContaBancariaSimplificada. Observe na linha 11 que há uma falha no **encapsulamento** dessa classe, pois foi permitido que o objeto apontado pela referência c2 alterasse diretamente o atributo *saldo* da classe, o que caracteriza um erro grave! Imagine se você, correntista de um banco, pudesse alterar o saldo da sua conta sem fazer nenhum depósito. Com certeza os bancos estariam falidos!. Para consertar isso, a classe deveria ter sido definida com todos os seus atributos declarados com o modificador **private**.

## Resumo

Nesta aula exploramos os princípios básicos da programação orientada a objetos. Vimos a importância de criar modelos para representarmos objetos, pessoas, itens, tarefas, processos, conceitos, ideias, etc. Discutimos que os objetos são criados com base nas classes que foram escritas em uma linguagem de programação qualquer. As classes representam o conceito geral de uma coisa, enquanto objetos representam instâncias concretas de uma classe. Uma classe especifica quais serão os atributos de todo objeto pertencente a ela bem como quais serviços esse objeto poderá executar. Os objetos armazenam dados em campos que contêm tipos. Os objetos têm métodos que utilizamos para a troca de mensagens entre eles.

O *escopo* dos campos e variáveis dentro de uma classe determina a sua visibilidade (isto é, se as variáveis ou campos podem ser acessadas ou modificadas em todos os métodos da classe, se somente em um determinado método ou mesmo se somente em parte de um determinado método).

O **encapsulamento** é a propriedade de implementar dados e procedimentos correlacionados em uma mesma entidade e de proteger sua estrutura interna, escondendo-a de observadores externos. O encapsulamento é implementado através dos modificadores de acesso. São eles: *public*, *private*, *protected* e *default*. Os objetos são acessados por referências, ou seja, quando declaramos uma variável para associar a um objeto, na verdade essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de referência. A criação de objetos em Java é feita através do operador ***new***.

O operador ***new***, depois de alocar a memória para esse objeto, devolve uma "flecha", isto é, um valor de referência. Quando você atribui isso a uma variável, essa variável passa a se referir para esse mesmo objeto. O método *main*é usado em Java para a criação de classes executáveis (aplicações). Dentro desse método poderemos ter a criação e manipulação de dados e instâncias de classes.

## Atividades de aprendizagem

1. Escreva um modelo para representar uma lâmpada que está à venda em um supermercado. Que dados devem ser representados por esse modelo?

2. Crie um modelo para representar um time de um esporte qualquer em um campeonato desse esporte. Que dados e operações esse modelo deve ter?

3. Modifique a operação **mostraDados** do modelo **ContaBancariaSimplificada** ( ) para que, caso o saldo esteja negativo, uma mensagem de alerta seja impressa. Dica: O saldo só poderá ser negativo se a conta for especial.

4. Crie uma classe em Java correspondente ao modelo criado na questão 1.

5. Crie uma classe em Java correspondente ao modelo criado na questão2.

6. Modifique o método "abreContaSimples" da classe "ContaBancariaSimplificada" de forma que o cliente só possa abrir uma conta com um valor mínimo de R$ 100,00.

**7.** Identifique e explique o(s) erro(s) na classe abaixo:

```
1 class Registro De Eleitor
2 {
3 /**
4 * Declaração dos campos desta classe
5 */
6   int tituloDeEleitor; // número do título do eleitor
7   String nome; // nome do eleitor
8   short zonaEleitoral; // número da zona eleitoral
9 } // fim da classe
```

**8.** Identifique e explique o(s) erro(s) na classe abaixo:

```
1 class Teste2
2 {
3 /**
4 * Declaração dos campos desta classe
5 */
6 int num1,num2;
7 /**
8 * Declaração dos métodos desta classe
9 */
10 int maior()
11 {
12   if (num1 > num2)
13     return true;
14   else return false;
15 }
16 void menor()
17 {
18 if (num1 < num2)
19   return num1;
20 else return num2;
21 }
22 } // fim da classe
```

**9.** Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class TesteImpressao
2 {
3 main(String[] args)
4 {
5    System.out.println("7+2="+(7+2));
6    System.out.println("7-2="+(7-2));
7    System.out.println("7*2="+(7*2));
8    System.out.println("7/2="+(7/2));
9    return true;
10 }
11 } // fim da classe
```

**10.** Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class TesteDatas
2 {
3 public void static main(String[] args)
4 {
5   Data2 hoje = new Data2();
6   hoje.inicializaData(7,1,2001);
7   Data2 amanha;
8   amanha.inicializaData(8,1,2001);
9   System.out.println(amanha.eIgual(hoje));
10 }
11 } // fim da classe
```

Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class TesteObjetos
2 {
3 public static void main(String[] args)
4 {
5   Data a;
6   Data b = new Data();
7   b = null;
8   b = a;
9 }
10 } // fim da classe
```

**11.** Escreva uma aplicação em Java que demonstre o uso de instâncias da classe, que deve ter sido criada como resposta ao exercício 2.

**12.** Escreva uma aplicação em Java que demonstre o uso de instâncias da classe, que deve ter sido criada como resposta ao exercício 1.

**13.** Escreva uma aplicação que demonstre o uso de instâncias da classe ContaBancariaSimplificada implementada na seção 3.6.3. Crie um método "transfere" do tipo *booleano* que tenha como parâmetros dois objetos do tipo Conta e o valor a ser transferido de uma conta para outra. Esse método deverá utilizar os métodos *deposita* e *retira* da classe. O tipo *booleano* de retorno do método é para sinalizar se a operação de transferência de valores foi realizada com sucesso, ou seja, se o valor a ser transferido da conta de origem for menor ou igual ao saldo dessa conta.

**14.** Qual a importância de usar o **encapsulamento** na definição de classes? Exemplifique.

**15.** A classe abaixo pode ser compilada sem erros. Quando for executado, o programa imprimirá o resultado da comparação na linha 11, que é *true*, mas o resultado da comparação na linha 12 é *false*. Explique por que.

```
1 class DemoDataCopiada
2 {
3 public static void main(String[] argumentos)
4 {
5   Data aniversariodaMaria = new Data();
6   Data meuaniversario= new Data();
7   Data corpuschristi2010;
8   meuaniversario.inicializaData(3,6,2010);
9   aniversariodaMaria.inicializaData (3,6,2010);
10  corpuschristi2010= meuaniversario;
11  System.out.println(corpuschristi2010 == meuaniversario);
12  System.out.println(corpuschristi2010 == aniversariodaMaria);
13 }
14 }
```

Poste suas soluções no AVEA.

# Aula 4 – Construtores, sobrecarga, atributos e métodos de classe

## Objetivos

Compreender o que são e para que servem os construtores.

Criar construtores para classes.

Conhecer a sobrecarga de métodos, saber utilizá-la e identificar quais os principais cuidados que devemos ter com ela.

## 4.1 Introdução

Na aula anterior vimos que podemos criar aplicações que utilizam instâncias de classes definidas pelo usuário ou já existentes. Para criar instâncias, precisamos usar a palavra-chave *new*, que criará a instância da classe e, em geral, associará a instância recém-criada a uma referência, para que métodos da instância possam ser executados.

Até agora, além de inicializar a instância da classe com a palavra-chave *new*, usamos métodos para inicializar os campos das instâncias. O uso desses métodos é de responsabilidade do programador usuário das classes – para criar as instâncias, o compilador obriga o uso da palavra-chave *new*, mas não obriga o uso dos métodos de inicialização. Dessa maneira, por esquecimento, um programador usuário pode criar a instância de uma classe mas não inicializar os seus dados.

Para exemplificar os erros que podem ocorrer quando instâncias são inicializadas mas os dados encapsulados não o são, consideremos as classes RegistroAcademicoSemConstrutor (listagem da Figura 4.1a e b) que encapsula os dados de um registro acadêmico simples (basicamente para cálculo da mensalidade) e a classe DemoRegistroAcademicoSemConstrutor (listagem da Figura 4.2) que demonstra usos de instâncias dessa classe.

```
1   /** A classe RegistroAcademicoSemConstrutor, que contém campos para
2    representar dados simples sobre um registro acadêmico.
3   */
4   class RegistroAcademicoSemConstrutor // declaração da classe
5   {
6    /**
7   * Declaração dos campos da classe
8   */
9    private String nomeDoAluno; // O nome do aluno
10   private int númeroDeMatrícula; // O número de matrícula
11   private byte códigoDoCurso; // O código do curso do aluno (1..4)
12   private double percentualDeCobrança; // O percentual a ser cobrado do aluno, 0 a 100%
13
14   /**
15   * O método inicializaRegistroAcademicoSemConstrutor recebe argumentos para
16   * inicializar os campos da classe RegistroAcademicoSemConstrutor.
17   * @param n o nome do aluno
18   * @param m o número de matrícula
19   * @param c o código do curso
20   * @param p o percentual de bolsa
21   */
22   public void inicializaRegistroAcademicoSemConstrutor(String n,int m,byte c,double p)
23   {
24   nomeDoAluno = n; númeroDeMatrícula = m;
25   códigoDoCurso = c; percentualDeCobrança = p;
26   } // fim do método inicializaRegistroAcademicoSemConstrutor
```

**Figura 4.1a: A classe RegistroAcademicoSemConstrutor, que encapsula alguns dados de um registro acadêmico**
Fonte: Elaborada pelos autores

```
27
28    /**
29    * O método calculaMensalidade calcula e retorna a mensalidade do aluno usando
30    * o código do seu curso e o percentual de cobrança.
31    * @return o valor da mensalidade do aluno
32   */
33    public double calculaMensalidade()
34  {
35    double mensalidade = 0; // valor deve ser inicializado
36   // Primeiro, dependendo do curso do aluno, determina a mensalidade básica
37    if (códigoDoCurso == 1) // Arquitetura
38     mensalidade = 450.00;
39   if (códigoDoCurso == 2) // Ciência da Computação
40   mensalidade = 500.00;
41   if (códigoDoCurso == 3) // Engenharia da Computação
42   mensalidade = 550.00;
43   if (códigoDoCurso == 4) // Zootecnia
44   mensalidade = 380.00;
45   // Agora calcula o desconto com o percentual de cobrança. Se o percentual de
46   // cobrança for zero, a mensalidade também o será.
47   if (percentualDeCobrança == 0) mensalidade = 0;
48   // Senão, calculamos com uma fórmula simples.
49   else mensalidade = mensalidade * 100.0 / percentualDeCobrança;
50   return mensalidade;
51   } // fim do método calculaMensalidade
52   } // fim da classe RegistroAcademicoSemConstrutor
```

**Figura 4.1b: A classe DemoRegistroAcademicoSemConstrutor, que demonstra o uso de instâncias da classe RegistroAcademicoSemConstrutor**
Fonte: Elaborada pelos autores

```
1  /**
2  * A classe DemoRegistroAcademicoSemConstrutor, que demonstra o uso de instâncias da
3  * classe RegistroAcademicoSemConstrutor.
4  */
5  class DemoRegistroAcademicoSemConstrutor // declaração da classe
6  {
7  /**
8  * O método main permite a execução desta classe. Este método contém declarações de
9  * algumas instâncias da classe RegistroAcademicoSemConstrutor, e demonstra seu uso.
10 * @param argumentos os argumentos que podem ser passados para o método via linha
11 * de comando, mas que neste caso serão ignorados.
12 */
13 public static void main(String[] argumentos)
14 {
15 // Declaramos duas referências a instâncias da classe
16 // RegistroAcademicoSemConstrutor, e as inicializamos com a palavra-chave new
17 RegistroAcademicoSemConstrutor michael = new RegistroAcademicoSemConstrutor();
18 RegistroAcademicoSemConstrutor roberto = new RegistroAcademicoSemConstrutor();
19 // Chamamos o método de inicialização somente para uma das instâncias
20 michael.inicializaRegistroAcademicoSemConstrutor("Michael Goodrich",34980030,
21 (byte)2,100.0);
22 // Calculamos a mensalidade dos dois alunos - mesmo que um não tenha sido
23 // inicializado !
24 System.out.println("A mensalidade do Michael é "+michael.calculaMensalidade());
25 System.out.println("A mensalidade do Roberto é "+roberto.calculaMensalidade());
26 } // fim do método main
27
28 } // fim da classe DemoRegistroAcademicoSemConstrutor
```

**Figura 4.2: Classe DemoRegistroAcademicoSemConstrutor**
Fonte: Elaborada pelos autores

Podemos ver que na Figura 4.1b uma instância da classe RegistroAcademico-SemConstrutor não teve seus dados inicializados, o que não impede o compilador de compilar corretamente a classe, nem a máquina virtual Java de a interpretar. O resultado do programa na figura 4.1b é mostrado a seguir:

- A mensalidade do Michael é 500.0

- A mensalidade do Roberto é NaN

A mensalidade do primeiro aluno foi calculada corretamente: a mensalidade de seu curso é 500 reais e o percentual de cobrança é 100% (sem bolsa). No caso do segundo aluno, os dados simplesmente não foram informados – para a sua instância, os valores numéricos foram considerados como zero, e a mensalidade calculada como 0 _ 100:0=0:0 (veja a linha 47 da Figura 4.1a). A divisão de zero por zero (quando ambos são valores de ponto flutuante) é representada por Java como o valor NaN (*Not a Number*, não é um número). Em muitas situações será necessário forçar o programador usuário a passar dados para as instâncias criadas em classes e programas para que estas tenham sentido. Isto pode ser feito usando construtores. Este capítulo discutirá a criação e uso de construtores e de sobrecarga de métodos, que permite que criemos métodos com nomes iguais, mas funções diferentes

## 4.2 O que são construtores?

*Construtores* são métodos especiais chamados automaticamente quando instâncias são criadas através da palavra-chave *new*. Através da criação de construtores, podemos garantir que o código que eles contêm será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com *new*, o que causará a execução do construtor.

Construtores são particularmente úteis para inicializar campos de instâncias de classes para garantir que, quando métodos dessas instâncias forem chamados, elas contenham valores específicos e não os *defaul*t Caso os campos de uma instância não sejam inicializados, os seguintes valores serão adotados:

- Campos do tipo *boolean* são inicializados automaticamente com a constante *false*.

- Campos do tipo *char* são inicializados com o caracter cujo código *Unicode* é zero e que é impresso como um espaço.

- Campos de tipos inteiros (*byte*, *short*, *long*, *int*) ou de ponto flutuante (f*loat*, *double*) são automaticamente inicializados com o valor zero, do tipo do campo declarado.

- Instâncias de qualquer classe, inclusive da classe *String*, são inicializadas automaticamente com *null*.

As diferenças básicas entre construtores e outros métodos são:

- Construtores devem ter **exatamente** o mesmo nome da classe a que pertencem, inclusive considerando maiúsculas e minúsculas.

- Construtores não podem retornar nenhum valor, nem mesmo *void*; portanto, devem ser declarados sem tipo de retorno.

- Construtores não devem receber modificadores como *public* ou *private*, e serão públicos se a classe for pública. Não há muito sentido em declarar um construtor como sendo *private*, a não ser quando um construtor privado for chamado de outro construtor público.

Exemplo de um construtor:

```
class Conta {
  int numero;
  Cliente titular;
  double saldo;
  double limite;
// construtor
 Conta() {
  System.out.println("Construindo uma conta.");
 }
// ..
}
```

Então, quando fizermos:

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

A mensagem "construindo uma conta" aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado.

## 4.3 Sobrecarga de métodos

A sobrecarga de métodos nada mais é do que termos dois métodos com o mesmo nome, porém listas de argumentos diferentes.

A sobrecarga permite criar diversas versões de um método, com listas de argumentos (valores) diferentes, para a conveniência dos chamadores. Por exemplo, se você tiver um método que use somente um *int*, o código que o chamar terá que converter, digamos, um *double* em um *int* antes de chamar seu método. Mas, se você sobrecarregou o método com outra versão que usa um *double*, tornou tudo mais fácil para o chamador.

Os métodos sobrecarregados têm muito mais flexibilidade.

Exemplo válido de sobrecarga de métodos:

```
class CSoma
{ public:
// Declaração do método Soma, parâmetro do tipo int, sem nome
int Soma( int , int );
// Declaração do método Soma, parâmetro do tipo float, com nomes
a e b
float Soma( float a, float b );
};
```

## 4.4 Cuidados com sobrecarga de métodos

Um cuidado adicional deve ser tomado pelo programador das classes quando for criar métodos sobrecarregados: Java permite que alguns tipos nativos de dados sejam "promovidos", isto é, aceitos como sendo de outros tipos contanto que nada se perca na representação. Dessa forma, um valor do tipo *byte* pode ser aceito por um método que espere um valor do tipo *int*, já que este pode representar *bytes* sem perda de informação.

O mesmo não ocorre em casos onde um método ou construtor espere um certo tipo de dado e a chamada ao método contenha, como argumento, um tipo de dado que nem sempre pode ser contido no esperado: por exemplo, nem sempre um valor do tipo *double* pode ser representado por um valor do tipo *int*. Em muitos casos, podemos usar a conversão explícita ou *cast* para forçar o rebaixamento, geralmente com perda de precisão.

Quando for necessário determinar qual construtor ou método sobrecarregado deve ser chamado, o compilador pode, se for necessário, promover tipos de dados para que um construtor ou método com a assinatura adequada seja usado, mas nunca fará o rebaixamento, o que pode causar erros de compilação.

## Resumo

Nesta aula vimos como podemos inicializar e criar instâncias de classes. Vimos também que existe um método especial chamado de Construtor, e que através da sua criação podemos garantir que o código que ele contém será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com *new*, o que causará a execução do construtor.

Vimos também que a sobrecarga de métodos nada mais é do que termos dois métodos com o mesmo nome, porém listas de argumentos diferentes. Pudemos ver que temos que tomar cuidado na hora de utilizar a sobrecarga de métodos, porque quando for necessário determinar qual construtor ou método sobrecarregado deve ser chamado, o compilador pode, se for necessário, promover tipos de dados para que um construtor ou método com a assinatura adequada seja usado, mas nunca fará o rebaixamento, o que pode causar erros de compilação.

## Atividades de aprendizagem

1. Escreva um construtor para a classe Data que receba os valores correspondentes ao dia, mês e ano, e inicialize os campos da classe, verificando antes se a data é válida.

2. Escreva um construtor para a classe Lâmpada, de forma que instâncias desta só possam ser criadas se um estado inicial for passado para o construtor. Este estado pode ser o valor *booleano* que indica se a lâmpada está acesa (*true*) ou apagada (*false*).

3. Defina uma classe Professor com os dados: nome do professor, nome do departamento, data de admissão, número de registro. Inclua na classe um construtor para setar os dados e um método para imprimir o conteúdo.

4. Reescreva a classe Professor de forma que a data de admissão seja um objeto da classe Data.

5. Construa uma classe ContaEmBanco. Atributos mínimos necessário: nome do cliente e saldo. Na construção, no mínimo o nome do cliente deve ser informado, opcionalmente com um depósito inicial.

6. Defina uma classe para conter informações sobre um funcionário de uma empresa (classe Funcionario). Quais são os atributos dessa classe? Inclua entre eles o salário que o funcionário deve receber por hora trabalhada. Implemente, para essa classe, pelo menos três métodos construtores: um que receba apenas o nome do funcionário e assuma valores *default* para os demais atributos (assuma que o funcionário deve receber dois reais por hora trabalhada); o segundo construtor deve receber, além do nome, o valor que o referido trabalhador deve receber por hora trabalhada. Identifique e implemente demais métodos que achar conveniente para um objeto da classe Funcionario.

7. Quais são os principais cuidados que devemos ter com a sobrecarga de métodos? Cite algum exemplo.

Poste suas respostas no AVEA.

# Aula 5 – Campos e métodos estáticos

## Objetivos

Definir o que são campos e métodos estáticos e como utilizá-los para escrever melhor seu programa em Java.

Saber diferenciar métodos estático de métodos não estáticos.

Saber como declarar campos que serão compartilhados entre todas as instâncias das classes de uma aplicação.

## 5.1 Introdução

Vimos nas aulas anteriores que cada instância de uma classe terá uma cópia de todos os campos declarados na classe – esta é uma das características mais úteis de linguagens de programação orientadas a objeto: a possibilidade da criação de novos "tipos de dados" que são compostos de outros dados convenientemente agrupados em uma única entidade.

As instâncias de uma classe são, então, independentes entre si: a modificação do campo **dia** de uma instância da classe Data não afeta o valor do mesmo campo em outra instância. Apesar de esperado, nem sempre esse comportamento é desejável: não foi visto, até agora, um mecanismo que permita o compartilhamento de informações entre todas as instâncias de uma classe, o que poderia ser usado, por exemplo, para algum tipo de controle interno.

Nesta aula veremos como podemos declarar campos que serão compartilhados entre todas as instâncias de uma mesma classe em uma aplicação, assim como a criação de métodos que não precisam de instâncias de classes para ser executados.

## 5.2 Campos estáticos em classe

Campos estáticos em uma classe são compartilhados por todas as instâncias dessa classe; em outras palavras, somente um valor será armazenado em um campo estático, e caso esse valor seja modificado por uma das instâncias dessa classe, a modificação será refletida em todas as outras instâncias dessa classe.

Campos estáticos são declarados com o modificador *static*, que deve ser declarado antes do tipo de dado do campo e pode ser combinado com modificadores de acesso como *public* e *private*.

Campos estáticos são também conhecidos como **campos de classes**, já que esses campos poderão ser acessados diretamente usando o nome da classe, sem que seja necessária a criação de uma instância da classe e uma referência para essa instância. Em contraste, campos que podem ter diferentes valores para cada instância da mesma classe (como os vistos nos exemplos anteriores) são conhecidos como **campos de instâncias**.

As duas maiores utilidades de campos estáticos em classes são manter uma informação ou estado para todas as instâncias de uma classe que possa ser modificada ou acessada por qualquer das instâncias e armazenar valores que não serão modificados nem serão variáveis por instâncias de classe, isto é, **valores constantes**.

Como exemplo de informação que deve ser compartilhada por todas as instâncias de uma classe, consideramos o problema da simulação de um caixa de banco. Suponhamos que nesse banco não exista um sistema de senhas ou fila única; então cada caixa é independente das outras em relação ao atendimento do cliente; isto é, cada caixa terá que ter sua própria fila, e os clientes serão atendidos à medida que o cliente anterior sair da fila. Para ilustrar, a Figura 5.1 mostra, à esquerda, um banco com cinco caixas, cada uma com sua fila, e à direita, um banco com cinco caixas e fila única.

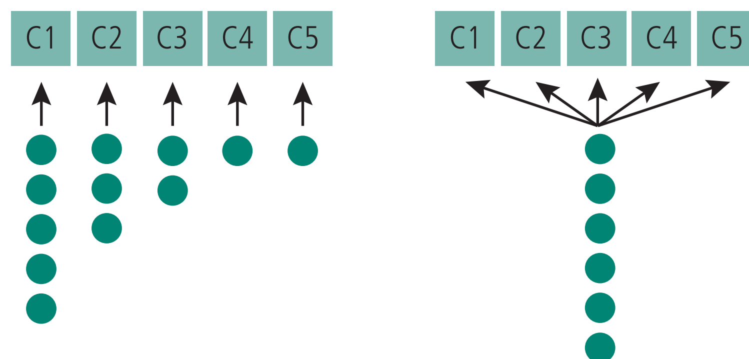



**Figura 5.1: Bancos com e sem fila única**
Fonte: Elaborada pelos autores

Para demonstração do uso de campos estáticos que representem um único valor para todas as instâncias de uma classe, vamos criar classes que simulem o atendimento de caixas em bancos com e sem fila única. Uma classe que simula o atendimento de uma caixa de banco com fila única é mostrada abaixo (Figura 5.2).

```
1  /**
2   * A classe SimuladorDeCaixaDeBanco0, que simula
     um caixa de banco, onde o    atendimento
3   * é feito através de uma senha que indica a ordem
     de atendimento. O nome da classe tem
4 * o número 0 porque esta classe será melhorada posteriormente.
5 */
6 class SimuladorDeCaixaDeBanco0 // declaração da classe
7 {
8 /**
9 * Declaração dos campos da classe
10 */
11 private int númeroDoCliente; // o número do cliente a ser atendido
12 private int númeroDoCaixa; // o número do caixa (para sua identificação)
13
14 /**
15 * O construtor da classe inicializa o número do caixa (para identificação),
16 * inicializa o contador de clientes e imprime uma mensagem.
17 * @param n o número do caixa dentro do banco
18 */
```

**Figura 5.2a: Classe Simulador de Caixa de Banco**
Fonte: Elaborada pelos autores

```
19 SimuladorDeCaixaDeBanco0(int n)
20 {
21  númeroDoCaixa = n;
22  númeroDoCliente = 0;
23  System.out.println("Caixa "+númeroDoCaixa+" iniciou operação.");
24 } // fim do construtor
25
26 /**
27 * O método próximoAtendimento simula o atendimento de um cliente.
28 */
29 public void iniciaAtendimento()
30 {
31  númeroDoCliente = númeroDoCliente + 1; // o próximo cliente será chamado
32  System.out.print("Cliente com a senha número "+númeroDoCliente+",favor");
33  System.out.println("dirigir-se ao caixa número "+númeroDoCaixa+".");
34 }
35
36 } // fim da classe SimuladorDeCaixaDeBanco0
```

**Figura 5.2b: Classe Simulador de Caixa de Banco – continuação**
Fonte: Elaborada pelos autores

A demonstração do funcionamento da classe SimuladorDeCaixaDeBanco0 através da criação e uso de várias instâncias da classe é feita abaixo (Figura 5.3).

```
1 /**
2 * A classe DemoSimuladorDeCaixaDeBanco0, que demonstra usos de instâncias da classe
3 * SimuladorDeCaixaDeBanco0, em especial, o problema que ocorre por causa da
4 * independência das instâncias da classe SimuladorDeCaixaDeBanco0.
5 */
6 class DemoSimuladorDeCaixaDeBanco0 // declaração da classe
7 {
8 /**
9 * O método main permite a execução desta classe. Este método contém declarações
10 * de algumas instâncias da classe SimuladorDeCaixaDeBanco0, e demonstra sua criação
11 * e uso.
12 * @param argumentos os argumentos que podem ser passados para o método via linha
13 * de comando, mas que neste caso serão ignorados.
14 */
15 public static void main(String[] argumentos)
16 {
17 // Declaramos cinco referências a instâncias da classe SimuladorDeCaixaDeBanco0,
18 // e as usamos para simular os caixas de um banco.
19 SimuladorDeCaixaDeBanco0 c1 = new SimuladorDeCaixaDeBanco0(1);
20 SimuladorDeCaixaDeBanco0 c2 = new SimuladorDeCaixaDeBanco0(2);
```

**Figura 5.3a: Classe simulador de caixa de banco com várias instâncias**
Fonte: Elaborada pelos autores

```
21 SimuladorDeCaixaDeBanco0 c3 = new SimuladorDeCaixaDeBanco0(3);
22 SimuladorDeCaixaDeBanco0 c4 = new SimuladorDeCaixaDeBanco0(4);
23 SimuladorDeCaixaDeBanco0 c5 = new SimuladorDeCaixaDeBanco0(5);
24 // Fazemos várias simulações de atendimento com balanceamento desigual.
25 c1.iniciaAtendimento();
26 c2.iniciaAtendimento();
27 c3.iniciaAtendimento();
28 c4.iniciaAtendimento();
29 c5.iniciaAtendimento();
30 c1.iniciaAtendimento();
31 c2.iniciaAtendimento();
32 c3.iniciaAtendimento();
33 c1.iniciaAtendimento();
34 c2.iniciaAtendimento();
35 c1.iniciaAtendimento();
36 c1.iniciaAtendimento();
37 c1.iniciaAtendimento();
38 c1.iniciaAtendimento();
39 } // fim do método main
40
41 } // fim da classe DemoSimuladorDeCaixaDeBanco0
```

**Figura 5.3b: Classe simulador de caixa de banco com várias instâncias – continuação**
Fonte: Elaborada pelos autores

Uma aplicação que usa instâncias da classe SimuladorDeCaixaDeBanco pode ser reescrita a partir das Figura 5.2a e Figura 5.2b, trocando-se as instâncias da classe SimuladorDeCaixaDeBanco por instâncias da classe SimuladorDe-CaixaDeBanco0. O resultado de uma aplicação que usa instâncias da classe SimuladorDeCaixaDeBanco é mostrado a seguir:

1. Caixa 1 iniciou operação.
2. Caixa 2 iniciou operação.
3. Caixa 3 iniciou operação.
4. Caixa 4 iniciou operação.
5. Caixa 5 iniciou operação.
6. Cliente com a senha número 1, favor dirigir-se ao caixa número 1.
7. Cliente com a senha número 2, favor dirigir-se ao caixa número 2.
8. Cliente com a senha número 3, favor dirigir-se ao caixa número 3.
9. Cliente com a senha número 4, favor dirigir-se ao caixa número 4.
10. Cliente com a senha número 5, favor dirigir-se ao caixa número 5.
11. Cliente com a senha número 6, favor dirigir-se ao caixa número 1.
12. Cliente com a senha número 7, favor dirigir-se ao caixa número 2.
13. Cliente com a senha número 8, favor dirigir-se ao caixa número 3.
14. Cliente com a senha número 9, favor dirigir-se ao caixa número 1.
15. Cliente com a senha número 10, favor dirigir-se ao caixa número 2.
16. Cliente com a senha número 11, favor dirigir-se ao caixa número 1.
17. Cliente com a senha número 12, favor dirigir-se ao caixa número 1.
18. Cliente com a senha número 13, favor dirigir-se ao caixa número 1.
19. Cliente com a senha número 14, favor dirigir-se ao caixa número 1.

Com este resultado, podemos ver que as instâncias da classe SimuladorDe-CaixaDeBanco.java compartilham o mesmo valor através do campo estático numeroDoCliente.

Um outro uso de campos estáticos em classes é o de criação de constantes, que serão compartilhadas por todas as instâncias da classe e, em geral, imutáveis. Constantes em classes geralmente são acessadas através da classe e não de instâncias dessa classe. Como exemplo, consideremos a classe ConstantesMatematicas, mostrada no código abaixo.

```
1 /**
2 * A classe ConstantesMatematicas, que contém vários valores que são constantes.
3 * Esta classe não tem métodos, e todos os campos são declarados como static, assim
4 * não é necessário criar instâncias desta classe para acessar os valores. Para que
5 * os valores possam ser acessados de fora da classe sem a necessidade de métodos os
6 * campos são declarados como public. Para garantir que os valores não poderão ser
7 * modificados, os campos também são declarados como final.
8 */
9 class ConstantesMatematicas // declaração da classe
10 {
11 /**
12 * Declaração dos campos da classe
13 */
14 // A raiz quadrada de 2
15 final static public double raizDe2 = 1.4142135623730950488;
16 // A raiz quadrada de 3
17 final static public double raizDe3 = 1.7320508075688772935;
18 // A raiz quadrada de 5
19 final static public double raizDe5 = 2.2360679774997896964;
20 // A raiz quadrada de 6: podemos usar as constantes já definidas
21 final static public double raizDe6 = raizDe2*raizDe3;
22 } // fim da classe ConstantesMatematicas
23
24 // Valores obtidos no livro Manual de Fórmulas e Tabelas Matemáticas, Murray R.
25 // Spiegel, Coleção Schaum, editora McGraw-Hill
```

**Figura 5.4: Classe ConstantesMatematicas**
Fonte: Elaborada pelos autores

Um ponto importante do código acima é que as constantes que são declaradas com o modificador static, para que sejam os mesmo valores independentemente de quantas instâncias da classe sejam criadas, também são declaradas com o modificador final. O modificador final faz com que os valores dos campos, depois de declarados, não possam mais ser modificados, o que é desejável para campos que representam constantes. A aplicação no código a seguir demonstra usos das constantes na classe ConstantesMatematicas.

```
1 /**
2 * A classe DemoConstantesMatematicas, que demonstra usos da classe
3 * ConstantesMatematicas, em especial, o acesso aos campos estáticos da classe.
4 */
5 class DemoConstantesMatematicas // declaração da classe
6 {
7 /**
8 * O método main permite a execução desta classe. Este método demonstra o acesso a
9 * campos estáticos da classe ConstantesMatematicas, através de instâncias e através
10 * do acesso direto.
11 * @param argumentos os argumentos que podem ser passados para o método via linha
12 * de comando, mas que neste caso serão ignorados.
13 */
```

**Figura 5.5a: Classe DemoConstantesMatematicas**
Fonte: Elaborada pelos autores

```
27 public static double pésParaCentímetros(double pés)
28 {
29 double centímetros = pés*30.48;
30 return centímetros;
31 }
32
33 /**
34 * O método milhasParaQuilômetros converte o valor passado em milhas para quilômetros.
35 * @param milhas o número de milhas
36 * @return o número de quilômetros correspondente ao número de milhas
37 */
38 public static double milhasParaQuilômetros(double milhas)
39 {
40 double quilômetros = milhas*1.609;
41 return quilômetros;
42 }
43
44 } // fim da classe ConversaoDeUnidadesDeComprimento
45
46 // Fórmulas obtidas no livro Manual de Fórmulas e Tabelas Matemáticas, Murray R.
47 // Spiegel, Coleção Schaum, editora McGraw-Hill
```

**Figura 5.5b: Classe DemoConstantesMatematicas – continuação**
Fonte: Elaborada pelos autores

Podemos ver no código apresentado acima que os valores constantes são iguais para todas as instâncias da classe ConstantesMatematicas, e que a criação de instâncias de uma classe onde todos os campos são declarados como *static* não é necessária: podemos acessar os campos diretamente (já que eles são declarados como *public*) através do nome da classe.

## 5.3 Métodos estáticos em classe

Métodos estáticos em uma classe também são declarados com o modificador *static*, que deve preceder o tipo de retorno do método e que pode ser combinado com modificadores de acesso ao método. A diferença principal entre métodos estáticos (também conhecidos como **métodos de classes**) e métodos não estáticos é que os métodos estáticos podem ser chamados sem a necessidade de criação de instâncias das classes às quais pertencem.

Já vimos anteriormente um método estático – o método *main* – que permite que uma classe seja executada como uma aplicação ou programa. Se uma classe, por exemplo, Teste tem o método estático *main* declarado adequadamente, não se precisa criar instâncias de Teste para executar o método *main*. Métodos estáticos podem ser usados em classes que tenham o método *main* para servirem como sub-rotinas deste, o que será descrito na seção 5.4.

Métodos estáticos são também adequados para implementar rotinas que sejam independentes de dados armazenados em classes, ou seja, métodos que só necessitem dos dados passados como argumentos para efetuar a tarefa requerida, e que sejam executados com o mesmo resultado, indepen-

dentemente de qual instância da classe a que pertencem seja usada para sua chamada. A aplicação mais frequente de métodos estáticos é a criação de **bibliotecas de métodos**, classes que contêm somente métodos estáticos, geralmente agrupados por função. Um exemplo de uma biblioteca de métodos (que tem métodos para calcular várias conversões de unidades de comprimento) é mostrado no próximo exemplo de código.

```
1 /**
2 * A classe ConversaoDeUnidadesDeComprimento, que contém vários métodos estáticos que
3 * convertem unidades de comprimento. Esta classe não tem campos, e todos os seus
4 * métodos são declarados como static, assim não é necessário criar instâncias desta
5 * classe para usar os métodos. Para que os métodos possam ser acessados de qualquer
6 * outra classe eles são declarados como public.
7 */
8 class ConversaoDeUnidadesDeComprimento // declaração da classe
9 {
10 /**
11 * O método polegadasParaCentímetros converte o valor passado em polegadas para
12 * centímetros.
13 * @param polegadas o número de polegadas
14 * @return o número de centímetros correspondente ao número de polegadas
15 */
16 public static double polegadasParaCentímetros(double polegadas)
17 {
18 double centímetros = polegadas*2.54;
19 return centímetros;
20 }
21
22 /**
23 * O método pésParaCentímetros converte o valor passado em pés para centímetros.
24 * @param pés o número de pés
25 * @return o número de centímetros correspondente ao número de pés
26 */
```

**Figura 5.6a: Classe ConversãoDeUnidadeDeComprimento**
Fonte: Elaborada pelos autores

```
27 public static double pésParaCentímetros(double pés)
28 {
29 double centímetros = pés*30.48;
30 return centímetros;
31 }
32
33 /**
34 * O método milhasParaQuilômetros converte o valor passado em milhas para quilômetros.
35 * @param milhas o número de milhas
36 * @return o número de quilômetros correspondente ao número de milhas
37 */
38 public static double milhasParaQuilômetros(double milhas)
39 {
40 double quilômetros = milhas*1.609;
41 return quilômetros;
42 }
43
44 } // fim da classe ConversaoDeUnidadesDeComprimento
45
46 // Fórmulas obtidas no livro Manual de Fórmulas e Tabelas Matemáticas, Murray R.
47 // Spiegel, Coleção Schaum, editora McGraw-Hill
```

**Figura 5.6b: Classe ConversãoDeUnidadeDeComprimento – continuação**
Fonte: Elaborada pelos autores

Os métodos na classe ConversaoDeUnidadesDeComprimento são demonstrados na aplicação abaixo (Figura 5.7).

```
1 /**
2 * A classe DemoConversaoDeUnidadesDeComprimento, que demonstra usos dos métodos
3 * estáticos da classe ConversaoDeUnidadesDeComprimento.
4 */
5 class DemoConversaoDeUnidadesDeComprimento // declaração da classe
6 {
7 /**
8 * O método main permite a execução desta classe. Este método demonstra o uso de
9 * métodos estáticos da classe ConversaoDeUnidadesDeComprimento, através da criação
10 * de instâncias e através do acesso direto.
11 * @param argumentos os argumentos que podem ser passados para o método via linha
12 * de comando, mas que neste caso serão ignorados.
13 */
14 public static void main(String[] argumentos)
15 {
16 // Criamos uma instância da classe ConversaoDeUnidadesDeComprimento. Como a classe
17 // não contém campos e os métodos são estáticos, não existe real diferença entre
18 // chamar os métodos de uma ou outra instância da classe.
```

**Figura 5.7a: Métodos da classe ConversaoDeUnidadeDeComprimento**
Fonte: Elaborada pelos autores

```
19 ConversaoDeUnidadesDeComprimento conv = new ConversaoDeUnidadesDeComprimento();
20 // Vamos converter alguns valores:
21 System.out.println("vinte pés são "+conv.pésParaCentímetros(20)+
22 " centímetros");
23 System.out.println("cinco polegadas são "+conv.polegadasParaCentímetros(5)+
24 " centímetros");
25 // É muito mais prático acessar os métodos diretamente a partir da classe:
26 System.out.println("vinte pés são "+
27 ConversaoDeUnidadesDeComprimento.pésParaCentímetros(20)+
28 " centímetros");
29 System.out.println("cinco polegadas são "+
30 ConversaoDeUnidadesDeComprimento.polegadasParaCentímetros(5)+
31 " centímetros");
32 } // fim do método main
33
34 } // fim da classe DemoConversaoDeUnidadesDeComprimento
```

**Figura 5.7b: Métodos da classe ConversaoDeUnidadeDeComprimento – continuação**
Fonte: Elaborada pelos autores

Podemos  ver na aplicação mostrada que não é necessário criar uma instância da classe ConversaoDeUnidadesDeComprimento para acessar seus métodos, embora isso seja possível. Curiosamente, a instância *conv*, declarada na linha 20, poderia ser inicializada com o valor *null*, e o programa ainda seria executado normalmente – o que importa nesse caso é que a instância é da classe ConversaoDeUnidadesDeComprimento, independentemente de ser inicializada ou não.

Variáveis que forem declaradas dentro de métodos estáticos serão automaticamente estáticas, mas como seu escopo é somente dentro do método, elas não poderão ser acessadas a partir de outros métodos.

## 5.4 Campos e métodos estáticos em aplicações

Conforme explicado em aulas anteriores, o método *main*, quando declarado apropriadamente, será o ponto de entrada que permite a execução de uma classe. O método *main* deve ser declarado como estático para que não seja necessário instanciar a classe a que pertence.

Eventualmente será interessante que o método *main* possa chamar um outro método que não faça parte de outra classe, mas esteja contido na mesma (um *método local*, mais conhecido como *sub-rotina*), geralmente para simplificar o código (embora seja aconselhável, sempre que seja possível, agrupar vários métodos com características em comum em uma classe, que funcionará como uma biblioteca de rotinas).

Se um método for chamado diretamente a partir do método *main*, esse método deverá ser obrigatoriamente declarado como estático. Se o método *main* for acessar campos declarados na sua classe, mas fora do método *main*, esses campos também deverão ser declarados como estáticos.

Partes do código que podem ser executadas repetidamente são candidatas à criação de métodos locais, como mostrado na aplicação abaixo (Figura 5.8).

```
1 /**
2 * A classe CalculoDePrecoDeTerreno, que calcula o preço de um terreno baseado
3 * em sua área e localização. O cálculo é feito por um método estático da classe,
4 * permitindo o seu reuso.
5 */
6 class CalculoDePrecoDeTerreno // declaração da classe
7 {
8 /**
9 * O método main permite a execução desta classe. Este método demonstra o uso de um
10 * método estático para calcular o preço de um terreno baseado na sua área e
11 * localização. O cálculo é feito usando um método estático nesta classe (subrotina).
12 * @param argumentos os argumentos que podem ser passados para o método via linha
13 * de comando, mas que neste caso serão ignorados.
14 */
15 public static void main(String[] argumentos)
16 {
17 double preço;
18 // Cálculo do preço do terreno no lote N1
19 System.out.print("O preço do terreno N1 é ");
20 preço = preçoDoTerreno(450,1); // calculamos o preço e o armazenamos na variável
21 System.out.println(preço); // imprimimos a variável
22 // Cálculo do preço do terreno no lote Q2
23 System.out.print("O preço do terreno Q2 é ");
24 preço = preçoDoTerreno(475,4);
25 System.out.println(preço);
```

**Figura 5.8a: Classe CalculoDePrecoDeTerreno**
Fonte: Elaborada pelos autores

```
26 // Cálculo do preço do terreno no lote F3
27 System.out.print("O preço do terreno F3 é ");
28 // Chamamos o método e imprimimos o seu resultado
29 System.out.println(preçoDoTerreno(525,2));
30 } // fim do método main
31
32 /**
33 * O método preçoDoTerreno calcula o preço de um terreno dependendo da sua área
34 * em metros quadrados e sua localização, que é um código entre 1 e 5.
35 * @param área a área do terreno em metros quadrados
36 * @param localização o código da localização do terreno (1 a 5)
37 * @return o preço do terreno
38 */
39 private static double preçoDoTerreno(double área,int localização)
40 {
41 double preço = 0; // deve ser inicializada com algum valor !
42 if (localização == 1) preço = área*22.00;
43 if (localização == 2) preço = área*27.00;
44 if (localização == 3) preço = área*29.50;
45 if (localização == 4) preço = área*31.50;
46 if (localização == 5) preço = área*34.50;
47 return preço;
48 } // fim do método preçoDoTerreno
49
50 } // fim da classe CalculoDePrecoDeTerreno
```

**Figura 5.8b: Classe CalculoDePrecoDeTerreno – continuação**
Fonte: Elaborada pelos autores

Curiosamente, o próprio método *main* pode ser chamado a partir de outros métodos, inclusive de métodos *main* de outras classes. Dessa forma, uma aplicação inteira (que seria executada pelo método *main* de uma classe) pode ser considerada como uma sub-rotina de outra aplicação.

O código abaixo mostra como métodos estáticos (*main*e outros) de outras classes podem ser chamados a partir do método *main*.

```
1 /**
2 * A classe DemoChamadaAoMain, que mostra como métodos estáticos (main e outros) de
3 * outras classes podem ser chamados a partir do método main de uma classe.
4 */
5 class DemoChamadaAoMain // declaração da classe
6 {
7 /**
8 * O método main permite a execução desta classe. Este método demonstra como outros
9 * métodos main, de outras classes, podem ser chamados a partir deste.
10 * @param argumentos os argumentos que podem ser passados para o método via linha
11 * de comando, mas que neste caso serão ignorados.
12 */
```

**Figura 5.9a: Métodos estáticos de outras classes chamados a partir do método *main***
Fonte: Elaborada pelos autores

```
13 public static void main(String[] argumentos)
14 {
15 // Executamos o método main da classe DemoConstantesMatematicas como se fosse uma
16 // subrotina. Como todos os métodos main esperam um array de Strings como
17 // argumento, podemos simplesmente repassar os argumentos deste método main para
18 // o da classe DemoConstantesMatematicas.
19 DemoConstantesMatematicas.main(argumentos);
20 // Executamos o método main da classe DemoConversaoDeUnidadesDeComprimento como
21 // se fosse uma subrotina.
22 DemoConversaoDeUnidadesDeComprimento.main(argumentos);
23 // Para demonstrar a chamada de métodos estáticos em outras classes, vamos
24 // calcular o preço de um terreno:
25 System.out.print("O preço do terreno J1 é ");
26 // Chamamos o método e imprimimos o seu resultado
27 System.out.println(CalculoDePrecoDeTerreno.preçoDoTerreno(600,5));
28 } // fim do método main
29
30 } // fim da classe DemoChamadaAoMain
```

**Figura 5.9b: Métodos estáticos de outras classes chamados a partir do método *main* – continuação**
Fonte: Elaborada pelos autores

## 5.5 Fábricas de instâncias

Métodos estáticos que retornem novas instâncias de classes são conhecidos como **fábricas de instâncias**.

Fábricas de instâncias são úteis para a criação simples e rápida de instâncias que sejam bem características de uma classe.

Nem sempre a criação dessas fábricas é justificável. É interessante notar que podemos ter fábricas de instâncias de classes declaradas em outras classes.

```
1 /**
2 * A classe DataComFabrica, que contém uma fábrica de instâncias da própria classe.
3 * Esta classe contém campos que representam uma data e métodos simples para a
4 * manipulação destes campos. Esta classe contém também o construtor (que não verifica
5 * a validade dos dados) e o método toString.
6 */
7 class DataComFabrica // declaração da classe
8 {
9 /**
10 * Declaração dos campos da classe
11 */
12 private byte dia,mês; // dia e mês são representados por bytes
13 private short ano; // ano é representado por um short
14 /**
15 * O construtor da classe DataComFabrica recebe argumentos para inicializar os campos
16 * da classe. Este construtor não verifica a validade da data para manter a classe
17 * simples.
18 * @param d o argumento correspondente ao método dia
19 * @param m o argumento correspondente ao método mês
20 * @param a o argumento correspondente ao método ano
21 */
```

**Figura 5.10a: Classe DataComFabrica**
Fonte: Elaborada pelos autores

```
22 DataComFabrica(byte d,byte m,short a)
23 {
24 dia = d; mês = m; ano = a;
25 } // fim do construtor
26 /**
27 * O método toString retorna uma String contendo os valores dos campos formatados
28 * @return uma String com a data formatada
29 */
30 public String toString()
31 {
32 return dia+"/"+mês+"/"+ano;
33 } // fim do método toString
34 /**
35 * O método Natal (uma fábrica de instâncias da classe Data) retorna o dia de Natal
36 * para o ano passado como argumento.
37 * @param ano o ano para o qual retornaremos a data de Natal
38 * @return uma instância da classe Data correspondente ao Natal daquele ano
39 */
40 public static DataComFabrica Natal(short ano)
41 {
42 return new DataComFabrica((byte)25,(byte)12,ano);
43 } // fim do método Natal
44 } // fim da classe DataComFabrica
```

**Figura 5.10b: Classe DataComFabrica – continuação**
Fonte: Elaborada pelos autores

A classe DataComFabrica, mostrada no código acima, contém um método estático que retorna uma nova instância da própria classe contendo a data do Natal de um determinado ano, que deve ser passado como argumento para este método. A classe DemoDataComFabrica, mostrada no código abaixo (Figura 5.11), demonstra usos desse método.

```
1 /**
2 * A classe DemoDataComFabrica, que demonstra usos da classe DataComFabrica.
3 */
4 class DemoDataComFabrica // declaração da classe
5 {
6 /**
7 * O método main permite a execução desta classe. Este método cria algumas
8 * instâncias da classe DataComFabrica usando a fábrica de instâncias desta
9 * classe.
10 * @param argumentos os argumentos que podem ser passados para o método via linha
11 * de comando, mas que neste caso serão ignorados.
12 */
13 public static void main(String[] argumentos)
14 {
15 // Declaramos algumas referências à instâncias da classe DataComFabrica que serão
16 // inicializadas através do método Natal (a fábrica de instâncias).
17 DataComFabrica NatalDe1966 = DataComFabrica.Natal((short)1966);
18 DataComFabrica NatalDe1970 = DataComFabrica.Natal((short)1970);
19 DataComFabrica NatalDe2000 = DataComFabrica.Natal((short)2000);
20 // Imprimimos as datas através da chamada implícita ao método toString
21 System.out.println(NatalDe1966);
22 System.out.println(NatalDe1970);
23 System.out.println(NatalDe2000);
24 } // fim do método main
25
26 } // fim da classe DemoDataComFabrica
```

**Figura 5.11: Classe DemoDataComFabrica**
Fonte: Elaborada pelos autores

Podemos ver que a criação de instâncias da classe DataComFabrica pode ser feita de maneira simples com a fábrica de instâncias.

## Resumo

Nesta aula vimos como podemos declarar campos que serão compartilhados entre todas as instâncias de uma mesma classe em uma aplicação, assim como a criação de métodos que não precisam de instâncias de classes para ser executados.

Campos estáticos em uma classe são compartilhados por todas as instâncias dessa classe; em outras palavras, somente um valor será armazenado em um campo estático, e caso esse valor seja modificado por uma das instâncias dessa classe, a modificação será refletida em todas as outras instâncias dessa classe.

Campos estáticos são declarados com o modificador *static*, que deve ser declarado antes do tipo de dado do campo e pode ser combinado com modificadores de acesso como *public* e *private*.

Campos estáticos são também conhecidos como campos de classes, já que estes campos poderão ser acessados diretamente usando o nome da classe.

Métodos estáticos em uma classe também são declarados com o modificador *static*, que deve preceder o tipo de retorno do método e que pode ser combinado com modificadores de acesso ao método. A diferença principal entre métodos estáticos (também conhecidos como métodos de classes) e métodos não estáticos é que os métodos estáticos podem ser chamados sem a necessidade de criação de instâncias das classes às quais pertencem.

O método *main*, quando declarado apropriadamente, será o ponto de entrada que permite a execução de uma classe. O método *main* deve ser declarado como estático para que não seja necessário instanciar a classe a que pertence.

Métodos estáticos que retornem novas instâncias de classes são conhecidos como fábricas de instâncias. Fábricas de instâncias são úteis para a criação simples e rápida de instâncias que sejam bem características de uma classe.

## Atividades de aprendizagem

1. Escreva, para a classe DataComFabrica (Figuras 5.10), um método primeiroDeAbril que se comporte como uma fábrica de instâncias.

2. O método *main* pode ser chamado a partir de outro método da mesma classe. Se isso for feito, que problemas podem ocorrer na aplicação ?

3. Escreva a classe ConversaoDeUnidadesDeArea com métodos estáticos para conversão das unidades de área segundo a lista abaixo.

   1 metro quadrado = 10.76 pés quadrados
   1 pé quadrado = 929 centímetros quadrados
   1 milha quadrada = 640 acres
   1 acre = 43.560 pés quadrados

4. Escreva uma classe que contenha métodos estáticos para calcular as médias de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos *int* e *double*.

5. Escreva uma classe que contenha métodos estáticos para calcular as somas de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos *int* e *double*.

6. Escreva uma classe que contenha métodos estáticos para retornar o maior de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos *int* e *double*. Dica: os métodos podem ser chamados em cascata: para calcular o maior de três valores a, b e c, pode-se calcular o maior valor de a e b, e comparar este resultado com c.

7. Escreva uma classe SerieLimitada, que encapsula um valor inteiro sequencial como os usados em notas e séries de gravuras. Essa classe deve permitir que um programa crie um número limitado de instâncias dela, cada uma numerada com um valor sequencial. O número total de instâncias é controlado pelo campo maximoDeInstancias, declarado como *static* final, e o de instâncias já criadas é controlado pelo campo contador, declarado como *static*. Escreva também uma aplicação que crie algumas instâncias da classe para demonstrar seu funcionamento.

Poste suas respostas no AVEA.

# Aula 6 – Reutilização de classes

## Objetivos

Compreender e utilizar os principais meios de reutilização de classes.

Saber utilizar o conceito de herança, polimorfismo, entre outros, para que se possa reutilizar esses códigos.

Aprender os dois tipos de polimorfismo e saber como aplicá-los no código.

Aprender como criar objetos de uma subclasse.

Utilizar o método final e a classe final.

## 6.1 Introdução

Uma das características mais interessantes de linguagens de programação orientadas a objetos é a capacidade de facilitar a reutilização de código – o aproveitamento de classes e seus métodos que já estejam escritos e que já tenham o seu funcionamento testado e comprovado. A reutilização de código diminui a necessidade de escrever novos métodos e classes, economizando o trabalho do programador e diminuindo a possibilidade de erros.

Linguagens procedurais podem implementar reutilização de código com funções que podem ser chamadas a partir de vários programas diferentes. Java e outras linguagens de programação orientadas a objetos vão mais além, permitindo a criação de classes baseadas em outras. As classes criadas com essa técnica poderão conter os métodos das classes originais, além de poder adicionar comportamento específico da nova classe.

## 6.2 Herança

A herança é a principal característica de distinção entre um sistema de programação orientado a objeto e outros sistemas de programação. As classes são inseridas em uma hierarquia de especializações de tal forma que uma

classe mais especializada herda todas as propriedades da classe mais geral a qual é subordinada na hierarquia. A classe mais geral é denominada **superclasse**, e a classe mais especializada, **subclasse**.

O principal benefício da herança é a reutilização de código. A herança permite ao programador criar uma nova classe programando somente as diferenças existentes na subclasse em relação à superclasse. Isso se ajusta bem à forma como compreendemos o mundo real, no qual conseguimos identificar naturalmente essas relações.

A fim de exemplificarmos este conceito, vamos considerar que queiramos modelar os seres vivos pluricelulares existentes no planeta. Podemos então começar com a classe SerVivo abaixo (Figura 6.1).

**Figura 6.1: Classe SerVivo**
Fonte: Elaborada pelos autores

Ela modela as características que todo ser vivo deve possuir, como a capacidade de reproduzir-se ou a necessidade de alimentar-se. Sendo assim, a classe SerVivo define atributos e métodos tais como:

- **Atributos:** Alimentos, Idade.
- **Métodos:** Nascer, Alimentar, Respirar, Crescer, Reproduzir, Morrer.

Os seres vivos, por sua vez, classificam-se em Animais e Vegetais, os quais possuem características próprias que os distingue (Figura 6.2).

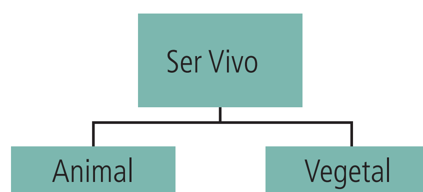



**Figura 6.2: Classificação de SerVivo**
Fonte: Elaborada pelos autores

Analisando o problema em questão (o de modelar os seres vivos), nós naturalmente identificamos classes que são especializações de classes mais genéricas, e o conceito de herança da orientação a objeto nos permite implementar tal situação.

Os animais e vegetais, antes de tudo, são seres vivos e cada subclasse herda automaticamente os atributos e métodos (respeitando as regras dos modificadores de acesso) da superclasse, neste caso, a classe SerVivo. Além disso, as subclasses podem prover atributos e métodos adicionais para representar suas próprias características. Por exemplo, a classe Animal poderia definir os seguintes métodos e atributos:

- **Atributos:** Forma de Locomoção, *Habitat*, Tempo Médio de Vida.

- **Métodos:** Locomover.

A herança na programação é obtida especificando-se qual superclasse a subclasse estende. Em Java isto é feito utilizando-se a palavra chave **extends**:

```
public class SerVivo {
//Definição da classe SerVivo
}
public class Animal extends SerVivo {
//Atributos e métodos adicionais que distinguem um Animal de um
SerVivo
//qualquer
}
```

Em Java, todas as classes, tanto as existentes nas APIs como as definidas pelos programadores, automaticamente derivam de uma superclasse padrão: a classe ***Object***.

Se uma classe não especifica explicitamente uma superclasse, como o caso da classe SerVivo, então podemos considerar que esta deriva diretamente de *Object*, como se ela tivesse sido definida como:

```
public class SerVivo extends Object { ... }
```

Além disso, Java permite apenas **herança simples**, isto é, uma classe pode estender apenas de uma **única** outra classe.

Resumindo, a classe SerVivo define os atributos e métodos que são comuns a qualquer tipo de ser vivo. A subclasse Animal herda esses métodos e atributos, já que Animal é um SerVivo , e tem que especificar apenas seus atributos e métodos específicos (Figura 6.3).

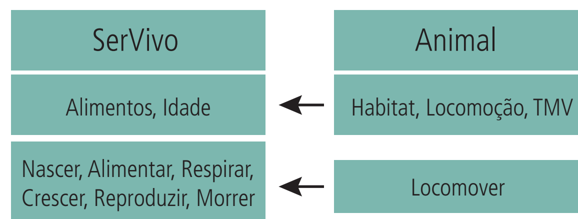



**Figura 6.3: Herança simples**
Fonte: Elaborada pelos autores

## 6.3 Polimorfismo

O termo **polimorfismo** origina-se do grego e quer dizer "o que possui várias formas". Em programação está relacionado à possibilidade de se usar o mesmo nome para métodos diferentes e à capacidade que o programa tem em discernir, dentre os métodos homônimos, aquele que deve ser executado. De maneira geral o polimorfismo permite a criação de programas mais claros, pois elimina a necessidade de darmos nomes diferentes para métodos que conceitualmente fazem a mesma coisa, e também programas mais flexíveis, dos quais facilita em muito a extensão.

O polimorfismo pode ser de duas formas; **estático** ou **dinâmico**:

- **Polimorfismo estático:** ocorre quando na definição de uma classe criamos métodos com o mesmo nome, porém com argumentos diferentes. Dizemos nesse caso que o método está **sobrecarregado** (*overloading*). A decisão de qual método chamar é tomada em tempo de compilação, baseada nos argumentos que foram passados.

Como exemplo de polimorfismo estático, podemos relembrar dos vários construtores que criamos para uma classe Círculo:

```
Círculo() {}
Círculo(Ponto centro, Ponto extremidade, Cor c) {
  raio = new Linha(centro, extremidade, c);
  cor = c;
}
Círculo (double x1, double y1, double x2, double y2, Cor c)
{
  raio = new Linha(new Ponto(x1, y1), new Ponto(x2, y2), c);
  cor = c ;
}
```

Todos esses métodos construtores possuem o mesmo nome, mas devem ser diferidos entre si pelos parâmetros que recebem. Quando num programa fazemos Círculo circulo = **new** Círculo(linha); o compilador consegue decidir em tempo de compilação qual método chamar, nesse caso, o método Círculo que recebe uma linha como parâmetro.

- **Polimorfismo dinâmico:** está associado com o conceito de herança e ocorre quando uma subclasse redefine um método existente na superclasse. Dizemos nesse caso que o método foi **sobrescrito** (*overriding*) na subclasse. A decisão de qual método executar é tomada somente em tempo de execução, como veremos mais adiante.

O polimorfismo dinâmico ocorre quando uma subclasse redefine um método de sua superclasse a fim de prover a ele um comportamento mais adequado às suas características. Vamos rever a classe animal conforme a definimos acima:

```
class Animal extends SerVivo{
  String formaLocomoção;
  String habitat;
  int tempoMedioVida;
public void locomover() { ... }
}
```

Como nem todo ser vivo nasce, cresce, alimenta-se, respira, se reproduz e morre da mesma maneira, é razoável que queiramos redefinir todos esses métodos na classe Animal.

```
class Animal extends SerVivo{
String formaLocomocao;
String habitat;
int tempoMedioVida;
public void locomover() { ... }
public void nascer() { ... }
public void crescer() { ... }
public void alimentar() { ... }
public void respirar() { ... }
public Animal reproduzir() { ... }
public morrer() { ... }
}
```

Os métodos na subclasse devem ser definidos com a mesma "**assinatura**" do método na superclasse, isto é, com o mesmo nome, tipo de retorno e argumentos.

Bem, mas a vantagem do polimorfismo dinâmico não é apenas a de permitir maior flexibilidade na modelagem das classes de objetos. Para entendermos o que há de mais fantástico nele, temos que nos atentar para o seguinte:

```
...
SerVivo x;  → Declara uma variável x do tipo Ser Vivo
Animal y = new Animal(); → Declara e inicializa uma varoável y do tipo Animal
x = y;
...
```

É extremamente comum e útil em OO atribuirmos a uma variável de um tipo base um objeto de um tipo derivado direta ou indiretamente deste tipo base.

Tendo isso em mente, poderíamos definir um método da seguinte forma:

```
void analisaSerVivo (SerVivo ser) {
//Este método faz a análise clínica de qualquer SerVivo e para isso precisa pedir
//ao animal que respire
ser.respirar();
. . .
}
```

e num determinado momento chamá-lo desta maneira:

```
. . .
Animal animal = new Animal();
analisaSerVivo(animal);
. . .
```

Nesse caso fica a questão: se o método tem um argumento declarado como SerVivo e recebe como parâmetro um objeto Animal, quando tiver que executar o método respirar(), qual método será efetivamente chamado, o método respirar definido em SerVivo ou o método respirar  definido em Animal?

O método executado será o mais apropriado, isto é, aquele pertencente ao objeto que foi passado à variável, nesse caso, o método respirar existente em Animal.

Isto é o polimorfismo dinâmico, que recebe este nome porque o compilador não consegue nesses casos decidir em tempo de compilação qual método chamar, já que uma variável de um tipo base pode receber qualquer tipo derivado. Essa decisão é tomada apenas em tempo de execução, quando aí o Java poderá saber qual objeto a variável está de fato referenciando.

Se o polimorfismo dinâmico não existisse, seria necessário um método analisaSerVivo para cada ser vivo existente e, sempre que um ser vivo novo fosse acrescentado, o código teria que ser modificado. Da forma como aqui está, o método continuará funcionando para qualquer ser vivo existente no projeto.

## 6.4 Criação de um objeto de uma subclasse e o ponteiro *super*

Objetos são sempre construídos da classe mais alta na hierarquia em direção à mais baixa, isto é, da classe **Object** até a classe que está sendo instanciada com o operador **new**. Isso garante que uma subclasse pode sempre contar com a construção apropriada de sua superclasse.

Por padrão, são chamados construtores *default* das superclasses (aqueles sem argumentos) quando a superclasse é criada. Se quisermos mudar esse comportamento, devemos usar o ponteiro ***super***.

O ponteiro *super* é uma referência para a superclasse e está disponível implicitamente em qualquer classe que possua um ancestral. Como vimos, os métodos na superclasse podem ser sobrescritos na subclasse através da criação, nesta, de métodos com a mesma assinatura do método da superclasse. Com o *super* é possível acessar um método específico da superclasse quando este foi sobrescrito pela subclasse. Por exemplo, vamos recordar a classe Animal definida acima, que implementava sua própria versão de todos os métodos de sua superclasse, isto é, SerVivo:

```
class Animal extends SerVivo{
String formaLocomoção;
String habitat;
int tempoMedioVida;
public void locomover() { ... }
public void nascer() { ... }
public void crescer() { ... }
public void alimentar() { ... }
public void respirar() { ... }
public Animal reproduzir() { ... }
public morrer() { ... }
}
```

Vamos supor que num determinado momento quiséssemos acessar o método alimentasse tal qual como foi implementado na superclasse. Para isso devemos usar o ponteiro *super* da seguinte maneira:

```
...
super.alimentar();    // Acessa o método alimentar-se de SerVivo.
...
```

Bem, mas como afinal de contas podemos mudar o comportamento padrão de quando uma subclasse é instanciada para os construtores *default* serem executados?

Um dos usos mais comuns da referência *super* é chamar um construtor de uma superclasse. Para isso, este deve ser acrescentado na **primeira linha** do construtor da subclasse. Por exemplo, se a classe SerVivo tiver um construtor que receba a idade do ser vivo como parâmetro

```
SerVivo (int idade){
        this.idade = idade;
}
```

e queremos que este construtor seja chamado quando a classe Animal for instanciada ao invés do construtor padrão SerVivo(), fazemos

```
Animal (int idade){
   super (idade);
   ...
}
```

## 6.5 Método final e classe final

- **Método final:** método definido como final não pode ser sobrescrito. Geralmente isso é feito com o intuito de garantir segurança, prevenindo que métodos vitais, tais como aqueles que fazem autenticação de usuários, sejam redefinidos.

Exemplo:

```
public final boolean checkPassword(String p) {
. . .
}
```

- **Classe final:** uma classe declarada como final não poderá ser estendida por outra classe. Esta é uma forte decisão no projeto e significa que a classe é autossuficiente para dispor de todos os seus requisitos atuais e futuros. Classes definidas como final podem ajudar o compilador a produzir códigos mais otimizados. Como uma classe final não pode ser estendida, quando o compilador encontra uma chamada a algum método dessa classe ele não gerará o código necessário para execução do polimorfismo dinâmico, o qual acarreta um *overhead* durante a execução, mas sim fará um acoplamento estático com o método em questão.

## Resumo

Nesta aula vimos os principais conceitos que diferenciam a linguagem orientada a objetos das demais linguagens de programação (herança, polimorfismo e outros). Vimos que o principal benefício da herança é a reutilização de código. A herança permite ao programador criar uma nova classe programando somente as diferenças existentes na subclasse em relação à superclasse. Isso se ajusta bem à forma como compreendemos o mundo real, no qual conseguimos identificar naturalmente essas relações.

Pudemos compreender que o termo polimorfismo origina-se do grego e quer dizer "o que possui várias formas". Em programação está relacionado à possibilidade de se usar o mesmo nome para métodos diferentes e à capacidade que o programa tem em discernir, dentre os métodos homônimos, aquele que deve ser executado. Vimos também os dois tipos de polimorfismo: estático e dinâmico.

Aprendemos ainda como criar objetos de subclasse e como utilizar o método final e a classe final.

## Atividades de aprendizagem

1. Criar uma estrutura hierárquica que contenha as seguintes classes: Veículo (classe abstrata), Bicicleta e Automóvel. Os métodos da classe Veículo são todos abstratos e possuem a seguinte assinatura:

   - listarVerificacoes()
   - ajustar()
   - limpar()

   Estes métodos são implementados nas subclasses Automóvel e Bicicleta. Acrescentar na classe Automóvel o método mudarOleo()

2. Seria possível evitar completamente a necessidade de sobreposição de métodos criando métodos em classes descendentes que tenham assinaturas diferentes. Por exemplo, a classe Pessoa poderia ter o método imprimePessoa para imprimir seus campos, e a classe Aluno, que estende a classe Pessoa, poderia ter o método imprimeAluno para imprimir seus campos. Que vantagens e desvantagens esta abordagem teria sobre a sobreposição de métodos ?

3. Explique, com suas palavras, por que construtores de superclasses não são herdados por subclasses.

4. Crie uma classe Equipamento com dois atributos privados.

5. Crie uma classe Computador com dois atributos à sua escolha. A classe Computador deverá herdar tudo da classe Equipamento.

6. Crie os métodos de acesso e de modificação para todos os atributos definidos em ambas as classes.

7. Crie uma classe TestaEquipamento, com o método *main,* crie nela um objeto da classe Equipamento e instancie os dois atributos que você declarou na classe Equipamento. Crie também um objeto da classe Computador, utilizando os dois atributos declarados na própria classe e os dois herdados da classe Equipamento.

8. O método *main* deve exibir as informações dos dois objetos criados.

Poste no AVEA as soluções das questões.

# Referências

BARNES, David J.; KOLLING, Michael. **Programação orientada a objetos com Java:** uma introdução prática usando o BLUEJ. São Paulo: Makron Books, 2004.

BORATTI, Isaías Camilo. **Programação orientada a objetos em Java.** Florianópolis: Visual Books, 2007.

PRESMAN, Rogers. **Engenharia de software.** São Paulo: Makron Books, 1995.

SIERRA, Kathy; BATES, Bert. **Use a cabeça Java.** Rio de Janeiro: Alta Books, 2007.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java.** Rio de Janeiro: Campus, 2003.

ISBN 978-85-67082-01-1