

PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Cristiano Roberto Franco





Copyright © UNIASSELVI 2014

Elaboração:
Prof. Cristiano Roberto Franco

Revisão, Diagramação e Produção:
Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

005.1

F825p Franco, Cristiano Roberto

Programação orientada a objetos / Cristiano Roberto Franco
Indaial : Uniasselvi, 2014.

222 p. : il

ISBN 978-85-7830- 855-1

1. Programação orientada a objetos
I. Centro Universitário Leonardo da Vinci.



APRESENTAÇÃO

Prezado(a) acadêmico(a)!

Bem-vindo ao mundo da Programação Orientada a Objetos. Termos como atributos, métodos, encapsulamento, herança, polimorfismo, abstração, composição, *design patterns*, propriedades, sobreposição, sobrecarga, entre muitos outros, que farão parte dos seus estudos nesta disciplina e, consequentemente, de seu dia a dia como programador.

Ao avaliar os índices que mostram as linguagens de programação mais utilizadas em escala mundial, encontramos, há aproximadamente dez anos, uma maioria absoluta de linguagens que utilizam os conceitos de programação orientada a objetos, evidência clara da utilização deste paradigma em larga escala. Este fato, por si só, justifica o aprendizado deste paradigma por todos que pretendem fazer da programação de computadores sua profissão.

Este caderno pressupõe alguma experiência anterior em programação, não fazendo parte de nossos objetivos o ensino de programação básica. Pretendemos fazer com que você utilize os conhecimentos adquiridos nas disciplinas de algoritmos e programação básica e passe a utilizá-los em combinação com os conceitos da Orientação a Objetos na resolução de problemas. Isso não quer dizer que você precisa ser um *expert* em programação para acompanhar o conteúdo deste caderno. O embasamento construído nas disciplinas anteriores, em combinação com materiais de referência da linguagem de programação que utilizaremos nos exercícios (Java) deve bastar.

Aproveitamos esse momento para destacar que os exercícios NÃO SÃO OPCIONAIS. O objetivo de cada exercício deste caderno é a fixação de determinado conceito através da prática. É aí que reside a importância da realização de todos. Sugerimos fortemente que em caso de dúvida em algum exercício você entre em contato com seu Tutor Externo ou com a tutoria da Uniasselvi e que não passe para o exercício seguinte enquanto o atual não estiver completamente compreendido. Aprender a programar objetos de forma orientada sem exercitar os conceitos de forma prática, somente com a leitura do caderno, é equivalente a ir a um restaurante e tentar matar a fome lendo o cardápio. Instruções detalhadas de como preparar o ambiente para a resolução dos exercícios em seu computador são mostrados na Unidade 1.

Você precisará de muita determinação e força de vontade, pois a Programação Orientada a Objetos não é fácil e você levará mais do que um semestre para compreendê-la totalmente, entretanto, aqui forneceremos um início sólido e consistente. Desta forma, passe por esse período de estudos com muita dedicação, pois você que hoje é acadêmico(a), amanhã será um profissional de Tecnologia da Informação com o compromisso de construir uma nação melhor.

Lembre-se que o encantamento com a programação deriva do seu entendimento; a beleza desta área do conhecimento é a compreensão da lógica envolvida na construção de programas. Por isso, bons programadores são apaixonados por linguagens de programação e ambientes de desenvolvimento e estão sempre buscando novos conhecimentos. Como disse Drew Houston, criador do *DropBox*: “Programar é a coisa mais parecida que temos com superpoderes”.

Bons códigos!

Prof. Cristiano Roberto Franco



Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, tablet ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você me verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!



Olá acadêmico! Para melhorar a qualidade dos materiais ofertados a você e dinamizar ainda mais os seus estudos, a Uniasselvi disponibiliza materiais que possuem o código QR Code, que é um código que permite que você acesse um conteúdo interativo relacionado ao tema que você está estudando. Para utilizar essa ferramenta, acesse as lojas de aplicativos e baixe um leitor de QR Code. Depois, é só aproveitar mais essa facilidade para aprimorar seus estudos!



BATE SOBRE O PAPO ENADE!



Olá, acadêmico!



Você já ouviu falar sobre o ENADE?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.



Vamos lá!



Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.



Que prova é essa?

É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O MEC – Ministério da Educação.



O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.

Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.

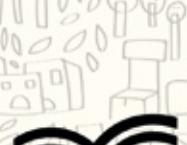


Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.

Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.



Você tem uma trilha de aprendizagem do ENADE, receberá e-mails, SMS, seu tutor e os profissionais do polo também estarão orientados.



Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!



SUMÁRIO

UNIDADE - 1 INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS	1
TÓPICO 1 - CONTEXTO HISTÓRICO.....	3
1 INTRODUÇÃO	3
2 PRECURSORES DA PROGRAMAÇÃO ORIENTADA A OBJETOS.....	4
3 A NOÇÃO DE CLASSES E OBJETOS.....	5
4 VANTAGENS DA PROGRAMAÇÃO ORIENTADA A OBJETOS.....	10
5 LINGUAGENS DE PROGRAMAÇÃO ORIENTADA A OBJETOS	12
6 HISTÓRIA DA LINGUAGEM DE PROGRAMAÇÃO JAVA	14
7 CARACTERÍSTICAS DA PLATAFORMA JAVA	19
7.1 HOTSPOT E JIT	24
7.2 PLATAFORMA OU LINGUAGEM?.....	26
RESUMO DO TÓPICO 1.....	29
AUTOATIVIDADE	30
 TÓPICO 2 - FERRAMENTAS E INSTALAÇÃO.....	33
1 INTRODUÇÃO	33
2 JAVA DEVELOPMENT KIT E INTEGRATED DEVELOPMENT ENVIRONMENT	33
2.1 CRIANDO UM PROJETO	37
2.2 CRIANDO UMA CLASSE.....	42
RESUMO DO TÓPICO 2.....	47
AUTOATIVIDADE	48
 TÓPICO 3 - CLASSES E OBJETOS	49
1 INTRODUÇÃO	49
2 CRIANDO CLASSES	49
3 CRIANDO OBJETOS.....	53
LEITURA COMPLEMENTAR.....	58
RESUMO DO TÓPICO 3.....	64
AUTOATIVIDADE	66
 UNIDADE 2 - ENCAPSULAMENTO, HERANÇA E POLIMORFISMO	69
TÓPICO 1 - ENCAPSULAMENTO.....	71
1 INTRODUÇÃO	71
2 ACOPLAMENTO E COESÃO.....	74
3 IMPLEMENTAÇÃO EM JAVA.....	79
3.1 VISIBILIDADE ENTRE PACOTES	87
RESUMO DO TÓPICO 1.....	91
AUTOATIVIDADE	93

TÓPICO 2 - HERANÇA	95
1 INTRODUÇÃO	95
2 HERANÇA	95
2.1 MECÂNICA DA HERANÇA	99
2.2 DICAS PARA FAZER UMA HERANÇA EFICAZ	100
2.3 IMPLEMENTAÇÃO DA HERANÇA NA LINGUAGEM DE PROGRAMAÇÃO JAVA	101
2.4 CLASSES ABSTRATAS	108
2.5 COMPOSIÇÃO AO INVÉS DE HERANÇA	109
RESUMO DO TÓPICO 2.....	114
AUTOATIVIDADE	115
TÓPICO 3 - POLIMORFISMO	117
1 INTRODUÇÃO	117
2 POLIMORFISMO	117
2.1 POLIMORFISMO DE INCLUSÃO	121
2.2 POLIMORFISMO PARAMÉTRICO	122
2.3 SOBREPOSIÇÃO	124
2.4 SOBRECARGA	126
2.5 INTERFACES	127
2.6 POLIMORFISMO EFICAZ	133
LEITURA COMPLEMENTAR.....	133
RESUMO DO TÓPICO 3.....	137
AUTOATIVIDADE	139
UNIDADE 3 - TÓPICOS AVANÇADOS DA LINGUAGEM E BOAS PRÁTICAS	141
COLEÇÕES E SORT.....	143
1 INTRODUÇÃO	143
2 COLEÇÕES E SORT.....	144
2.1 ARRAYLIST	144
2.2 HASHMAP	148
2.3 HASHSET	150
2.4 IGUALDADE ENTRE OBJETOS	153
2.5 HASHCODE	159
2.6 ORDENAÇÃO	159
RESUMO DO TÓPICO 1.....	165
AUTOATIVIDADE	167
TÓPICO 2 - STATIC E TRATAMENTO DE EXCEÇÕES	169
1 INTRODUÇÃO	169
2 STATIC	169
2.1 SINGLETON	174
2.2 TRATAMENTO DE EXCEÇÕES	177
2.3 CRIANDO SUAS PRÓPRIAS EXCEÇÕES	184
RESUMO DO TÓPICO 2.....	187
AUTOATIVIDADE	189

TÓPICO 3 - JDBC E DAO	191
1 INTRODUÇÃO	191
2 MAPEAMENTO OBJETO RELACIONAL	192
3 ARQUITETURA DO DATA ACCESS OBJECT	194
4 JDBC	196
5 IMPLEMENTAÇÃO DO DATA ACCESS OBJECT (DAO)	198
LEITURA COMPLEMENTAR.....	210
RESUMO DO TÓPICO 3.....	217
AUTOATIVIDADE	218
REFERÊNCIAS.....	219

INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- reconhecer alguns dos principais paradigmas utilizados na programação de computadores;
- diferenciar e caracterizar classes e objetos;
- entender de que forma os objetos interagem entre si para a resolução de problemas;
- preparar um ambiente de desenvolvimento que permita a utilização de uma linguagem de programação Orientada a Objetos.

PLANO DE ESTUDOS

Esta unidade de ensino está dividida em três tópicos, sendo que no final de cada um deles, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – CONTEXTO HISTÓRICO

TÓPICO 2 – FERRAMENTAS E INSTALAÇÃO

TÓPICO 3 – CLASSES E OBJETOS



CONTEXTO HISTÓRICO

1 INTRODUÇÃO

Conforme Black (2012), os conceitos da programação orientada a objetos estão intrinsecamente ligados com os trabalhos pioneiros de Ole-Johan Dahl e Kristen Nygaard, no projeto da linguagem SIMULA, iniciados no Centro de Computação Norueguês em 1961. A linguagem de programação SIMULA introduziu a noção de classes, instâncias, subclasses, métodos virtuais e subrotinas como partes de um paradigma explícito de programação, além de já utilizar o conceito de *garbage collection*.

A linguagem de programação *Smalltalk*, desenvolvida no lendário Xerox PARC por um time liderado por Alan Kay na década de 70, introduziu o termo programação orientada a objetos para representar o uso de objetos e mensagens como base para a computação. Os criadores do *Smalltalk* foram essencialmente influenciados pelas ideias introduzidas na SIMULA, embora adicionassem características de outras linguagens, como tipagem dinâmica, por exemplo. É importante salientar que tanto SIMULA, quanto *Smalltalk* sofreram influência e influenciaram diversas outras linguagens de programação, mas essas questões não fazem parte do escopo deste Caderno de Estudos.

A programação orientada a objetos se estabeleceu como o paradigma dominante aproximadamente no início da década de 1990, quando linguagens de programação que implementavam seus conceitos, como por exemplo C++ e Delphi, estavam disponíveis em larga escala. Paralelamente ao crescimento da programação orientada a objetos entre os desenvolvedores, acontecia também um aumento no interesse dos usuários pelas GUIs (interfaces gráficas de usuário) dos sistemas operacionais e aplicativos.

Com o sucesso da programação orientada a objetos, suas características foram adicionadas a diversas linguagens de programação que inicialmente não haviam sido projetadas para tal, como Ada, Cobol, Fortran e Basic, levando invariavelmente a problemas de manutenção e compatibilidade de código (WIKIPEDIA, 2012).

Indiscutivelmente, o sucesso de linguagens de programação como o Java, desenvolvido inicialmente pela *Sun Microsystems* e hoje propriedade da *Oracle* e o C#, desenvolvido pela *Microsoft*, é devido em parte à sua aderência em maior ou menor grau aos princípios da programação orientada a objetos e à simplicidade que oferecem aos desenvolvedores na aplicação destes princípios.



Você sabia que o Xerox PARC é responsável por muitas das inovações tecnológicas que utilizamos em nossos computadores hoje em dia? Como exemplo, podemos citar o mouse, a interface gráfica do usuário, o protocolo Ethernet e as telas LCD. Maiores detalhes podem ser encontrados em: <<http://www.administradores.com.br/artigos/marketing/a-revolucao-da-inovacao-na-xerox/22829/>>.

2 PRECURSORES DA PROGRAMAÇÃO ORIENTADA A OBJETOS

Atualmente, quando utilizamos um computador, estamos tirando proveito de aproximadamente 50 anos de evolução e pesquisas. Antigamente, os programadores inseriam os programas diretamente na memória do computador, através de bancos de chaves chamados de *switches*. Estes programas eram escritos em linguagem binária, o que os tornava propensos a erros e extremamente complexos para manter (SINTES, 2002).

Quando os computadores se tornaram mais comuns, linguagens de programação mais simples, chamadas de alto nível começaram a aparecer, como, por exemplo, a linguagem de programação FORTRAN. O surgimento dessas linguagens foi motivado por diversos fatores, entre eles a necessidade de uma maneira mais simples de escrever programas e consequentemente qualificar programadores de forma mais rápida, uma vez que a linguagem binária levava anos para ser dominada.

Essas linguagens foram chamadas de procedurais, pois permitem que um programador reduza um programa inteiro em procedimentos menores, atacando a complexidade por partes. Esses procedimentos definem a estrutura global do programa que executa até que tenha chamado toda sua lista de procedimentos. O paradigma procedural apresentou diversas melhorias em relação à linguagem binária, facilitando o trabalho de entendimento, depuração e criação de programas.

Um dos problemas desse paradigma é sua concentração nos dados em detrimento dos comportamentos. Como os procedimentos operam separadamente sobre os dados, estes não podem ser protegidos ou encapsulados. A consequência é que os procedimentos precisam saber detalhes sobre os dados e, em caso de alteração em alguma estrutura, todos os procedimentos que a acessam provavelmente precisarão de manutenção (SINTES, 2002).

Sintes (2002) afirma ainda que a programação modular, presente em linguagens como Modula2, tentou resolver estas limitações dividindo os programas em componentes chamados de módulos, onde comportamento e dados estariam presentes. Quando outros módulos precisam interagir com um módulo específico, eles o fazem através da interface do módulo, diminuindo os problemas relacionados à manutenção do código fonte. Três deficiências impediram o paradigma modular de se tornar o paradigma dominante:

- 1) Os módulos não são extensíveis: quando alguma alteração em um módulo é necessária, obrigatoriamente você deve abri-lo e interagir com o código fonte já existente.
- 2) Não é possível basear um módulo em outro: caso dois módulos sejam semelhantes ou até mesmo idênticos, não é possível reutilizar o código já escrito, a não ser via cópia. O problema com essa abordagem, além do fato de ser mais trabalhosa, é que em caso de manutenção daquele código fonte o programador deverá lembrar todos os lugares que o utilizam e replicar a alteração.
- 3) O paradigma modular ainda herda aspectos do paradigma procedural, uma vez que os módulos são acionados via procedimentos.

Estas características do paradigma modular exigem que o programa que os utilize conheça exatamente o tipo do módulo com o qual está trabalhando, direcionando para um código repleto de instruções condicionais (*if*, *switch case*) e tornando complexo o processo de manutenção do código fonte.

3 A NOÇÃO DE CLASSES E OBJETOS

O paradigma modular resolveu diversos problemas do paradigma procedural, entretanto, ao utilizá-lo os programadores conseguiram identificar algumas deficiências que prejudicavam sua produtividade.

A programação orientada a objetos abordou estas deficiências adicionando os conceitos de herança e polimorfismo e retirando o aspecto procedural do controle do programa. Quem controla o fluxo de execução do programa são os próprios objetos, que se comunicam entre si através de mensagens.

Sintes (2002) define objeto como uma construção de *software* que encapsula estado e comportamento, permitindo que o programa seja modelado em termos reais e abstrações.

Já Bates e Sierra (2010) colocam que uma classe é o projeto de um objeto. Ela informa à máquina virtual como criar um objeto desse tipo específico. Cada objeto criado a partir dessa classe terá seus valores distintos para as variáveis de instância da classe.

Em suma, um objeto é um “pedaço” de um programa de computador que possui internamente um conjunto de valores e procedimentos para manipular estes valores. Um programa orientado a objetos é composto de diversos “pedaços” que se comunicam entre si e, em conjunto ou individualmente, resolvem problemas.

Para melhorar sua compreensão do conceito, faremos uma analogia entre um *software* desenvolvido utilizando programação orientada a objetos e alguma situação do mundo real. Imagine uma situação onde você está assistindo a algum programa de TV e tem ao seu alcance o controle remoto (Figura 1). Considere como objetos deste *software* a TV, o controle remoto e você mesmo. De repente você lembra que gostaria de assistir a outro canal e para resolver este problema, aciona um botão no controle remoto que faz com que a TV mude a sintonia para o canal desejado. Pensando em termos de objetos, o objeto **você** interagiu com o objeto **controle remoto** que interagiu com o objeto **TV** para resolver seu problema e realizar a mudança de canal.

FIGURA 1 - PESSOA, TV E CONTROLE REMOTO



FONTE: Microsoft Cliparts, 2014

Imagine agora que por um motivo qualquer, o cabo que liga a TV à antena se desconecte, ocasionando a perda de sinal. Para resolver o problema, você se dirige até a TV e reconecta o cabo da antena, o que restaura o sinal. Perceba que neste caso, fazendo a analogia com um *software* orientado a objetos, o objeto você interagiu diretamente com o objeto TV para resolver o problema, pois a solução não estava disponível no objeto controle remoto.

Neste paradigma de programação, o trabalho do programador consiste em observar o domínio do problema a ser resolvido e modelá-lo em termos de objetos e suas relações. Imaginemos um sistema bancário, onde há a necessidade de registrar contas correntes, correntistas e operações bancárias. Cada correntista do banco seria representado por um objeto distinto, associado a uma conta corrente distinta e esta conta corrente associada a um ou mais objetos representando operações bancárias.

Para exemplificarmos uma situação mais específica relacionada a um objeto, imagine uma situação onde exista um correntista associado a uma conta corrente. Internamente este objeto correntista possui diversos valores, entre eles nome, endereço, data de nascimento, CPF, estado civil e o próprio objeto conta corrente. Agora vamos supor que a pessoa representada por este objeto seja uma mulher que contraia matrimônio no mundo real e consequentemente assuma o sobrenome do marido e se mude para outro endereço. Estas alterações devem ser refletidas no sistema bancário e o cadastro desta correntista deve ser atualizado. Em termos de programação orientada a objetos, o objeto correntista deve ter seus valores internos alterados e esta alteração será feita pelos procedimentos internos do próprio objeto. Esta alteração não refletiria nos objetos conta corrente e operações bancárias, pois não diz respeito a eles. Tal característica dos objetos é conhecida como encapsulamento e trataremos dela com maiores detalhes mais tarde, neste Caderno de Estudos.

Agora vamos analisar o seguinte: um banco possui diversos correntistas e contas bancárias e cada conta bancária deve estar relacionada a diversas operações bancárias. Como faremos para representar dois ou mais correntistas diferentes? Como faremos para representar duas ou mais contas correntes? E quanto às operações bancárias?

É neste contexto que utilizaremos o conceito de classe. Enquanto os objetos representam as entidades que fazem parte de um domínio, seus valores internos e operações, a classe é o elemento que dá forma a tais objetos. Para exemplificar, utilizando o mesmo exemplo do sistema bancário, o que define quais informações que todos os objetos terão internamente e que tipo de operação poderá ser feita com essas informações é a classe. Considere a classe como um molde, uma forma através da qual todos os objetos correntistas são moldados. O mesmo ocorre para conta corrente e operação bancária.

Um objeto Correntista seria um tipo de dado na memória que contivesse internamente os valores {nome: Kathy Sierra; endereço: Rua XV de Novembro, 23; data de nascimento: 20/08/1979; CPF: 020523444-64; estado civil: casada}. Perceba que podemos criar e manter quantos objetos correntistas quiséssemos, excetuando a limitação de memória do computador, cada um com seus valores internos individuais e que, a informação sobre quais informações eles contêm (nome, endereço, data de nascimento, CPF, estado civil e o próprio objeto conta corrente) em conjunto com as operações realizáveis com estas informações estariam definidas na classe.

Estendendo o conceito para outro exemplo, vamos considerar uma classe de automóvel, que internamente contém as informações modelo, motorização e as operações acelerar e parar. A classe Automóvel dá forma aos objetos que o sistema efetivamente utiliza, em que onde os objetos são representados pelo Gol, Cruze e Jetta, cada um com valores distintos para modelo e motorização.

O termo técnico utilizado na programação orientada a objetos para definir essa situação é instância. Dizemos que Gol, Cruze e Jetta são instâncias da classe Automóvel, ou seja, são objetos instanciados que obedecem a uma formatação pré-estabelecida. Essa formatação implica que todos os objetos instanciados a partir da classe Automóvel tenham internamente as informações de modelo e motorização e as operações de acelerar e parar.



A **instanciação** é o momento quando criamos um objeto em memória com base na classe que o define.

A partir do momento em que definimos uma classe, é como se criássemos um novo tipo de dado para declaração de variáveis. Uma classe define todas as características comuns a um tipo de objeto. Por exemplo, em uma linguagem de programação geralmente temos variações do tipo inteiro, caractere, booleano etc. A criação de uma classe Automóvel permite que utilizemos esse tipo para a declaração de variáveis (Quadro 1).

QUADRO 1 - DECLARAÇÃO DE VARIÁVEIS EM JAVA

```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         int idade;
10        char sexo;
11        Automovel Auto;
12    }
13 }
14

```

FONTE: O autor

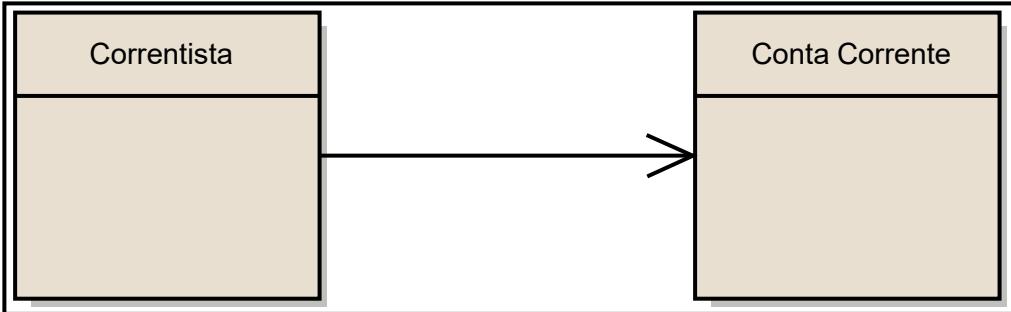
Uma classe é composta geralmente por três elementos básicos:

- 1) Nome: o nome de uma classe é sempre formado por um substantivo, que pode ser simples ou composto. Na linguagem de programação Java (que utilizaremos ao longo deste caderno), os nomes das classes sempre começam com letra maiúscula, caso sejam substantivos simples ou compostos. Exemplos de bons nomes para classes são: Automóvel, Correntista, Conta Corrente, Gerador de Relatórios.
- 2) Atributos: os atributos de uma classe são as informações internas, como o nome ou endereço da correntista ou a motorização do automóvel. Essas informações em geral são declaradas com um tipo de dados, da mesma forma que as variáveis em um programa. Um detalhe a ser destacado é que é comum os objetos possuírem atributos que sejam outros objetos. Por exemplo, um objeto correntista poderia ter um atributo do tipo conta corrente atrelado ao mesmo. Neste tipo de situação, onde duas classes são definidas de forma a associarem objetos, temos um relacionamento chamado **Associação**, conforme mostra a Figura 2. Dizemos que o valor dos atributos de um objeto enquanto ele está na memória é seu **estado**.
- 3) Métodos ou operações: descrevem os procedimentos que podem ser realizados com os atributos da classe. Em geral são nomeados com verbos no imperativo ou infinito indicando qual operação será realizada. Por exemplo: acelerar, parar, emitir extrato, realizar saque. Existe uma categoria especial de métodos que são chamados de construtores. O objetivo dos métodos construtores é construir o objeto no momento de sua criação, garantindo que esteja em um estado válido, sendo que uma classe pode ter mais de um método construtor. Aos métodos de um objeto enquanto ele está na memória damos o nome **comportamento**.



A notação utilizada na Figura 2 para a definição das classes e seu relacionamento é conhecida como UML (Unified Modeling Language) e será tema de disciplinas específicas ao longo de seu curso. Maiores detalhes sobre a UML podem ser acessados em: <http://www.omg.org/gettingstarted/what_is.uml.htm>.

FIGURA 2 - RELACIONAMENTO DE ASSOCIAÇÃO ENTRE DUAS CLASSES



FONTE: O autor

Ainda com relação ao comportamento dos objetos, existe uma categoria especial de métodos que permitem o acesso externo. Os métodos com essa característica fazem parte da interface pública do objeto e é através deles que são trocadas as mensagens. Quando um objeto A se comunica com um objeto B através de uma mensagem, isso simplesmente significa que A chamou um método da interface pública de B, solicitando que este mude seu estado.

Em resumo, podemos dizer que quando instanciamos um objeto em memória a partir de uma classe que o define, o valor de seus atributos é conhecido como seu **estado** e as operações que ele pode realizar são o que definem seu **comportamento**. Muitas operações têm como objetivo a mudança do estado do objeto.

4 VANTAGENS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

Sintes (2002) afirma que a programação orientada a objetos define **seis objetivos** para o desenvolvimento de *software*:

- 1) Natural: a programação orientada a objetos **produz software** mais inteligível, pois permite que você defina os componentes de seu *software* com base em objetos e comportamentos do mundo real. **Permite ainda que você modele um problema em nível funcional** e não em nível de implementação, deixando tempo para se concentrar no problema a ser resolvido.
- 2) Confiável: *software* útil deve ser confiável. Infelizmente existe uma cultura de tolerância ao erro na área de desenvolvimento de *software*. Por exemplo, quando foi a última vez que sua geladeira quebrou? **Programas orientados a objetos bem projetados e cuidadosamente escritos tendem a ser mais confiáveis**. Sua natureza modular permite que sejam feitas manutenções em **uma parte do software sem afetar outras**. Os objetos isolam o conhecimento e a responsabilidade de onde pertencem. Além disso, a orientação a objetos intrinsecamente aprimora os testes, ao isolar conhecimento e responsabilidade em um único local. Uma vez que você tenha validado um componente, você pode reutilizá-lo com mais confiança.

- 3) Reutilizável: da mesma forma que um construtor ou engenheiro eletricista reaproveitam tijolos, e circuitos, a programação orientada a objetos incentiva a reutilizar soluções para os problemas. É possível reutilizar classes orientadas a objetos bem feitas em diversos programas diferentes e ainda estender seu comportamento através do que é conhecido como polimorfismo. Através da programação orientada a objetos, você pode modelar ideias gerais e usar essas ideias para resolver problemas específicos. Objetos específicos são em geral construídos utilizando partes genéricas reaproveitadas.
- 4) Manutenível: o ciclo de vida de um programa não encerra quando você o entrega no cliente. Em geral, o desenvolvimento de um programa representa apenas 20% a 30% do tempo gasto, enquanto a manutenção pode chegar a representar 80% deste tempo. Um código orientado a objetos bem escrito e projetado garante que uma correção em um lugar não tenha consequências inesperadas em outro, além de facilitar o entendimento por parte de outros desenvolvedores.
- 5) Extensível: quando você construir uma biblioteca de objetos, também desejará estender a funcionalidade de seus próprios objetos. A programação orientada a objetos apresenta ao programador diversos recursos para estender código, como herança, polimorfismo, sobreposição, delegação e uma variedade de padrões de projeto.
- 6) Oportuno: o ciclo de vida de um projeto de *software* é normalmente medido em semanas. A programação orientada a objetos facilita os rápidos ciclos de desenvolvimento, pois ao dividir o problema em vários objetos o desenvolvimento de cada parte pode ocorrer paralelamente e em classes independentes.

Outra vantagem é que através da reutilização, os programadores aprendem a compartilhar o código que criam. Compartilhar significa encorajar outros desenvolvedores a utilizarem o seu código e consequentemente, expô-lo a mais gente que pode encontrar *bugs*. Como dizem Hunt e Thomas (1999, p. 64): “ninguém, na breve história da atividade de desenvolvimento de software já escreveu código perfeito. Você também não o fará. Aceite seus erros”. O compartilhamento ainda possui a vantagem de levar o programador a escrever código através de interfaces limpas e fáceis de utilizar, melhorando automaticamente a qualidade de seu código. Uma boa prática de programação orientada a objetos diz que se deve programar para a interface e não para a implementação, ou seja, seus objetos devem ser construídos pensando no próximo programador que os utilizará.

**Fazer funcionar da melhor forma possível e de sorte
que a maior parte das pessoas vão entender**

5 LINGUAGENS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Para que possamos programar de forma orientada a objetos, é necessário que utilizemos uma linguagem que dê suporte a suas características. Sintes (2002) coloca que as três características mais importantes da programação orientada a objetos são o encapsulamento, a herança e o polimorfismo. Para uma linguagem de programação ser chamada de orientada a objetos, ela deve dar suporte a essas três características.



Encapsulamento, herança e polimorfismo serão abordados nas Unidades 2 e 3 deste Caderno de Estudos.

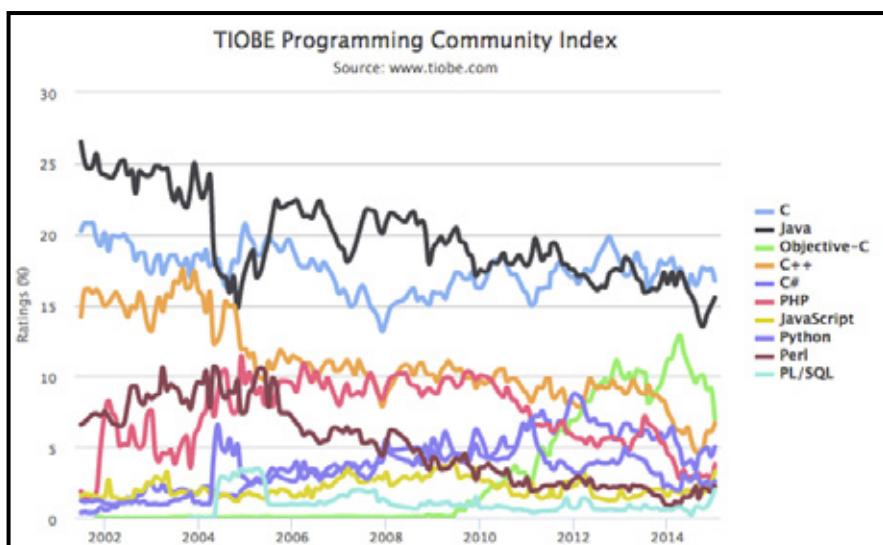
Para demonstrarmos os conceitos e as técnicas envolvidas na programação orientada a objetos é necessário que se utilize uma linguagem de programação adequada. Existem diversas linguagens de programação no mercado que atenderiam a essa demanda, entre elas podemos destacar *Python*, *C#*, *C++*, *Objective C*, *Java* e *SmallTalk*.

Neste caderno, todos os exemplos de código fonte e as atividades serão resolvidos utilizando a linguagem de programação Java. Entre as vantagens do Java, podemos destacar:

- Segue com o rigor adequado os conceitos de encapsulamento, herança e polimorfismo. Esta característica da linguagem facilita o aprendizado da programação orientada a objetos e mesmo a aplicação desta em outra linguagem.
- Possui sintaxe semelhante a do C e C++, o que diminui o tempo de migração para quem já está familiarizado com estas.
- As características de Java inspiraram fortemente outras linguagens de programação, como por exemplo, o C#. Se você conhecer Java, o tempo de migração para C# é bastante reduzido, pois esta última possui diversas classes que funcionam exatamente como no Java.
- Existem diversas ferramentas gratuitas para desenvolvimento e aprendizado. A Oracle e a própria comunidade disponibilizam uma variedade bastante interessante de ferramentas e complementos para programar em Java. Entre eles podemos destacar as IDEs (*Integrated Development Environment*) Netbeans e Eclipse.
- Java é multiplataforma, caso você não seja um usuário Windows e use Linux, ou mesmo planeja desenvolver para Linux, Unix, ou MacOs, pode utilizar exatamente o mesmo código fonte, não havendo necessidade de conversão.

- Java possui diversas características que o tornam robusto e seguro, sendo uma das principais alternativas para o desenvolvimento de *software* que tenha esses requisitos não funcionais como premissa.
- Java ainda está entre as linguagens de programação mais utilizadas no mundo, conforme verificamos de acordo com os índices do TIOBE e da IEEE (Figura 3 e Figura 4, respectivamente), o que aumenta seu apelo no mercado e consequentemente a empregabilidade de quem a domina.

FIGURA 3 - LINGUAGENS DE PROGRAMAÇÃO



FONTE: TIOBE, 2014. Disponível em: <<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>>. Acesso em: 3 jun. 2014.

Figura 4 - Linguagens de programação mais populares por plataforma de desenvolvimento

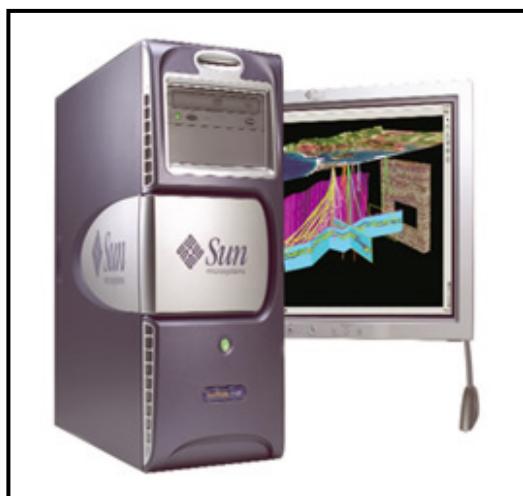
Language Rank	Types	Spectrum Ranking
1. Java	🌐📱💻	100.0
2. C	💻📱💻	99.2
3. C++	💻📱💻	95.5
4. Python	🌐💻	93.4
5. C#	🌐📱💻	92.2
6. PHP	🌐	84.6
7. Javascript	🌐📱	84.3
8. Ruby	🌐	78.6
9. R	💻	74.0

FONTE: IEEE, 2014. Disponível em: <<http://spectrum.ieee.org/static/interactive-the-top-programming-languages>>. Acesso em: 3 jun. 2014.

6 HISTÓRIA DA LINGUAGEM DE PROGRAMAÇÃO JAVA

No final de 1990, a *Sun Microsystems* liderava o mercado mundial de computadores *workstation* (Figura 5). Na época, a empresa tinha a fama de possuir os melhores engenheiros do Vale do Silício e um deles, chamado Patrick Naughton decidiu pedir demissão, entretanto, ao fazê-lo, justificou a Scott Mcnealy (um dos diretores da *Sun*) que estava saindo por que não gostava da maneira como as coisas estavam sendo feitas na empresa. McNealy pediu então que Patrick escrevesse um memorando “sem censura”, mostrando qual sua visão para a empresa. O memorando foi impresso e então distribuído para que qualquer um tivesse acesso a ele. Dois engenheiros da *Sun*, James Gosling e Bill Joy, leram o memorando e disseram que se sentiam da mesma forma.

FIGURA 5 - WORKSTATION DA SUN



FONTE: PROACT, 2014. Disponível em: <<http://www.tech.proact.co.uk/>>. Acesso em: 3 jun. 2014.

Temendo perder um time inteiro de engenheiros, Mcnealy fez uma contraproposta. Ele forneceria um local, verba, liberaria os engenheiros solicitados, e, ainda permitiria que trabalhassem no que quisessem desde que fosse feito da forma descrita no memorando. Nasceu então o Projeto *Green*, cujo objetivo era “Desenvolver e licenciar um ambiente operacional para dispositivos de consumo que habilitam serviços e informação para estar persuasivamente presente através da infraestrutura digital emergente”. A equipe do projeto pode ser visualizada na Figura 6.

James Gosling, que havia escrito sua primeira linguagem de programação com 14 anos, ficou responsável pela criação da linguagem de programação que permitiria a interação com o dispositivo.

FIGURA 6 - EQUIPE DO PROJETO GREEN



FONTE: Disponível em: <<https://duke.kenai.com/green/index.html>>. Acesso em 10 jun. 2014.

Ao final de dois anos, a equipe do Projeto *Green* finalmente apresenta o Star7, um PDA criado para os mais diversos fins, entre eles o controle da automação doméstica. O Star7 incluía um sistema operacional (*GreenOS*), um visor *touchscreen* animado, um *toolkit* para desenvolvimento, bibliotecas padrão e a linguagem *Oak*, que funcionava de forma independente de processador, através de uma máquina virtual. Para demonstrar o funcionamento, fizeram o Star7 interagir com uma televisão e um videocassete, simplesmente arrastando e tocando na interface gráfica sem utilizar nenhum fio que os conectasse a estes dispositivos.

Dadas às devidas proporções, poder-se-ia dizer que o Star7, mostrado na Figura 8, foi uma espécie de precursor dos *tablets*. O dispositivo era revolucionário também na questão de interface com o usuário, pois disponibilizava ajuda através de um agente inteligente representado por um avatar e chamado de Duke (Figura 7). Duke, que foi inspirado na insígnia do uniforme dos personagens do seriado de televisão *Star Trek*, viria a se tornar o mascote da plataforma Java.

FIGURA 7 - ORIGEM DE DUKE, MASCOTE DA PLATAFORMA JAVA



FONTE: Adaptado de *TrekCore* (2014)

FIGURA 8 - STAR7 E SUA INTERFACE GRÁFICA



FONTE: *Tech Insiders* (1997). Disponível em: <<http://techinsider.org/java/research/1998/05.html>>. Acesso em: 3 jun. 2014.

Conforme *Tech Insider* (1997), o projeto ganhou destaque e começou a atrair clientes potenciais na indústria da TV a cabo. Uma nova empresa chamada de *FirstPerson* foi criada para lidar com as questões de mercado. Depois de diversas negociações malsucedidas, o consenso geral era de que as empresas de TV a cabo não estavam preparadas para lidar com o seu conteúdo da forma interativa como o Star7 permitia. Devido ao fracasso das negociações com as empresas de TV a cabo, surgiu a possibilidade de se aproveitar pelo menos a linguagem de programação na mídia de comunicação que recentemente havia se tornado popular: a internet. Quando isso ocorreu, a linguagem já havia sido renomeada para Java (em homenagem a um tipo especial de café), pois já existia uma linguagem de programação com o nome *Oak*.

O navegador *web* mais popular da época se chamava *Mosaic*, o que levou a equipe a produzir um clone do mesmo, chamado de *WebRunner*, com a diferença de ser habilitado para Java. (Figura 9). O *WebRunner* executava somente uma demo, mas era uma demo impressionante para 1994. O navegador exibia páginas dinâmicas e conteúdo executado diretamente nele, algo que nunca havia sido feito antes (*TECH INSIDER*, 1997).

Em 1995, a *Sun* resolveu disponibilizar a linguagem de programação Java livremente para *download* na internet, sabendo que era (e ainda é) a melhor forma de se incentivar a adoção de uma nova tecnologia e ferramenta. O Java atingiu a marca de 10.000 *downloads* muito antes do que qualquer um dos integrantes da equipe imaginava. Para ter uma ideia do que esse número significa, lembre-se de que faz quase 20 anos que isso aconteceu e que estamos falando da versão 1.0 de uma linguagem de programação que quase ninguém conhecia. O volume de tráfego ocasionado pelos *downloads* do Java e pelos *e-mails* contendo dúvidas e sugestões obrigou a *Sun* a aumentar a largura de banda de sua conexão com a internet (*TECH INSIDER*, 1997).

Por fim, para ratificar o sucesso da nova linguagem de programação, Marc Anderseen, responsável pelo navegador de Internet *Netscape Navigator*, assinou um contrato onde garantia a execução da linguagem Java dentro de seu produto.

Desde sua introdução em maio de 1995, a Plataforma Java foi adotada pela indústria de Tecnologia da Informação de forma mais rápida que qualquer outra tecnologia em computação. Atualmente, boa parte dos grandes *players* da indústria mundial utiliza a tecnologia Java como parte integrante de seus produtos de *software*. Exemplos de aplicações Java em larga escala podem ser encontrados aqui mesmo no Brasil, como o Imposto de Renda Pessoa Física, o sistema de loterias, o sistema de pagamentos bancários do banco central e o sistema dos correios (CPAI, 2011).

FIGURA 9 - ANÚNCIO DO WEBRUNNER

WebRunner

WebRunner is a World Wide Web browser that brings true interactivity to the Internet. WebRunner makes the Internet "come alive". It builds on the network browsing techniques established by Mosaic and expands them by adding dynamic behavior that transforms static documents into dynamic applications capable of real-time interactive response. Using WebRunner you can create applications that range from interactive games to dynamic forms to customized newspapers to interactive shopping ... the possibilities are endless.

WebRunner also provides a new way for users to access these applications. Software transparently migrates across the network. There is no such thing as "installing" software. It just comes when you need it. Content developers can embed new software, media types and protocols in WWW pages, and the extensions reach the user's system automatically.

What makes this dynamic behavior possible is Java (formerly "Oak"), the underlying environment in which WebRunner is built. Java is a simple, dynamic, multithreaded, safe, compact and portable object-oriented programming language and runtime system. Java has an architecture-neutral distribution format, so that Java content runs on anyone's WebRunner home page, regardless of the underlying CPU architecture.

WebRunner Alpha2 Features:

- Full-Function WWW Browser
 - HTML-compatible with Mosaic and Netscape
 - supports all standard Internet protocols
 - fast performance
- Enables Interactive Content
- Dynamic Content Loading
 - occurs transparently across the network
 - add new protocols and applications on the fly
 - platform-independent
- WWW Newsreader
- Security and Authentication Support
 - file system protection and code checking
- Includes full Java language and runtime system
- Available on Solaris

WebRunner Beta Features:

- WYSIWYG HTML Editor
 - no HTML knowledge necessary
 - implements all common HTML extensions
 - drag and drop links, images, audio, applications into browser
- Security and Authentication Support
 - S-HTTP, RSA public key encryption and authentication
- HTTP server with support for real-time data feeds
- Available on Solaris, Windows95, and MacOS

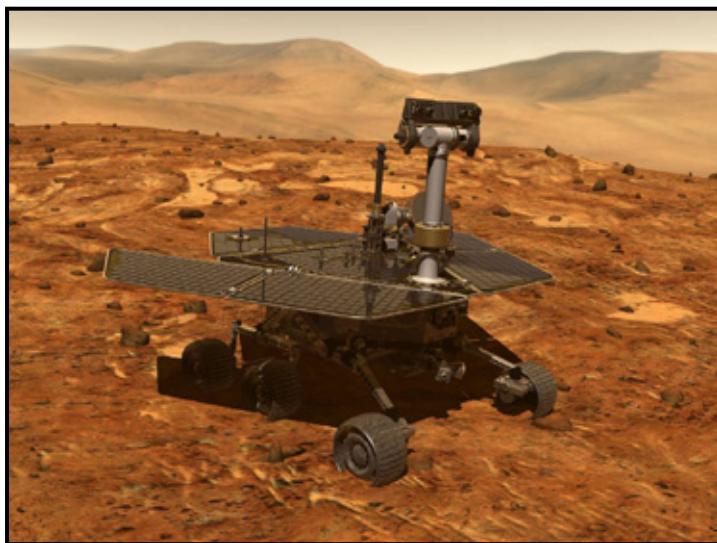
For instructions on installing WebRunner, visit <http://tachyon.eng/folio/>



FONTE: TECH-INSIDER, 1997. Disponível em: <<http://tech-insider.org/java/research/1998/05.html>>. Acesso em: 8 jul. 2014

Conforme Oracle (2011), Java está em computadores, impressoras, roteadores, dispositivos VOIP, caixas eletrônicos, videogames, sistemas de TV via satélite e cabo, cartões de crédito, telefones celulares, *smartphones*, leitores de *bluray* e até mesmo em Marte (Figura 10). Por tudo que abordamos neste tópico, consideraremos que existem mais benefícios em aprender Java do que em aprender qualquer outra linguagem de programação.

FIGURA 10 - VEÍCULO NÃO TRIPULADO EM MARTE



FONTE: Nasa, 2014. Disponível em: <<http://mars.jpl.nasa.gov/mer/gallery/artwork/rover2browse.html>>. Acesso em: 20 ago. 2014.

7 CARACTERÍSTICAS DA PLATAFORMA JAVA

Para entendermos de forma clara o conceito de multiplataforma, inicialmente consideraremos uma linguagem de programação que não possua essa característica, como o C++, por exemplo. Nesse tipo de linguagem, o código binário (executável) possui instruções e bibliotecas específicas para se comunicar com o Sistema Operacional. Para exemplificar, imagine um programa escrito em C++ e construído totalmente em ambiente *Windows*. O executável desse programa possui instruções específicas para interagir com o Sistema Operacional em questão, fazendo alocação de memória, acesso a disco, acesso à rede, acesso a bibliotecas, desenho de telas etc. Suas chamadas de sistema e bibliotecas não seriam compatíveis com um sistema operacional *Linux* ou *MacOS*, não permitindo a comunicação entre o sistema operacional e o programa. Caso houvesse compatibilidade entre as chamadas de sistema de dois sistemas operacionais distintos, como ocorre entre algumas versões do *Windows*, o programa funcionaria perfeitamente.

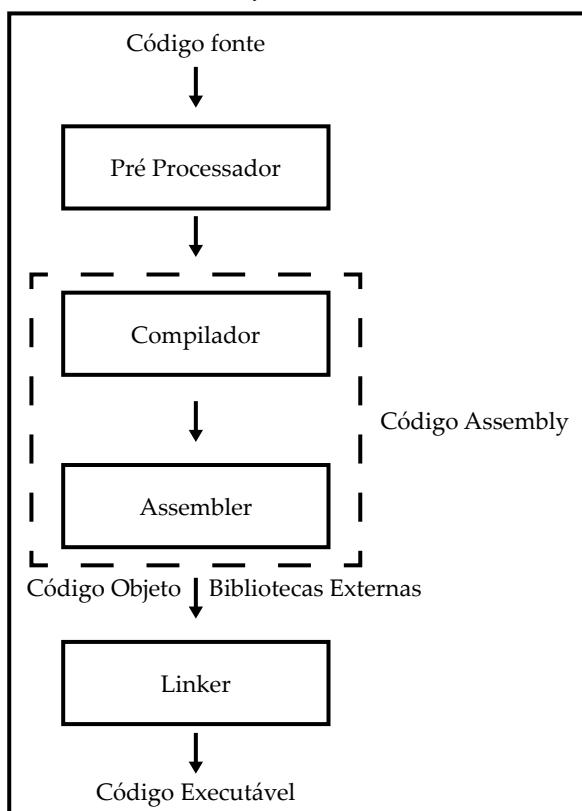


Uma chamada de sistema (system call) é o mecanismo através do qual os programas se comunicam com o núcleo do sistema operacional, alocando e liberando recursos.

Caso precisemos que determinado programa possa executar em dois sistemas operacionais com chamadas de sistema distintas, como, por exemplo, *Linux* e *Windows*, este programa deverá ser compilado duas vezes, uma para cada sistema operacional. Este procedimento ocorre com diversos *softwares* conhecidos, como o *Google Chrome*, *Mozilla Firefox*, e *BROffice*, que possuem versões para os dois sistemas operacionais.

A Figura 11 ilustra o processo de compilação de um programa escrito na linguagem de programação C++. Nesta figura, podemos perceber que o código fonte é submetido inicialmente ao pré-processador, cuja função se restringe a organizar o código fonte removendo comentários, organizando variáveis e inserindo o conteúdo de arquivos referenciados, caso haja algum. O compilador pega o código fonte organizado pelo pré-processador e o transforma em linguagem *assembly*. O *assembler* converte a linguagem *assembly* para o que é conhecido como código objeto. Caso o programa refcrcie bibliotecas externas ou funções contidas em arquivos com código fonte diferentes do principal, o *linker* combina todos os objetos com o resultado compilado destas funções em um único arquivo com código executável (O PROGRAMAÇÃO EM C, 1999).

FIGURA 11 - COMPILAÇÃO DE UM PROGRAMA EM C++



FONTE: Disponível em: <http://www.urisan.tche.br/~janob/Cap_1.html>. Acesso em: 1 ago. 2014.

A plataforma Java utiliza o conceito de máquina virtual, inspirado particularmente pela linguagem de programação *Eiffel*. De acordo com este conceito, os programas desenvolvidos não precisam conhecer as chamadas de sistema ou as bibliotecas do sistema operacional, visto que sua execução é completamente controlada pela máquina virtual. Em outras palavras, se o programa precisa alocar um espaço em disco, ele solicita à máquina virtual, que atua como intermediária e repassa a solicitação diretamente ao núcleo do sistema operacional. Logicamente, a máquina virtual deve conhecer as chamadas de sistema do sistema operacional onde estiver executando. Por isso, no caso do Java, existem máquinas virtuais disponíveis para *download* voltadas para diversos sistemas operacionais. Caso você pretenda executar o programa escrito em Java no sistema operacional *Linux*, deve utilizar a máquina virtual desenvolvida para este sistema.

A utilização de uma máquina virtual para execução dos programas traz diversas vantagens ao desenvolvedor, entre elas podemos destacar:

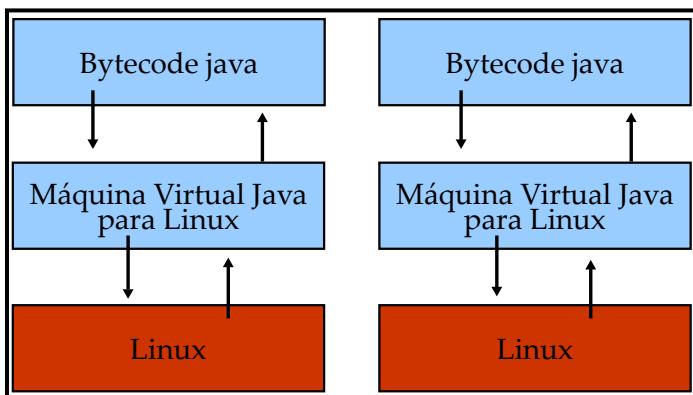
- Segurança: toda e qualquer interação feita com o sistema operacional passa por diversas validações da máquina virtual antes de ser efetivamente realizada. Essa característica torna extremamente difícil escrever código malicioso em Java.
- Gerenciamento de memória: a memória alocada pelos programas Java é completamente controlada pela máquina virtual, que os gerencia em uma *sandbox* separada do sistema operacional. Em um nível mais micro, a máquina virtual aloca e desaloca espaço para os objetos em memória.
- Robustez: Erros graves que possam ocorrer por qualquer motivo dentro de um programa que execute dentro da máquina virtual são contidos por ela e não repassados ao sistema operacional.
- Independência de Plataforma: os programas são escritos para a máquina virtual, que faz a intermediação entre estes e o sistema operacional, permitindo que o mesmo programa compilado em um sistema execute em outro.



Em segurança da informação, uma sandbox (caixa de areia) é um mecanismo utilizado para separar e testar programas que não foram testados ou não são confiáveis.

O aspecto que mais nos interessa é a independência de plataforma. Mas como o Java realiza isso? Caelum (2014) coloca que 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". Precisamos reescrever um mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis. O Java utiliza do conceito de máquina virtual, onde existe, entre o sistema operacional e a aplicação, uma camada extra responsável por "traduzir" – mas não apenas isso – o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento, conforme a Figura 12.

FIGURA 12 - FUNCIONAMENTO DA MÁQUINA VIRTUAL JAVA



FONTE: Disponível em: <<http://www.caelum.com.br/apostila-java-orientacao-objetos/o-que-e-java/#2-3-maquina-virtual>>. Acesso em: 12 jun. 2014.

Dessa forma, a maneira com a qual você abre uma janela no *Linux* ou no *Windows* é a mesma: você ganha independência de sistema operacional. Ou, melhor ainda, independência de plataforma em geral: não é preciso se preocupar em qual sistema operacional sua aplicação está rodando, nem em que tipo de máquina, configurações etc. (CAELUM, 2014).

Essa camada, a máquina virtual, não entende código Java, ela entende um código de máquina específico. Esse código de máquina é gerado por um compilador Java, como o javac, e é conhecido por "*bytecode*", pois existem menos de 256 códigos de operação dessa linguagem, e cada "*opcode*" gasta um *byte*. O compilador Java gera esse *bytecode* que, diferente das linguagens sem máquina virtual, vai servir para diferentes sistemas operacionais, já que ele vai ser "traduzido" pela JVM (CAELUM, 2014).



Write once, run anywhere (escreva uma vez, execute em qualquer lugar) era o slogan que a Sun Microsystems utilizava para promover o Java, fazendo referência à característica multiplataforma.

O objetivo deste *bytecode* é funcionar como um intermediário entre o programa e o sistema operacional. Caso você compile um programa Java no *Linux*, basta pegar o binário gerado pelo compilador (*bytecode*) e executá-lo em um ambiente *Windows* onde a máquina virtual Java esteja instalada. O Quadro 2 mostra uma classe escrita em código fonte Java e um trecho desta classe gerada em *bytecode*.

QUADRO 2 - BYTECODE GERADO PELA JAVA VIRTUAL MACHINE

<pre>public class Fatorial { public static final int factorial(int n) { return (n == 0)?1:n*fatorial(n - 1); } }</pre>	<pre>// método factorial 0: iload_0 1: ifne #8 4: iconst_1 5: goto #16 8: iload_0 9: iload_0 10: iconst_1 11: isub 12: invokestatic Faculty.fac (I)I (12) 15: imul 16: ireturn</pre>
---	---

FONTE: Adaptado de: <<https://www.ime.usp.br/~kon/MAC5715/slides/bytecdodes.pdf>>. Acesso em: 12 jun. 2014.

7.1 HOTSPOT E JIT

Existem centenas, se não milhares de linguagens de programação para se desenvolver *softwares*. Algumas delas são chamadas de linguagens compiladas, pois o que executará é resultado do trabalho de um compilador, enquanto outras linguagens são chamadas de interpretadas, pois dependem de um interpretador.

Nas linguagens compiladas, o compilador constrói um (ou mais) arquivo binário, onde transforma código fonte para código nativo da plataforma para a qual se está compilando. O Pascal, por exemplo, em seu processo de compilação transforma um arquivo com extensão .pas para um arquivo com extensão .exe. Em geral, as linguagens de programação compiladas apresentam como características a tipagem forte e a compilação de todo o programa de uma só vez. A tipagem forte é a característica que exige que o tipo da variável seja informado na declaração da mesma. A compilação de todo o programa significa que, somente após a checagem léxica e sintática de todos os arquivos fonte referenciados, o compilador gerará o binário e consequentemente permitirá a execução do programa. Como este binário é gerado em código nativo, na maioria das vezes os programas compilados apresentam melhor desempenho do que os interpretados.

Nas linguagens interpretadas, como VBScript e PHP, não existe necessidade de declaração do tipo de variáveis, o que leva o interpretador a fazer a checagem em tempo de interpretação e automaticamente acarreta uma redução no desempenho. Os programas interpretados não geram código nativo, então exigem a presença de um interpretador no ambiente onde funcionarão. Para você entender melhor o executável gerado pelo Pascal ou C++ pode simplesmente ser instalado em um computador (desde que o sistema operacional seja compatível com o da compilação) que o mesmo funcionará mesmo que neste computador não existe um compilador Pascal ou C++. Já um programa escrito em Vbscript ou PHP exige a presença de um interpretador no ambiente onde será instalado. Caso você queira montar um servidor *web* que permita a colocação de páginas dinâmicas escritas em PHP, deve obrigatoriamente instalar o interpretador PHP no servidor. Outra característica das linguagens interpretadas é que não existe a avaliação do programa como um todo, e o interpretador as interpreta em geral linha a linha. Em contrapartida, uma vantagem das linguagens de programação interpretada, é que não se perde tempo com o processo de compilação de um programa, que pode levar muito tempo nas linguagens de programação compiladas, dependendo do tamanho do programa.

As características, vantagens e desvantagens das abordagens interpretadas e compilada são ilustradas no Quadro 3.

QUADRO 3 - COMPARAÇÃO ENTRE AS ABORDAGENS ESTRUTURADA E INTERPRETADA

Abordagem	Vantagens	Desvantagens
Compilador	<ul style="list-style-type: none"> Permite estruturas de programação mais complexas. Gera arquivo executável, gerando maior autonomia e segurança. Mais desempenho. 	<ul style="list-style-type: none"> Correção de erros de forma estática. Tradução de código fonte em várias etapas. Consumo mais memória.
Interpretador	<ul style="list-style-type: none"> Consumo menos memória. Tradução em uma única etapa. 	<ul style="list-style-type: none"> Menos desempenho. Não gera executável, acarretando menos segurança no código fonte.

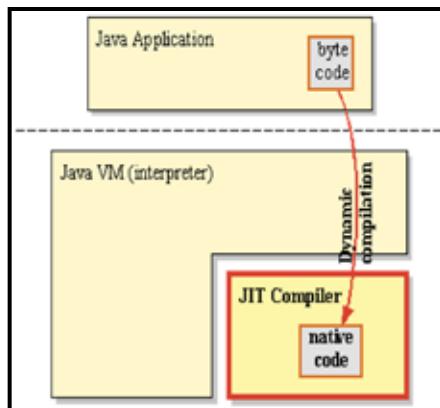
FONTE: Adaptado de: <<http://www.timaster.com.br/>>. Acesso em: 12 set. 2014.

A linguagem de programação Java utiliza uma abordagem híbrida em sua arquitetura, conhecida como *mixed mode*. Por esse motivo é comum ouvirmos que a linguagem de programação Java é compilada e interpretada. As características principais do *mixed mode* são:

- Inicialmente o código fonte Java é compilado, fazendo checagem léxica e sintática de todos os arquivos-fonte envolvidos.
- Os *bytecodes* são gerados e interpretados pela máquina virtual Java (JVM).
- Com o objetivo de aumentar o desempenho, a JVM detecta os *hotspots* e elabora uma estratégia adaptativa onde o código existente nesses locais é compilado para código nativo. Em máquinas virtuais modernas, esse processo pode acontecer diversas vezes, até se chegar a um desempenho ótimo, chegando a gerar instruções para processadores específicos.

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 laços de repetição. Quando a JVM julgar necessário, ela vai compilar estes códigos para instruções realmente nativas da plataforma, tendo em vista que isso vai provavelmente melhorar o desempenho 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? Teoricamente, compilar dinamicamente, a medida do necessário, pode gerar uma *performance* melhor. O motivo é simples: imagine um arquivo executável gerado pelo *Visual Basic*, pelo *gcc* ou pelo *Delphi*; ele é estático. Ele já foi otimizado baseado em heurísticas, 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, conforme ilustrado na Figura 13. É por esse motivo que as JVMs mais recentes em alguns casos chegam a ganhar de códigos C compilados com o *GCC 3.x* (CAELUM, 2014).

FIGURA 13 - PROCESSO DE GERAÇÃO DE BYTECODE E INTERPRETAÇÃO E COMPILAÇÃO DE BYTECODE



FONTE: Disponível em: <http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=4688>. Acesso em: 12 set. 2014.

7.2 PLATAFORMA OU LINGUAGEM?

Mas afinal, Java é uma linguagem ou uma plataforma? Para respondermos a essa questão, precisamos inicialmente entender o que é uma plataforma quando se trata de desenvolvimento de *software*.

Conforme Techopedia (2010), uma plataforma é um grupo de tecnologias que são usadas como base para o desenvolvimento de aplicações. Esta plataforma deve encapsular um conjunto de padrões e desta forma habilitar os desenvolvedores a desenvolverem *software* que seja compatível com ela. No caso da plataforma Java a definimos desta forma, porque além da linguagem de programação, ainda existem uma máquina virtual e um conjunto de bibliotecas que obrigatoriamente a acompanham.

Os componentes mais importantes da plataforma Java são o Java *Runtime Environment* (JRE) e o Java *Development Kit* (JDK). Quando instalado em um computador, o JRE fornece ao sistema operacional a máquina virtual que é necessária para executar aplicações Java. Já o JDK é uma coleção de ferramentas que podem ser utilizadas por um programador para CRIAR aplicações Java. Logicamente, o JRE é parte integrante do JDK.

Um detalhe importante a ser salientado, é que a Máquina Virtual Java, assim como a linguagem de programação, possui especificações muito bem definidas e abertas. Isso significa que qualquer um pode escrever um compilador e máquina virtual para o Java, desde que obedeça a estas especificações. Isso pode ser comprovado pela existência de outras máquinas virtuais, pois além da HotSpot da Oracle/Sun, existem ainda a J9 da IBM, a Jrockit da Oracle/BEA, a Apple JVM, entre (SILVEIRA, et al., 2011).

A plataforma Java é dividida em quatro grandes grupos, cada um para um tipo diferente de aplicação (ORACLE, 2012):

- 1) Java *Standard Edition* (JSE) – disponibiliza as principais funcionalidades da linguagem de programação Java. Ela define os tipos primitivos e objetos desde os mais básicos até os de mais alto nível, utilizados para redes, segurança, acesso a banco de dados, interface gráfica e *parsing* de XML. A JSE é utilizada geralmente para o desenvolvimento de aplicações *desktop*.
- 2) Java *Enterprise Edition* (JEE) – construída no topo da JSE, este grupo de tecnologias disponibiliza uma API (*application program interface*) e ambiente de execução para o desenvolvimento de aplicações de grande escala.
- 3) Java *Mobile Edition* (JME) – provê um conjunto de bibliotecas para a utilização em JVMs em dispositivos menores, como telefones celulares e microchips. Utiliza um subconjunto das bibliotecas da JSE em combinação com bibliotecas especialmente desenvolvidas para aplicações em dispositivos menores.
- 4) JavaFX – plataforma para a criação de RIA (*Rich Internet Applications*) usando interfaces de usuários leves. As aplicações desenvolvidas utilizando estas tecnologias utilizam gráficos acelerados por *hardware* para aproveitar as vantagens de clientes de alto desempenho. Possui ainda um conjunto de APIs de alto nível para conexão com redes e fontes de dados. Podem ser usadas como clientes JEE.

Atualmente, a *Oracle* utiliza uma estratégia de valorização, especialmente da plataforma Java, focando inclusive no desenvolvimento de aplicações em outras linguagens de programação.

Silveira et al. (2011) colocam que, com o passar do tempo e o amadurecimento da plataforma Java, percebeu-se que a linguagem ajudava em muitos pontos, mas também impedia um desenvolvimento mais acelerado em partes específicas. Com isso, outras linguagens, até então consideradas secundárias pelo mercado, começaram a tomar força por possuírem maneiras diferenciadas de resolver os mesmos problemas, mostrando-se mais produtivas em determinadas situações. Para aplicações Web em Java, por exemplo, é comum adotar outras linguagens. Grande parte do tempo utiliza-se CSS para definir estilos, HTML para páginas, JavaScript para código que será rodado no cliente, SQL para bancos de dados e XMLs de configuração.

A *Sun* percebeu posteriormente que não apenas a linguagem Java poderia se aproveitar das vantagens da plataforma: o excelente compilador JIT, algumas das melhores implementações de *Garbage Collector* existentes, a portabilidade e a quantidade de bibliotecas já consolidadas no mercado e prontas para serem usadas. Hoje, a JVM é capaz de interpretar e compilar o código escrito em diversas linguagens. Com o Rhino, código JavaScript pode ser executado dentro da JVM. Assim como código Ruby pode ser executado com JRuby, Python com Jython e PHP com Quercus. Existem linguagens criadas especificamente para rodar sobre a JVM, como Groovy, Beanshell, Scala e Clojure (SILVEIRA et al. 2011).

Essa diferenciação é importante, pois ao iniciar seus estudos na Programação Orientada a Objetos, utilizando a linguagem de programação Java, é comum perceber que sua produtividade é menor, comparada a outras linguagens do mercado como PHP, Phyton ou Ruby. É preciso que você entenda que, apesar de servir, também, para sistemas e equipes pequenas, a premissa principal da plataforma Java é atender aplicações de grande porte que executem em ambientes com múltiplos sistemas operacionais, com centenas ou até mesmo milhares de usuários.

Não tenha dúvidas que criar a primeira versão de uma aplicação usando Java, mesmo utilizando IDEs e ferramentas poderosas, será mais trabalhoso que muitas linguagens *script* ou de alta produtividade. Porém, com uma linguagem orientada a objetos e madura como o Java, será extremamente mais fácil e rápido fazer alterações no sistema, desde que você siga as boas práticas e recomendações sobre *design* orientado a objetos. Além disso, a quantidade enorme de bibliotecas gratuitas para realizar os mais diversos trabalhos (tais como relatórios, gráficos, sistemas de busca, geração de código de barra, manipulação de XML, tocadores de vídeo, manipuladores de texto, persistência transparente, impressão etc.) é um ponto fortíssimo para adoção do Java: você pode criar uma aplicação sofisticada, usando diversos recursos, sem precisar comprar um componente específico, que costuma ser caro (CAELUM, 2014).



A partir da versão 1.2, o Java ficou conhecido como o Java 2 1.2, principalmente para diferenciá-lo do Javascript. Vieram as versões 2 1.3 e 2 1.4 para então finalmente, na versão 1.5, o nome da tecnologia ser alterado para Java 5, por questões de marketing. Atualmente estamos na versão Java 8.

RESUMO DO TÓPICO 1

Neste tópico vimos:

- A Programação Orientada a Objetos surgiu como uma evolução da programação procedural e da programação modular.
- A principal evolução trazida pela Programação Orientada a Objetos foi a possibilidade de reuso e extensão de código através da herança e polimorfismo.
- Os programas orientados a objetos baseiam-se na existência de objetos e nas mensagens trocadas entre estes.
- A instanciação é o momento onde criamos um objeto com base em uma classe.
- Uma classe é composta por um nome, um conjunto de atributos e um conjunto de métodos, também conhecidos como operações.
- A linguagem de programação Java é considerada multiplataforma, pois o resultado de sua compilação, os bytecodes, são interpretados por uma máquina virtual que os isola de detalhes do sistema operacional.
- A plataforma Java se divide basicamente em quatro grandes grupos:
 - JavaSE – normalmente utilizado para desenvolvimento de aplicações desktop.
 - JavaEE – Apropriado para aplicações em ambientes distribuídos e de grande porte.
 - JavaME – versão do Java compatível com dispositivos menores, tais como cartões, telefones, microchips etc.
 - JavaFX – Opção fornecida pela Oracle para a criação de RIA em Java.
- A JVM utiliza estratégias como o HotSpot e o JIT para otimizar o desempenho de aplicações Java, levando a linguagem a ser mais rápida até mesmo que C, em determinadas situações.

AUTOATIVIDADE



1 Assinale a alternativa CORRETA:

- a) () Uma classe é uma espécie de forma que vai definir tributos e/ou comportamentos de todos aqueles objetos que forem instanciados a partir dela.
- b) () A POO (programação orientada a objetos) é equivalente à programação estruturada, pois se fundamenta no relacionamento entre variáveis e classes, através de polimorfismo e interfaces.
- c) () Um objeto é a representação prática de um método de uma classe, pois todo método precisa retornar um tipo de informação.
- d) () A partir do momento em que criamos um objeto, temos um novo TIPO disponível para utilização.

2 Descreva três vantagens da programação orientada a objetos sobre a programação estruturada:

3 Assinale a alternativa CORRETA:

- a) () As mensagens trocadas entre os objetos são conhecidas como atributos.
- b) () Os atributos e os métodos dos objetos são conhecidos como o ESTADO dos mesmos.
- c) () A programação orientada a objetos exige a utilização de uma máquina virtual que proteja o sistema operacional de possíveis erros de um programa.
- d) () O encapsulamento consiste em proteger os dados de um objeto do acesso externo, liberando-os conforme houver necessidade.

4 Observe as afirmações a seguir classificando-as em verdadeiras (V) ou falsas (F):

- () A linguagem de programação Java teve sua arquitetura baseada na arquitetura da linguagem de programação C++.
- () Para uma linguagem ser considerada Orientada a objetos, ela deve implementar os conceitos de encapsulamento, herança e polimorfismo.
- () A extensibilidade dos objetos facilita seu reuso por outros programadores.
- () A plataforma Java é composta por uma máquina virtual, uma linguagem de programação, um conjunto de bibliotecas e um mecanismo de instalação ou *deployment*.

- () O JRE é o ambiente de execução das aplicações Java, enquanto o JDK reúne um conjunto de ferramentas para desenvolvedores.
 - () O Java é considerado uma linguagem de programação compilada e interpretada. Ela é compilada pelo compilador javac e depois tem seus *bytecodes* interpretados pela JVM.
 - () *Hotspot* é o nome do recurso da JVM para otimizar *bytecode* em tempo de interpretação.
- 5 Descreva as principais diferenças entre um programa compilado e um interpretado.



FERRAMENTAS E INSTALAÇÃO

1 INTRODUÇÃO

No tópico anterior estabelecemos as bases para o estudo da Programação Orientada a Objetos, abordando o histórico, algumas características e a linguagem de programação que utilizaremos no resto da disciplina.

No Tópico 2 preparamos o ambiente que servirá de base para que você possa realizar seus estudos e praticar programação orientada a objetos através de exemplos práticos.

2 JAVA DEVELOPMENT KIT E INTEGRATED DEVELOPMENT ENVIRONMENT

O primeiro passo é a instalação do Java *Development Kit* (JDK), que consiste da máquina virtual Java, da linguagem de programação e das ferramentas necessárias para a construção de aplicações Java. É importante salientar a diferença entre o JDK e o Java *Runtime Environment* (JRE). Enquanto o primeiro traz um conjunto de ferramentas para permitir o trabalho do desenvolvedor, o segundo consiste apenas da máquina virtual, ou seja, o ambiente de execução. Como queremos desenvolver em Java e não somente executar aplicações, a primeira opção é a correta.

Escolha a plataforma adequada, conforme a Figura 14, e proceda com a instalação. Salientamos que a utilização desta ferramenta para fins acadêmicos é totalmente gratuita.



Os procedimentos descritos neste caderno foram feitos em um computador com sistema operacional Windows 7, Service Pack 1. Outras versões do Windows funcionarão praticamente da mesma forma, com pequenas diferenças. Para procedimentos de instalação em sistemas Linux ou MacOS, entre em contato com os professores da disciplina.

FIGURA 14 - SELEÇÃO DE PLATAFORMA PARA O JDK

Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- [Java Developer Newsletter](#) (tick the checkbox under Subscription Center > Oracle Technology News)
- [Java Developer Day hands-on workshops \(free\)](#) and other events
- [Java Magazine](#)

[JDK MD5 Checksum](#)

Looking for JDK 8 on ARM?
JDK 8 for ARM downloads have moved to the [JDK 8 for ARM download page](#).

Java SE Development Kit 8u20

You must accept the [Oracle Binary Code License Agreement](#) for Java SE to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Linux x86	135.24 MB	jdk-8u20-linux-i586.rpm
Linux x86	154.87 MB	jdk-8u20-linux-i586.tar.gz
Linux x64	135.6 MB	jdk-8u20-linux-x64.rpm
Linux x64	153.42 MB	jdk-8u20-linux-x64.tar.gz
Mac OS X x64	209.11 MB	jdk-8u20-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	137.02 MB	jdk-8u20-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	97.09 MB	jdk-8u20-solaris-sparcv9.tar.gz

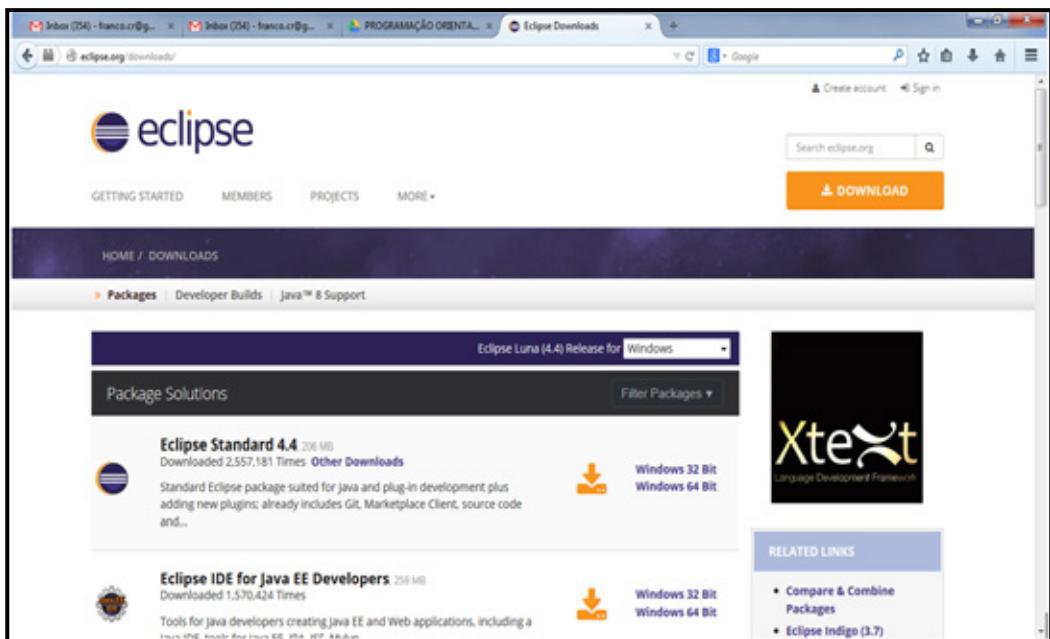
FONTE: Disponível em: <<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>>. Acesso em: 12 set. 2014.

O procedimento de instalação do JDK é simples e, uma vez que ele estiver concluído, podemos instalar o ambiente de desenvolvimento, também conhecido por *Integrated Development Environment* (IDE). A instalação do JDK ANTES da IDE ocorre porque a própria IDE que utilizaremos nesta disciplina foi desenvolvida utilizando a linguagem de programação Java, o que obriga a existência prévia de uma máquina virtual para seu funcionamento. O objetivo principal de uma IDE é auxiliar o desenvolvedor, fazendo com que seu trabalho de programação seja mais produtivo. Entre as características mais comuns das IDEs, podemos destacar:

- Editor de código fonte.
- Compilador.
- *Linker*.
- Ferramenta para modelagem.
- Geração de código fonte.
- Distribuição da aplicação e auxílio na criação do instalador.
- Testes automatizados.
- Refatoração.

Existem diversas IDEs para desenvolvimento Java no mercado, cada uma visando atender a necessidades específicas. Como nesta disciplina, nosso principal objetivo é o aprendizado da Programação Orientada a Objetos, utilizaremos a IDE Eclipse, que pode ser obtida livremente através da URL <<http://www.Eclipse.org/downloads>> (Figura 15).

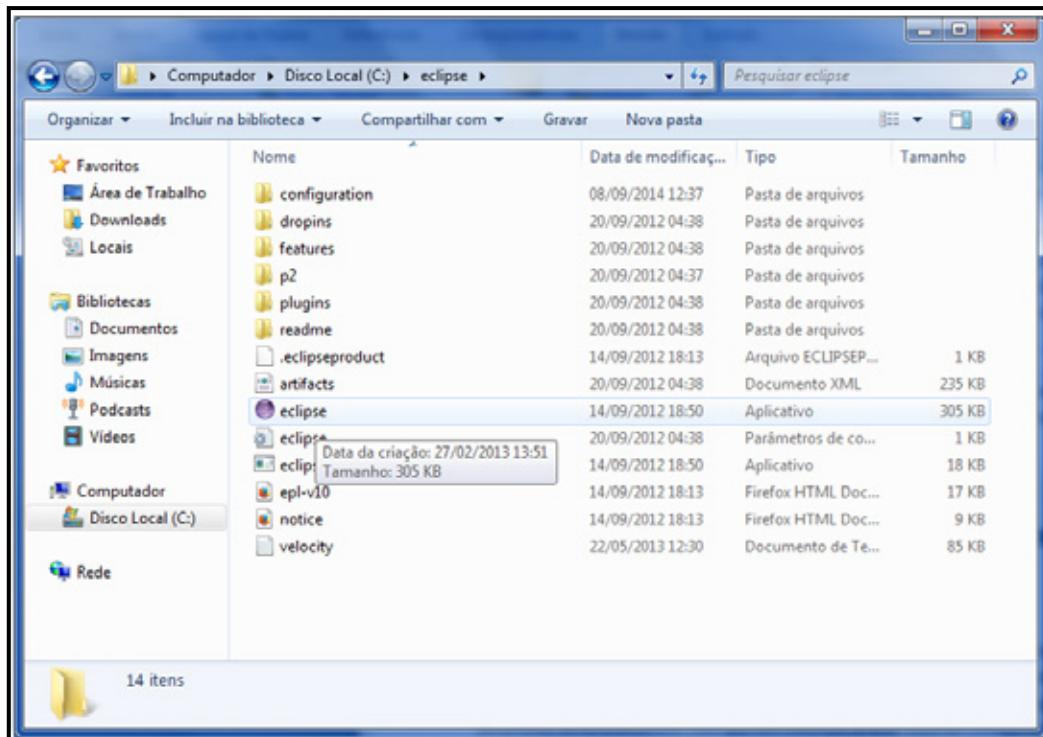
FIGURA 15 - SITE PARA DOWNLOAD DO ECLIPSE



FONTE: Disponível em: <<http://Eclipse.org/downloads/>>. Acesso em: 12 set. 2014.

A instalação do Eclipse consiste simplesmente na descompactação do arquivo *zip* que foi obtido do site. Esta descompactação pode ser feita onde você desejar, visto que os arquivos dos projetos desenvolvidos utilizando a IDE estarão em um local diferente. Após a descompactação, basta clicar duas vezes no arquivo Eclipse (Figura 16), que o mesmo iniciará. Sua primeira inicialização demora um pouco, pois a IDE fará automaticamente algumas configurações de ambiente. As demais inicializações ocorrerão de forma mais rápida, como você perceberá.

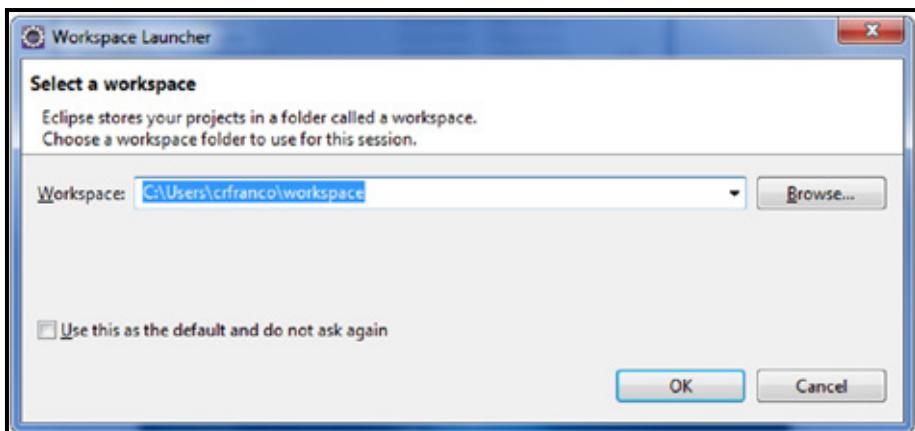
FIGURA 16 - EXECUTÁVEL DO ECLIPSE



FONTE: O autor

Antes de efetivamente abrir, o Eclipse perguntará qual é o local onde deseja colocar o *workspace*, conforme Figura 17. O *workspace* é o local onde seus projetos serão armazenados com todos os códigos fontes pelo Eclipse. Sugerimos fortemente que você selecione um local para o *workspace* que seja de fácil acesso e *backup* constante. Em nosso exemplo, o *workspace* foi colocado dentro da pasta do usuário crfranco, que é o mesmo local onde o Windows armazena os documentos, imagens, vídeos etc. Caso você deseje que esta tela de configuração não apareça mais, basta clicar no *checkbox* da parte inferior, onde está escrito: *Use this as the default and do not ask again*. Ao selecionar esta opção, o Eclipse entende que sempre que abrir, o *workspace* será aquele que já estiver configurado.

FIGURA 17 - LOCAL DO WORKSPACE



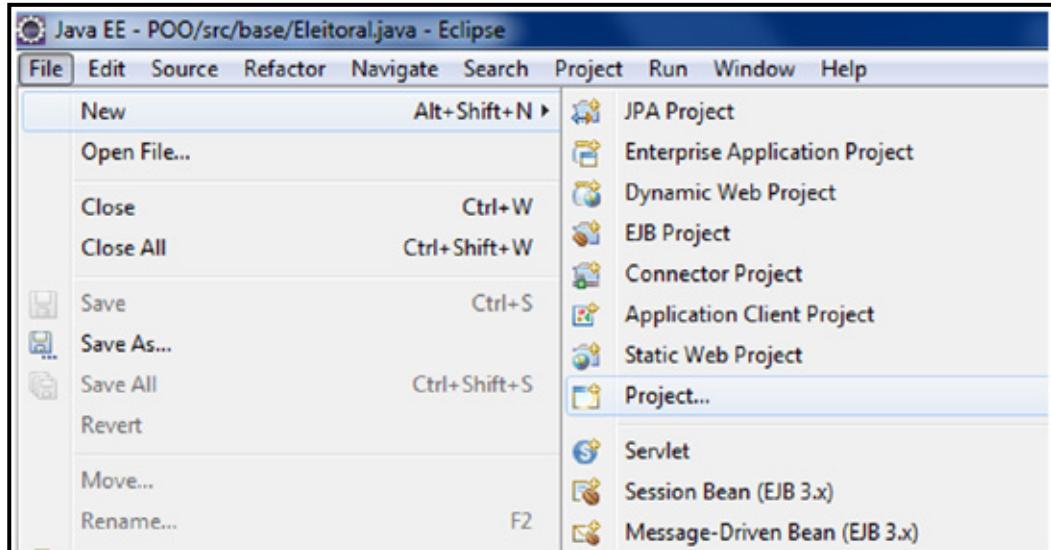
FONTE: O autor

2.1 CRIANDO UM PROJETO

E então, vamos colocar as mãos à obra? Agora instruiremos você como criar um projeto dentro do Eclipse. O primeiro conceito a ser entendido é que tudo que acontece dentro do Eclipse, acontece dentro de um projeto. Mesmo que você queira somente uma classe para demonstrar determinado conceito e não use mais de 10 linhas de código, esta classe deverá ser criada dentro de um projeto.

O Eclipse mostra uma tela de boas-vindas, que pode ser ignorada e fechada neste momento. Para criarmos nosso primeiro projeto, vamos selecionar a opção no menu do Eclipse: **File -> New -> Project**, conforme destacado pela figura 18.

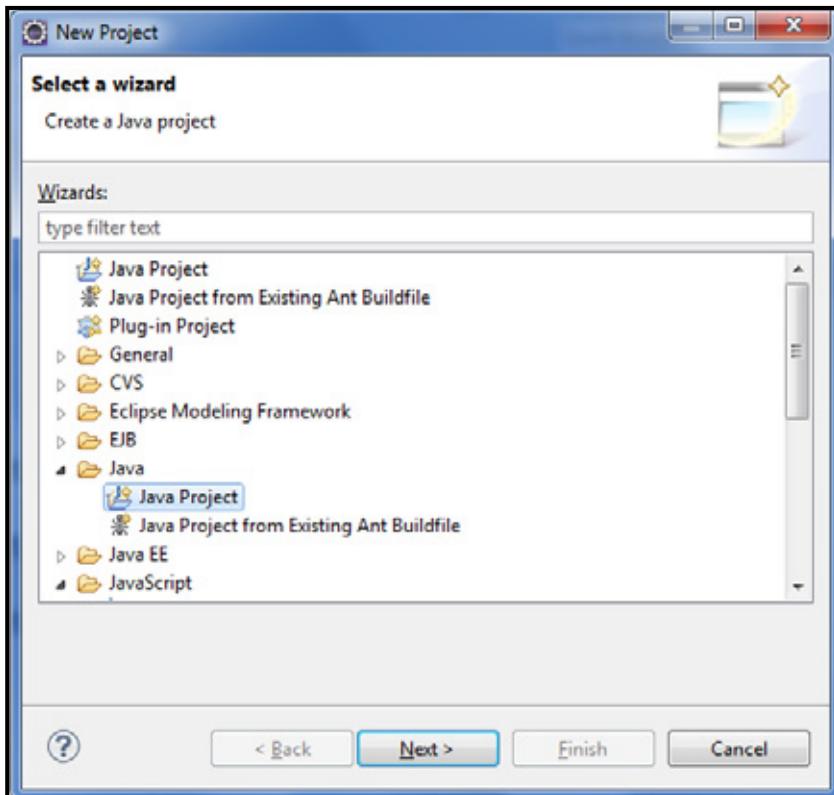
FIGURA 18 - CRIAÇÃO DE NOVOS PROJETOS NO ECLIPSE



FONTE: O autor

Selecionada esta opção surge um diálogo questionando qual é o tipo de projeto que queremos criar, conforme a Figura 19. Devemos sempre selecionar a opção *Java Project*, no contexto dos objetivos desta disciplina.

FIGURA 19 - SELEÇÃO DO TIPO DE PROJETO JAVA

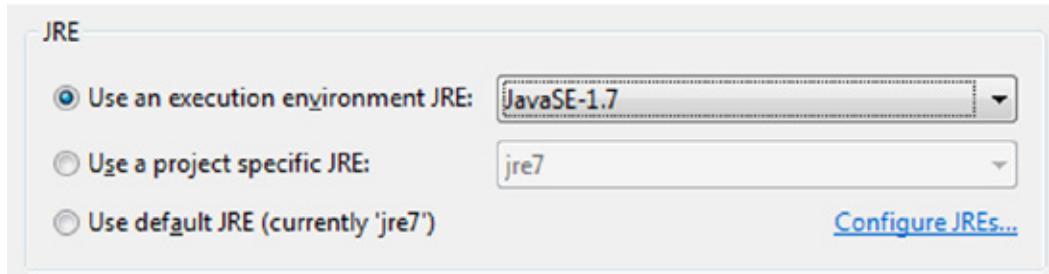


FONTE: O autor

O próximo diálogo que surge apresenta três informações importantes para o seu projeto, conforme destacado a seguir:

- 1) *Project Name*: Neste espaço você colocará o nome de seu projeto. Como este é nosso primeiro projeto, sugerimos o nome PRIMEIRO, todo em letras maiúsculas. Outra opção importante é a localização do *Workspace*, que ainda pode ser alterada antes da criação do projeto. Sugerimos que você não mexa na localização do *Workspace*.
- 2) *JRE*: Aqui você seleciona qual versão do Java será utilizada para compilar e executar este projeto. Você pode selecionar a Versão 1.8 ou 1.7 do JavaSE, conforme ilustrado na Figura 20.
- 3) *Project Layout*: Nesta opção o Eclipse permite a configuração dos diretórios do projeto em disco, separando os arquivos de código-fonte dos arquivos binários. Recomendamos que você mantenha a opção *Create separate folders for sources and class files*, pois mantém o projeto mais organizado.

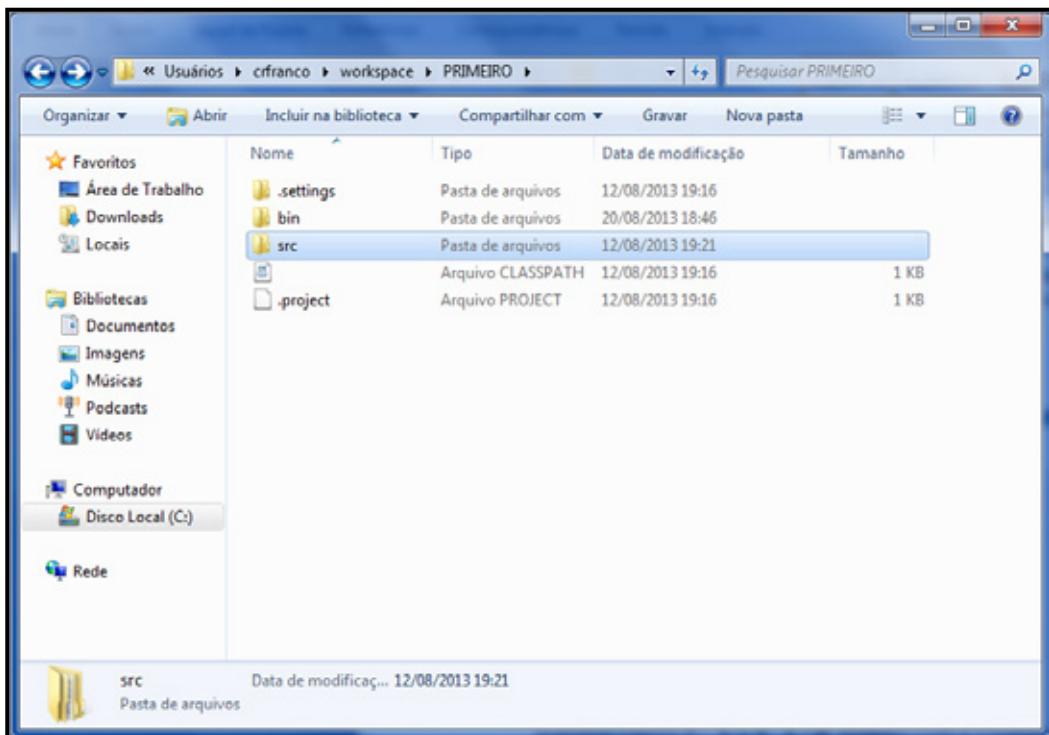
FIGURA 20 - OPÇÃO DE JRE PARA O PROJETO



FONTE: O autor

Após selecionar estas opções, basta clicar no botão *Finish* e deixar o Eclipse preparar o projeto para você. Em seu disco rígido, foi criada uma pasta com o nome PRIMEIRO dentro de seu *Workspace*, conforme Figura 21. Por hora basta você saber que os subdiretórios bin e src são responsáveis por conter os arquivos com extensão .class (binários) e os arquivos com extensão .java (códigos fonte), respectivamente.

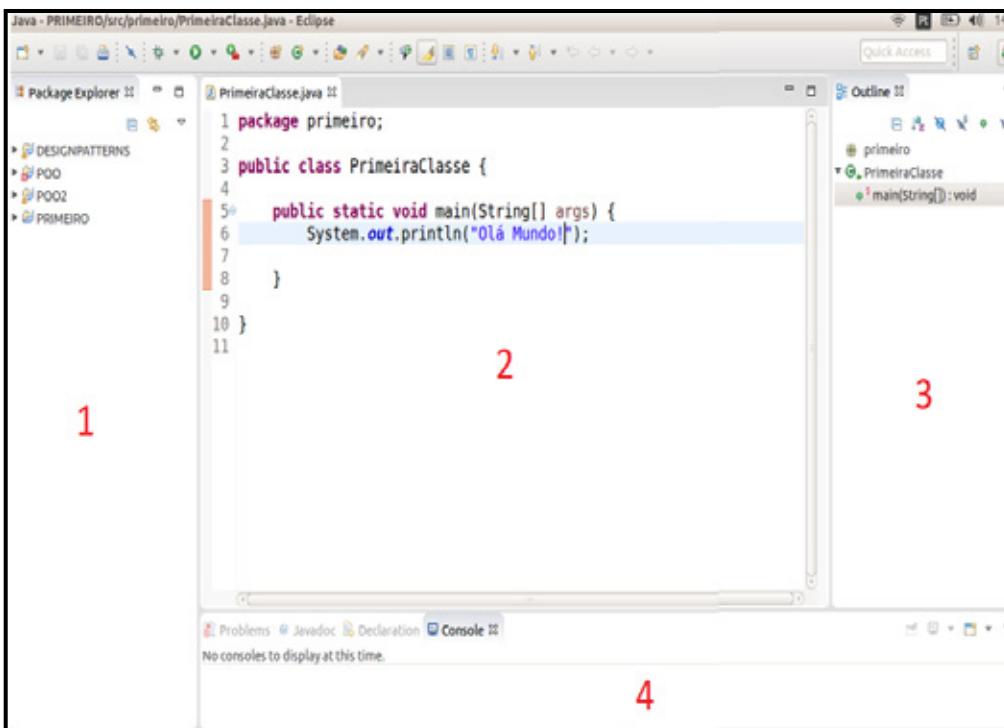
FIGURA 21 - ESTRUTURA DO PROJETO CRIADO PELO ECLIPSE



FONTE: O autor

Nesta disciplina abordaremos somente uma pequena parte das funcionalidades que a IDE Eclipse apresenta, no sentido de permitir que você consiga criar e testar seus próprios projetos. A Figura 22 destaca alguns dos elementos mais importantes do ambiente:

FIGURA 22 - IDE ECLIPSE



FONTE: O autor

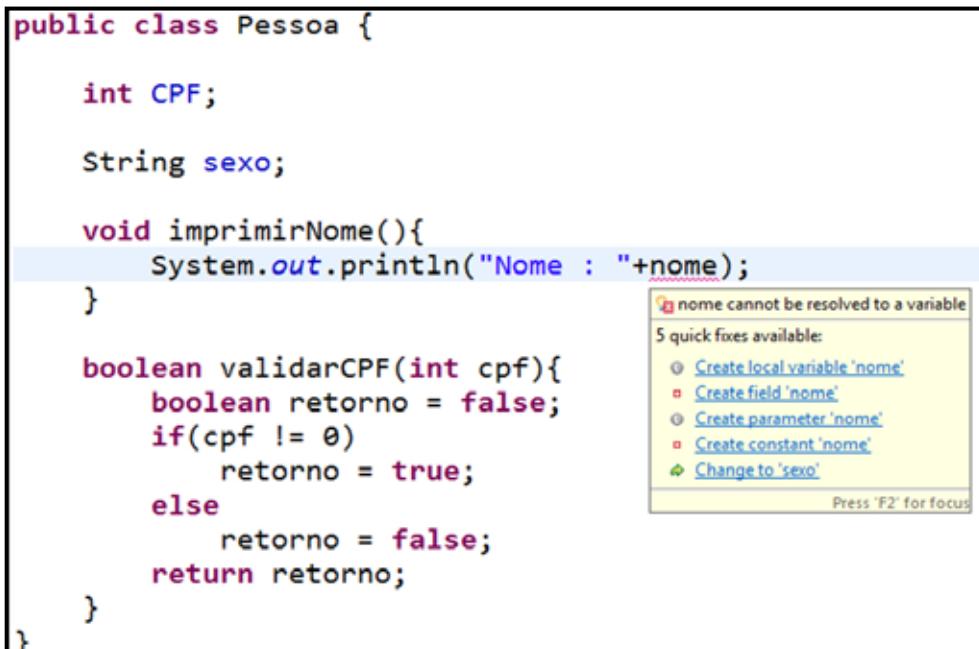
No lado esquerdo, destacado pelo número 1, está o *Package Explorer*. A função desta ferramenta é possibilitar a você uma visualização de forma hierárquica de todos os projetos, pacotes, classes e bibliotecas que estiverem em seu *Workspace*. A navegação neste ambiente é similar à navegação em qualquer navegador de arquivos, como o *Windows Explorer*, por exemplo. Conforme formos avançando na disciplina, mais familiarizado você ficará com a utilização do *Package Explorer*.

No centro do ambiente, destacado pelo número 2, está o editor de código fonte, onde efetivamente faremos a escrita das classes. O editor do Eclipse apresenta inúmeras características para auxiliar a escrita do código-fonte entre as quais podemos destacar:

- 1) Diferenciação entre palavras reservadas da linguagem e variáveis criadas pelo usuário.
- 2) Ajuda de contexto para descobrirmos quais métodos e atributos de um objeto estão disponíveis para utilização.
- 3) Compilação em tempo de interpretação. Basta que um arquivo de código fonte seja salvo para que o Eclipse proceda com sua compilação e mostre imediatamente erros que possam existir.

- 4) Sugestão de correção. Ao encontrar um erro de compilação no código, o Eclipse inicialmente marca o erro sublinhando-o em vermelho, conforme pode ser visto no Quadro 4. Neste caso, o Eclipse está nos dizendo que a variável nome não existe na classe e sugere que a criemos. Ao selecionar qualquer uma das opções, o Eclipse já procede com a correção, executando a ação escolhida. Apesar de esta ser uma funcionalidade extremamente poderosa, devemos utilizá-la com cuidado, pois uma ação errada pode levar a diversos erros de difícil correção posterior.
- 5) Documentação *on-line*. Ao colocar o *mouse* sobre determinada Classe ou método, automaticamente a documentação da linguagem aparece na tela, facilitando em muito o trabalho do desenvolvedor.
- 6) Intelisense. Ao digitar parte de um comando, o Eclipse mostra diversas opções para automaticamente completar o que está faltando, sugerindo inclusive nomes para variáveis com base em parâmetros definidos há métodos.

QUADRO 4 - SUGESTÃO DE CORREÇÃO DE ERROS NO ECLIPSE



The screenshot shows a Java code editor with the following code:

```

public class Pessoa {

    int CPF;

    String sexo;

    void imprimirNome(){
        System.out.println("Nome : "+nome);
    }

    boolean validarCPF(int cpf){
        boolean retorno = false;
        if(cpf != 0)
            retorno = true;
        else
            retorno = false;
        return retorno;
    }
}

```

A tooltip window is open over the word "nome" in the line "System.out.println("Nome : "+nome);". The tooltip says: "nome cannot be resolved to a variable" and lists 5 quick fixes available:

- Create local variable 'nome'
- Create field 'nome'
- Create parameter 'nome'
- Create constant 'nome'
- Change to 'sexo'

At the bottom right of the tooltip, it says "Press 'F2' for focus".

FONTE: O autor

Na parte direita da IDE está o *Outline* (número 3), ferramenta que tem a mesma função do *Package Explorer*, só que atuando em classes. Ao editar uma classe no Eclipse, o *Outline* imediatamente mostra suas características e comportamentos de forma hierárquica, permitindo que se tenha uma visualização mais ampla do que se está fazendo.

Já na parte inferior da tela, destacado pelo número 4, está o *Console*. Através desta ferramenta é possível fazer entrada de dados via teclado e visualização de resultados através da saída padrão, ou seja, é possível observar os resultados dos testes de nossos programas.



Por não ser o foco principal da disciplina, abordaremos somente funcionalidades básicas do Eclipse. Caso você queira se aprofundar na ferramenta, sugerimos o site <<http://eclipse.org/users/>>, onde você encontrará mais documentação sobre a mesma.

2.2 CRIANDO UMA CLASSE

Conforme aprendemos no Tópico 1, na Programação Orientada a Objetos, os programas funcionam através da existência de objetos distintos e interação destes entre si. **Para termos objetos, necessariamente precisamos das classes.** Está pronto para a criação de sua primeira classe? Então, vamos lá!

Inicialmente vamos falar sobre um aspecto importante da programação através da linguagem Java: os pacotes. Pacotes são subdiretórios que você pode criar em seu projeto com diversas finalidades. Uma das principais finalidades para a qual os pacotes são usados é a separação das classes por responsabilidade. Digamos que você tenha um *software* que utilize classes para a geração de telas e classes para conexão e operação com bancos de dados. Você poderia colocar as classes de tela em um pacote com o nome *gui* (de *graphic user interface*) ou qualquer outro nome significativo, enquanto as classes de banco de dados ficariam em um pacote chamado *banco* ou *dados*.

Outra motivação para a utilização dos pacotes é a possibilidade de **duplicação de nomes de classes**. Não é incomum um projeto Java conter dezenas e até mesmo centenas de classes, desenvolvidas por equipes diferentes que podem inclusive estar separadas geograficamente. Digamos que exista uma equipe responsável pela criação das classes que cuidam da conexão com redes TCP/IP e esta equipe crie uma classe com o nome *conexão*. Em outra localidade, a equipe que cuida da parte responsável pela conexão com o banco de dados cria uma classe com o mesmo nome. Neste caso, sem a existência de pacotes, teríamos um erro de compilação, pois o Java não permite a existência de duas classes com o mesmo nome em um mesmo projeto. **Para exemplificar de que forma resolvériamos a situação acima com a utilização de pacotes, a equipe que trabalha nas classes de rede as colocaria em um pacote chamado *rede*, enquanto a equipe do banco de dados criaria um pacote chamado *dados*.** A existência destes pacotes

torna possível a identificação única das classes através do que é conhecido como *fully qualified name*. Por meio desta prática, a classe de rede seria conhecida como rede. Conexão e a classe de dados apareceriam como dados. Conexão. A própria linguagem Java é repleta destes tipos de exemplo, como a classe *Rectangle*, que está disponível nos pacotes *java.awt* e *graphics*. O Quadro 5 mostra algumas boas práticas para a criação de nomes para pacotes:



Para saber mais sobre as regras e boas práticas de nomenclatura para pacotes, sugerimos o site: <<http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>>.

QUADRO 5 - REGRAS DE NOMENCLATURA PARA PACOTES

- Nomes de pacotes são escritos em letra minúscula, para evitar conflitos com nomes de classes ou interfaces.
- As empresas em geral utilizam seu nome de domínio na internet para iniciar seus nomes de pacotes, por exemplo: com.exemplo.meupacote para a empresa cujo domínio for exemplo.com.
- Todos os pacotes da linguagem Java começam com java ou javax.

FONTE: Disponível em: <<http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>>. Acesso em: 12 set. 2014.

Para que possamos aderir às boas práticas de desenvolvimento recomendadas pela *Oracle*, criaremos um pacote para colocar nossas classes. No *Package Explorer* (*package* significa pacote em inglês), clique no triângulo que aparece do lado esquerdo do nome do projeto. A partir daí, você deve ver o diretório *src*, onde ficarão armazenados os pacotes e arquivos de código fonte. Clique com o botão direito do *mouse* no diretório *src* e escolha opção *New->Package*. Coloque o nome *primeiro* para o pacote e clique no botão *Finish*. A partir deste momento, nosso projeto já conta com a estrutura mostrada na Figura 23.

FIGURA 23 - ESTRUTURA DO PROJETO

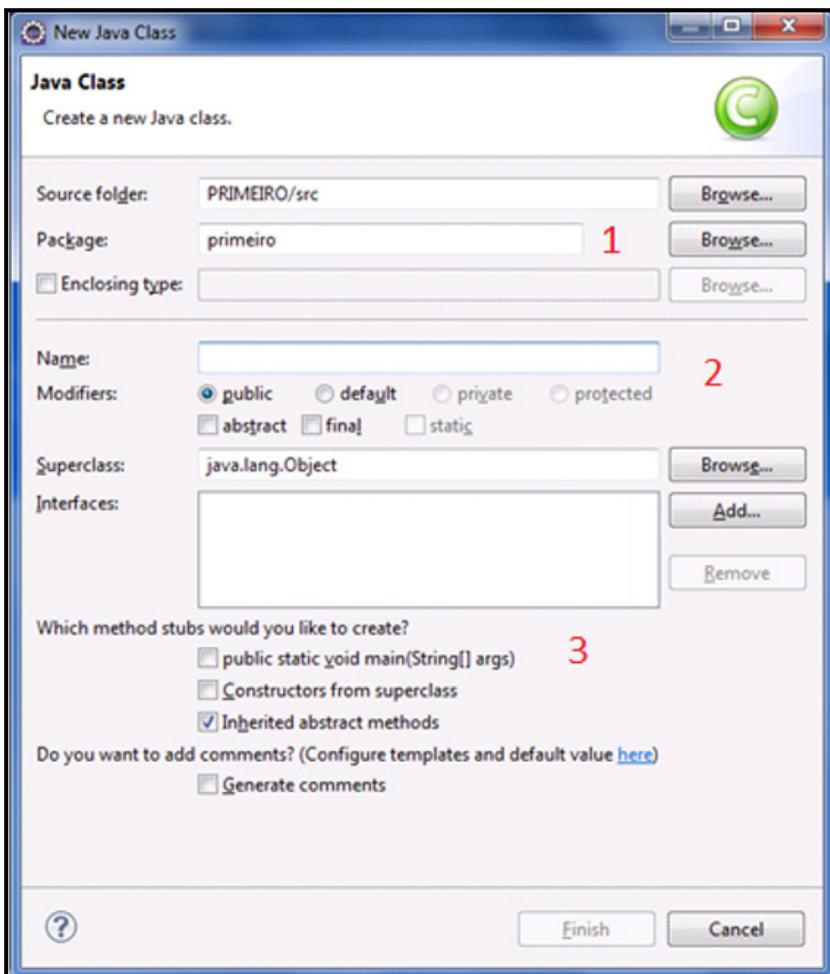


FONTE: O autor

Agora clique com o botão direito do *mouse* sobre o pacote *primeiro* e escolha a opção *New->Java Class*. A partir desta ação surge um diálogo com duas seções principais para observarmos (Figura 24):

- 1) *Source Folder* e *Package*. Estes dois campos já virão preenchidos com as informações respectivas, caso a criação da classe seja feita na sequência que mostramos acima. Esta seção está destacada pelo número 1.
- 2) Nome da classe e demais configurações. Por hora ignoraremos os demais campos e preencheremos somente os campos *Name* e o *check box* com a opção *public static void main (String[] args)*. O nome que daremos para a classe será *Primeira Classe* e selecionaremos a opção do *check box* descrita acima para possibilitarmos a execução do programa e consequente visualização de algum resultado. Estas opções estão destacadas pelos números 2 e 3.

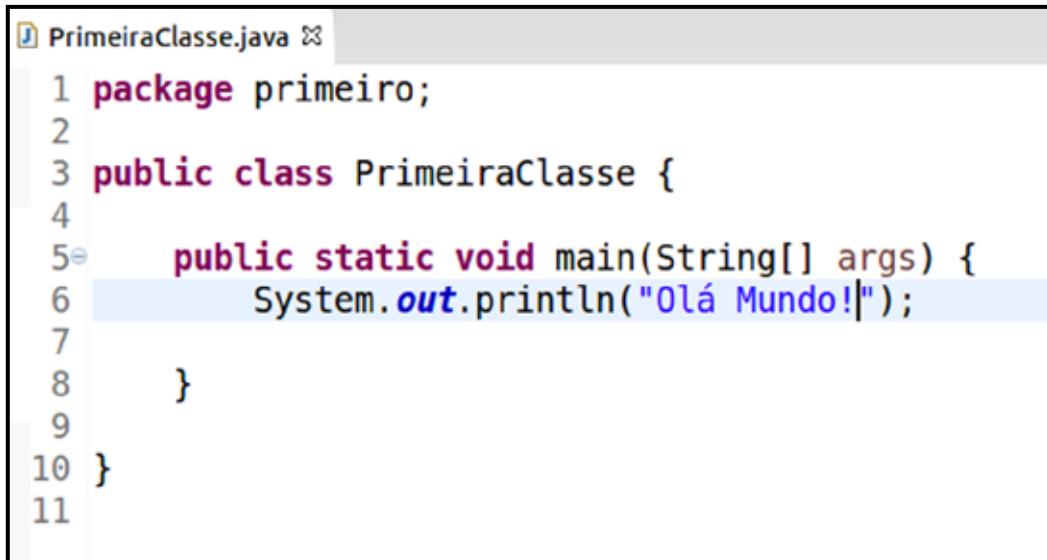
FIGURA 24 - CRIAÇÃO DE CLASSES NO ECLIPSE



FONTE: O autor

Uma vez pressionado o botão *Finish*, o Eclipse procede com a criação da classe que aparece no editor de código fonte. Para compilar o programa basta pressionar o atalho **ctrl+s** ou clicar no ícone salvar na parte superior da tela. Nossa primeira classe terá a importante função de imprimir na saída padrão a mensagem “Olá mundo!” Para realizarmos isso, basta você digitar a linha de código que aparece no Quadro 6. Momentaneamente ignoraremos o significado desta e das demais linhas de código da classe, uma vez que o objetivo do exemplo é aprender a criar projetos, pacotes, classes e colocá-los para executar. Lembre-se de que o Java é *case sensitive*, ou seja, palavras iguais escritas com letras maiúsculas e minúsculas são diferentes. Ao terminar de copiar o código fonte, salve o arquivo para que ele seja compilado.

QUADRO 6 - PRIMEIRA CLASSE EM JAVA



```

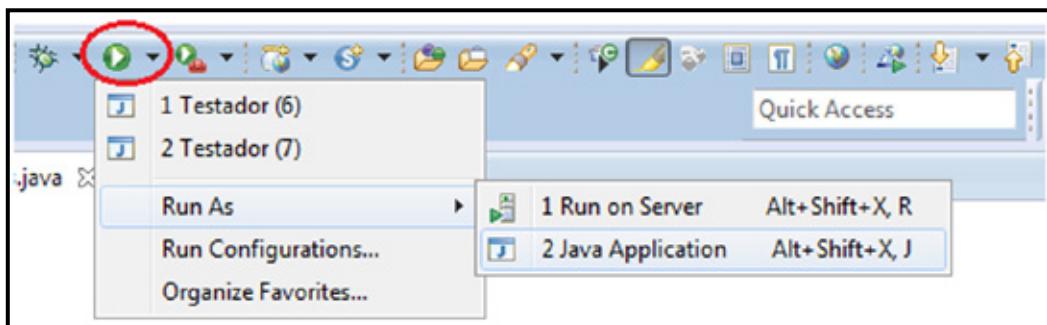
1 package primeiro;
2
3 public class PrimeiraClasse {
4
5     public static void main(String[] args) {
6         System.out.println("Olá Mundo!");
7     }
8 }
9
10 }
11

```

FONTE: O autor

Para executar esta classe e visualizar o resultado, devemos clicar no botão destacado na Figura 25 e selecionar as opções *Run As->Java Application*. A partir desta ação o programa executa e a mensagem é exibida na saída padrão (*console*). Parabéns! Você escreveu e executou seu primeiro programa em Java!

FIGURA 25 - EXECUÇÃO DE PROGRAMAS NO ECLIPSE



FONTE: O autor

No próximo tópico deste caderno, destacaremos a criação de classes com atributos e métodos e a instanciação destas classes de forma a visualizar seu estado e comportamento em tempo de execução.

RESUMO DO TÓPICO 2

Neste tópico vimos:

- Para criar programas utilizando a linguagem de programação Java, é necessária a instalação do JDK apropriado ao seu sistema operacional.
- Para que o trabalho de programação seja mais produtivo, é recomendado o uso de uma IDE, ferramenta que dispõe de diversos recursos para auxiliar o programador.
- Existem diversas IDEs no mercado para desenvolver em Java, destacando-se o Eclipse e o NetBeans.
- Os packages são utilizados para agruparem as classes de acordo com sua responsabilidade dentro da aplicação.
- Para evitar confusão com nomes de classes, os pacotes devem ser nomeados sempre com letras minúsculas.
- Outra vantagem da utilização de packages é a possibilidade de existência de classes com o mesmo nome, uma vez que o compilador considera o fully qualified name para esta verificação.
- O fully qualified name de uma classe é simplesmente o nome da classe antecedido do nome do(s) pacote(s) onde ela estiver contida. O fully qualified name da classe Rectangle pode ser java.awt.Rectangle ou graphics.Rectangle.
- Para criarmos nossas classes no Eclipse, elas devem necessariamente estar contidas em um projeto.
- Classes devem ser nomeadas usando substantivos simples ou compostos que sempre começam com letra maiúscula. Telefone, Teclado, Estudante, Gerador de Telas são bons nomes para classes.

AUTOATIVIDADE



- 1 De acordo com os procedimentos vistos neste tópico, crie um projeto no Eclipse chamado de Exercício, contendo um pacote chamado topico1:
- 2 Dentro do projeto criado na questão 1, crie mais dois pacotes, chamados topico2 e topico3.
- 3 Dentro dos pacotes topico1 e topico3 crie uma classe chamada de Classe Teste e dentro do pacote topico2, crie um subpacote chamado de topico2.1. Dentro de topico2.1 crie outra classe chamada Classe Teste.
- 4 Nas linhas a seguir escreva o fully qualified name de todas as classes existentes no projeto Exercício.

- 5 Encontre o diretório do projeto dentro de seu workspace e liste todos os subdiretórios do projeto nas linhas a seguir:

- 6 Qual é o objetivo da existência dos diretórios bin e src?



CLASSES E OBJETOS

1 INTRODUÇÃO

Entre os principais tópicos necessários para a compreensão da programação orientada a objetos destacam-se os conceitos de classe e objeto. O perfeito entendimento sobre a maneira através da qual as classes moldam a estrutura dos objetos tornará menos complexo o trabalho de analisar os problemas computacionais através desse novo paradigma de programação.

Neste tópico abordaremos de forma prática a criação das classes e a instanciação de objetos através de métodos construtores. Com vistas ao conceito de objetos, faremos ainda a utilização de atributos e a invocação de métodos.

Todos os conceitos serão exemplificados através de código fonte, o que necessariamente implica seu entendimento sobre os procedimentos básicos de criação de projetos e classes na IDE Eclipse.

2 CRIANDO CLASSES

No tópico anterior fizemos a criação de uma classe simplesmente para demonstrar o funcionamento básico da IDE na edição e execução de código. A partir deste tópico todas as nossas classes serão criadas com um objetivo e responsabilidade bem definidos, como pregam as boas práticas da programação orientada a objetos.

Nossa primeira classe se chamará Pessoa e possuirá os atributos CPF, nome e sexo. É importante destacar que os nomes escolhidos para as classes sempre devem ser substantivos simples ou compostos iniciados com letra maiúscula. Nomes como Animal, Veículo, Cidade, Aluno, Folha de Pagamento, Conta Corrente são exemplos de bons nomes para classes. Devemos sempre ter em mente que as classes serão utilizadas para instanciar os objetos de nosso *software*, e que deverão representar conceitos de negócio ou mesmo conceitos de tecnologia aplicados ao negócio. Caso uma classe seja responsável por gerar relatórios dentro do sistema, ao invés de chamá-la de Gerar Relatórios, um nome mais adequado seria Gerador de Relatórios, pois este último é um substantivo.

Nome de classe = Sempre substantivos

No Quadro 7 está o código fonte da classe Pessoa até o momento. Na linha 1 está a declaração do pacote onde criamos a classe. A declaração do pacote, caso exista um, é sempre a primeira linha de um arquivo de código fonte Java. Na linha 2 está a declaração da classe, sendo que o nome da classe será sempre o nome do arquivo físico no disco rígido. Para comprovar este fato, basta ir ao diretório src dentro de seu projeto. Elucidaremos o significado da palavra *public* na próxima unidade. Ao final da linha 3 está o caractere chave, ({), que representa o início de um bloco. Todo início de bloco é completado obrigatoriamente no final de bloco, representado neste caso pelo caractere da linha 9 (}). Podemos dizer que o escopo da classe Pessoa vai da linha 3 até a linha 9, e que todos os comandos e declarações entre o início e o final do bloco pertencem a ela.

QUADRO 7 - CLASSE PESSOA

```

1 package primeiro;
2
3 public class Pessoa {
4
5     int CPF;
6     String nome;
7     String sexo;
8
9 }
```

FONTE: O autor

Nas linhas 5, 6 e 7 estão declarados os atributos da classe. Os atributos representam as informações que a classe deve armazenar internamente e que variam de objeto para objeto, pois cada pessoa que representarmos terá valores distintos para o conjunto CPF, nome e sexo.

A linguagem de programação Java foi construída respeitando o conceito conhecido como tipagem forte, onde toda declaração de variável deve, obrigatoriamente, incluir o tipo da mesma, ao contrário de linguagens como PHP e Ruby, por exemplo. Nas linhas 5, 6 e 7 podemos perceber que o CPF é do tipo inteiro, enquanto o nome e o sexo são do tipo *String*. Quando instanciarmos um objeto, os valores que este assumir para os atributos são chamados de **Estado** do objeto.



Caso você esteja com dificuldades na sintaxe da linguagem de programação Java, a leitura complementar desta unidade traz um guia de referência.

Não existe limitação no número e tipo de atributos de uma classe, como veremos nos exemplos mais avançados. Um detalhe importante a ser relembrado, é que uma vez que criamos a classe Pessoa, esta vira um tipo que pode ser utilizado a partir de então em nosso programa.

Uma das diferenças mais significativas entre a programação orientada a objetos e os outros paradigmas de programação, é que na programação orientada a objetos ocorre a união entre as informações a serem armazenadas pelos objetos e as operações a serem realizadas com essas informações. Complementaremos o código fonte da classe pessoa com as operações que Pessoa pode realizar (também conhecidas como métodos), conforme demonstrado no Quadro 8.

QUADRO 8 - CLASSE PESSOA

```

1 package primeiro;
2
3 public class Pessoa {
4
5     int CPF;
6     String nome;
7     String sexo;
8
9     void imprimirNome(){
10         System.out.println("Nome : "+nome);
11     }
12
13     boolean validarCPF(int cpf){
14         boolean retorno = false;
15         if(cpf != 0)
16             retorno = true;
17         else
18             retorno = false;
19         return retorno;
20     }
21 }
22

```

FONTE: O autor

Nas linhas 9 e 13 estão declarados os dois métodos implementados na classe Pessoa. Na linha 9 está a assinatura do método imprimirNome, onde podemos perceber, através da palavra reservada *void*, que este método não possui retorno. Um método pode ou não retornar informações, de acordo com a definição do desenvolvedor. Qualquer tipo de dados pode ser utilizado para retorno, inclusive as classes que você criou. Os parênteses vazios indicam que o método não recebe parâmetro quando chamado. Da mesma forma que no escopo da classe, os caracteres { e } indicam o início e o fim do método. A função deste método é simplesmente imprimir o valor contido no atributo nome do objeto em questão.

Na linha 13 está a assinatura do método validarCPF, onde podemos perceber o retorno de um *boolean* e o recebimento de um parâmetro do tipo *int*. Na linha 14 declaramos uma variável local, cujo escopo de existência restringe-se ao método validarCPF. Mas o que isso significa na prática? Significa que caso fizéssemos uma chamada à variável *retorno* fora do escopo deste método, o Eclipse nos traria um erro de compilação, informando que a variável não existe. A lógica para validação do CPF é simples e somente serve para demonstrar de que forma um método seria implementado, sendo inadequada para utilização em um programa real. Todo método que apresenta retorno diferente de *void* exigirá a colocação de pelo menos um comando *return*, conforme apresentado na linha 19. Qualquer tipo de dados pode servir como retorno, inclusive classes recém-criadas. Ao conjunto de métodos de uma classe e seus objetos damos o nome de comportamento. Uma classe pode ter quantos métodos quiser, com quaisquer tipos de retorno.

Uma ressalva é feita para as classes que possuem muitos atributos e métodos, pois em geral essa prática fere o conceito de coesão ou SRP (*Single Responsibility Principle*). Por este princípio, cada classe deve ter uma única responsabilidade dentro do programa, ou seja, cada classe somente existe por uma razão. Em função disso, normalmente as classes não apresentam um número muito grande de atributos ou métodos e os métodos são geralmente curtos. Para exemplificarmos o conceito, imagine uma classe em sua aplicação que trate das conexões com o banco de dados e com as instruções relativas a operações CRUD. Esta classe estaria com responsabilidades demais, ferindo o conceito de SRP. Neste caso, o correto seria separar as responsabilidades em pelo menos duas classes distintas, uma para controlar as conexões com o banco de dados e outra para gerenciar as instruções de operações CRUD.



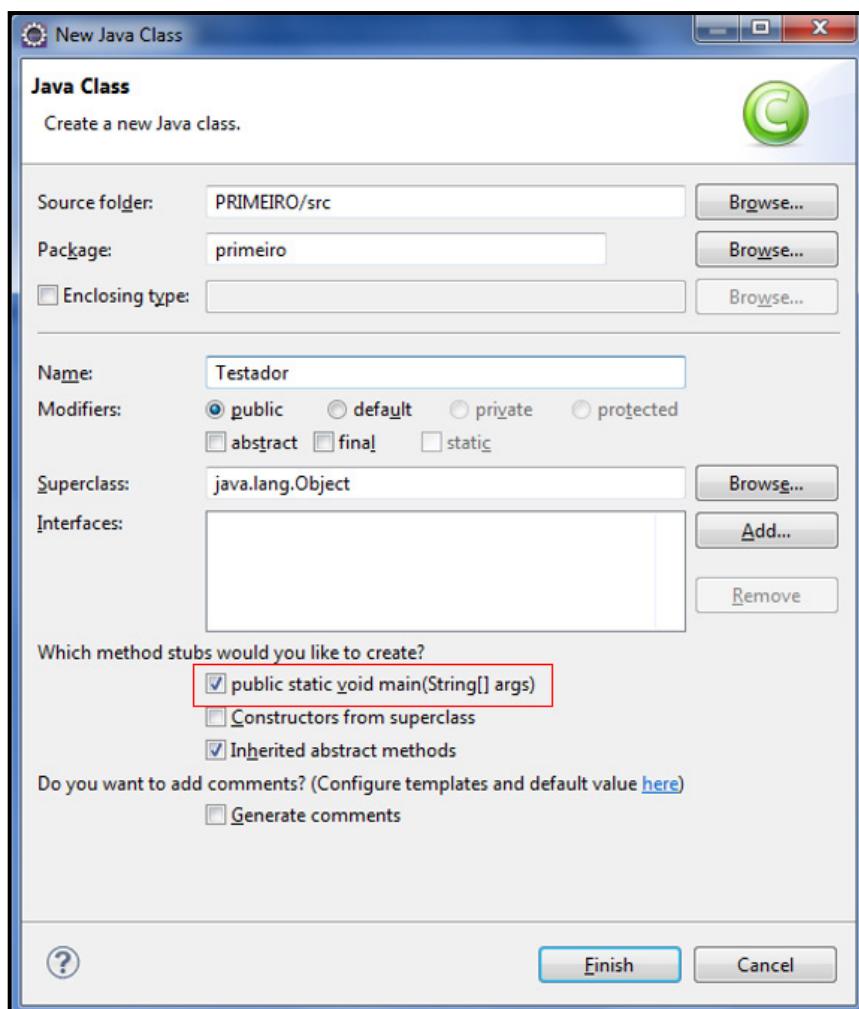
Você sabia que o acrônimo CRUD é utilizado para se referir a operações de criação (create), leitura (read), atualização (update) e exclusão (delete) de informações em softwares?

A partir deste momento, nosso programa poderá fazer uso da classe Pessoa como se fosse um tipo de dados já presente na linguagem, tendo a certeza de que todos os objetos serão criados com o estado e comportamento definidos na classe.

3 CRIANDO OBJETOS

Antes de passarmos para a criação efetiva de objetos, cabe um destaque para o método que é considerado como o ponto de entrada das aplicações em Java para desktop. Uma aplicação Java pode conter dezenas e até mesmo centenas de classes e seu fluxo de execução consiste na instanciação de objetos e troca de mensagens entre estes. Mas quem controla esse fluxo? A própria aplicação, com a ressalva de que UMA classe deve ser escolhida para iniciar o processo. A Figura 26 e o Quadro 9 mostram como criar este ponto de entrada e de que forma o código-fonte gerado aparecerá no Eclipse, respectivamente.

FIGURA 26 - CRIAÇÃO DE CLASSES COM MÉTODO MAIN NO ECLIPSE



FONTE: O autor

QUADRO 9 - MÉTODO MAIN

```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11     }
12
13 }
```

FONTE: O autor

O Eclipse gera o esqueleto do método na classe que escolhemos para ser o ponto de entrada da aplicação, bastando para isso selecionar o *Check Box* destacado pelo retângulo vermelho na Figura 26. Em nossos exemplos iniciais, utilizaremos o ponto de entrada como uma maneira de testar o código que escrevemos nas outras classes. Por hora, o que é importante você saber sobre o método *main*, é que o mesmo representa o ponto de entrada em aplicações para JavaSE (JEE, e JME se comportam de forma diferente) e que este pode receber um vetor de *Strings* como parâmetro.

Uma vez que o método *main* e a classe que queremos testar estão prontos, podemos proceder com os testes. De forma a propiciar uma melhor compreensão para você, as etapas serão indicadas pelas linhas de código do Quadro 10.

QUADRO 10 - CLASSE TESTADOR

```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         Pessoa p;
10        int contador;|
11
12    }
13
14 }
```

FONTE: O autor

Na linha 9, fazemos a declaração da variável onde a referência para o objeto do tipo Pessoa será armazenada. Lembre-se de que a partir do momento que a classe Pessoa ficou pronta, o Java a entende como um tipo de dados da linguagem. Para ilustrar este conceito, na linha 10 trazemos a declaração de um inteiro. Perceba que o procedimento para a declaração dos dois tipos de variável é o mesmo, onde o tipo da variável procede ao nome escolhido para a mesma. No exemplo em questão, escolhemos a letra p, mas poderíamos ter escolhido qualquer outro nome de variável, desde que este respeitasse as regras da linguagem.

Dando continuidade ao nosso exemplo, no Quadro 11, trazemos a instanciação do objeto propriamente dito (linha 11) e a colocação de valores fictícios em seus atributos (linhas 11 a 14). O comando `new` garante que agora temos em memória um objeto do tipo Pessoa, referenciado pela variável p. Isso significa que somente temos acesso aos atributos deste objeto através de p. Essa restrição fica evidente quando verificamos de que forma o CPF, o nome e o sexo são atribuídos para o objeto. O comando `p.nome` implica que buscaremos o endereço referenciado na memória por p e que leremos ou escreveremos algum valor para nome. Lembre-se de que TODOS os objetos que forem do tipo Pessoa compartilham estado e comportamento.

QUADRO 11 - INSTANCIACÃO DO OBJETO

```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         Pessoa p;
10
11         p = new Pessoa();
12         p.CPF = 12345323;
13         p.nome = "Joshua Bloch";
14         p.sexo = "Masculino";|
15     }
16 }
17
18 }
```

FONTE: O autor

QUADRO 12 - INSTANCIAMENTO DE UM SEGUNDO OBJETO

```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         Pessoa p;
10
11         p = new Pessoa();
12         p.CPF = 12345323;
13         p.nome = "Joshua Bloch";
14         p.sexo = "Masculino";
15
16         Pessoa p1 = new Pessoa();
17         p1.CPF = 2334333;
18         p1.nome = "James Gosling";
19         p1.sexo = "Masculino";
20     }
21 }
```

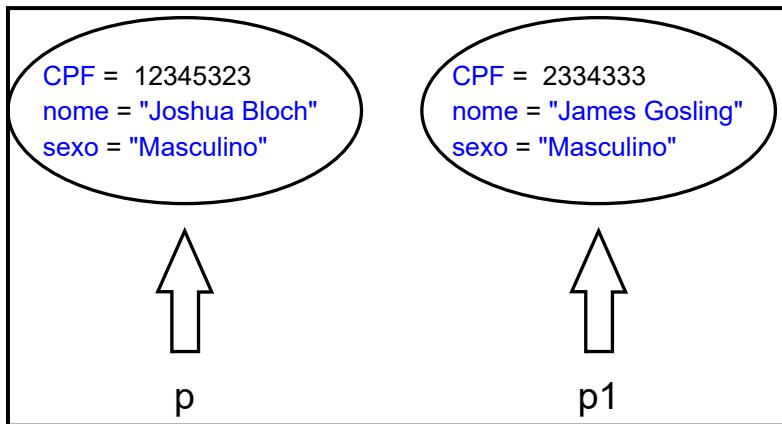
FONTE: O autor

Entre as linhas 16 a 19 do Quadro 12, fazemos a criação de um segundo objeto do tipo Pessoa. O procedimento é praticamente igual ao da criação do primeiro objeto, havendo diferença somente na linha 16, pois neste exemplo declaramos e instanciamos o objeto em uma linha só. Na prática, para o compilador, os dois códigos ficarão idênticos, havendo variação somente para os valores dos atributos.

Com essa configuração temos dois objetos existentes na memória, cada um acessível através de sua variável específica. As duas áreas de memória são absolutamente distintas e, desde que se utilize a referência correta, alterações em um deles não refletirão em alterações no outro. A Figura 27 ilustra de forma simplificada de que forma a memória da JVM registrará estes objetos.

Além do estado, representado pelos valores de seus atributos, cada objeto possui também um comportamento, representado por seus métodos/operações. É através destes métodos que os objetos interagem entre si, procedimento também comumente conhecido como troca de mensagens. Para exemplificarmos a invocação de métodos, consideraremos somente o objeto representado por p. Pela definição da classe Pessoa, podemos visualizar que existem dois métodos, um para verificar a validade de um CPF e outro para imprimir o nome da pessoa.

FIGURA 27 - OBJETOS P E P1 NA JVM



FONTE: O autor

No Quadro 13 está demonstrada a invocação destes métodos. Na linha 13 fazemos uma validação do CPF através do método validarCPF. Basicamente perguntamos se o retorno da chamada do método é igual a *true*, indicando que o CPF é válido, para somente então atribuí-lo ao objeto. É importante ter em mente que podemos colocar qualquer valor de retorno para métodos e que este valor deverá ser resultado da lógica aplicada dentro do mesmo. Em nosso caso, a lógica aplicada no método é simples e serve apenas para demonstração. Caso esteja com dúvidas, basta olhar o código-fonte do método dentro da classe Pessoa.

QUADRO 13 - INVOCAÇÃO DE MÉTODOS

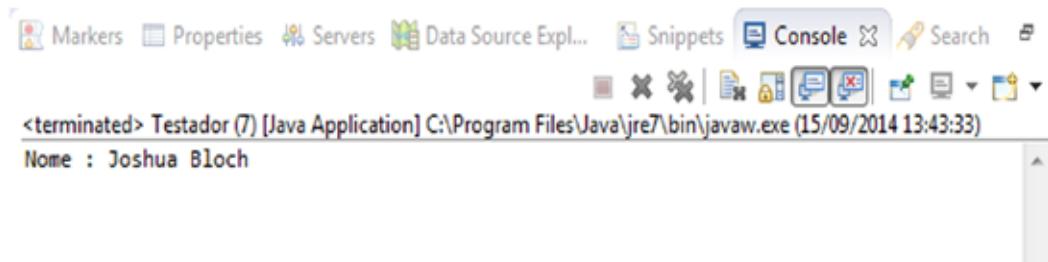
```

1 package primeiro;
2
3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         Pessoa p;
10        int cpf = 12345323;
11        p = new Pessoa();
12
13        if (p.validarCPF(cpf) == true)
14            p.CPF = cpf;
15
16        p.nome = "Joshua Bloch";
17        p.sexo = "Masculino";
18
19        p.imprimirNome();
20    }
21 }
```

FONTE: O autor

Na linha 19, chamamos o método `imprimirNome()` e o resultado deverá ser a impressão do valor contido no atributo `nome` do objeto na saída padrão, conforme a Figura 28. Lembramos que, antes de colocar valores para os atributos ou mesmo invocar métodos de uma classe, é essencial que exista um objeto instanciado através do `new`. Tentar atribuir valores para `p.nome` ou mesmo invocar `p.imprimirNome()` antes de instanciá-lo resultará em um erro conhecido como `NullPointerException`.

FIGURA 28 - RESULTADO NO CONSOLE



FONTE: O autor

LEITURA COMPLEMENTAR

Guia de Referência Rápida da Linguagem de Programação Java – Disponível originalmente em: <http://www.tutorialspoint.com/java/java_quick_guide.htm>. Traduzido e adaptado livremente pelo autor.

1) Identificadores

Todos os componentes Java requerem nomes. Os nomes usados para as classes, variáveis e métodos são chamados de identificadores. As regras para criação de identificadores são as seguintes:

- Todos os identificadores devem começar com uma letra (A a Z ou A a Z), de caráter monetário (\$) ou um sublinhado (_).
- Após o primeiro caractere, identificadores podem ter qualquer combinação de caracteres.
- Uma palavra-chave não pode ser utilizada como um identificador.
- Mais importante – identificadores são *case sensitive*.
- Exemplos de identificadores válidos: `idade`, `salário $`, `_value`, `__1_value`.
- Exemplos de identificadores ilegais: `123abc`, `-salario`.

2) Palavras reservadas

O Quadro 14 mostra as palavras reservadas em Java. Estas palavras reservadas não podem ser usadas como constante ou variável ou quaisquer outros nomes de identificadores.

QUADRO 14 – PALAVRAS RESERVADAS DO JAVA

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

FONTE: Disponível em: <http://www.tutorialspoint.com/java/java_quick_guide.htm>.

3) Comentários

Java suporta comentários de uma única linha e comentários de várias linhas, muito semelhantes ao C e C++. Todos os comandos disponíveis dentro de qualquer comentário são ignorados pelo compilador Java. O Quadro 15 ilustra os dois tipos de comentários do Java.

QUADRO 15 - EXEMPLOS DE COMENTÁRIOS DO JAVA

```

1 package livro;
2
3 public class MyFirstJavaClass {
4
5     /*
6      * Este é o meu primeiro programa java. Isto irá imprimir "Olá mundo", como
7      * a saída Este é um exemplo de comentários multi-linha.
8      */
9
10    public static void main(String[] args) {
11        // Este é um exemplo de uma única linha de comentário
12        /* Este é também um exemplo de linha simples comentário. */
13        System.out.println("Olá Mundo");
14    }
15 }
16

```

FONTE: Adaptado de: <http://www.tutorialspoint.com/java/java_quick_guide.htm>.

4) Tipos primitivos

Há oito tipos de dados primitivos suportados pelo Java. Tipos de dados primitivos são predefinidos pela linguagem e nomeado por uma palavra-chave. Vamos agora olhar em detalhes os oito tipos de dados primitivos.

- *byte*
- *short*
- *int*
- *long*
- *float*
- *double*
- *boolean*
- *Char*

5) Tipos Referência

- Variáveis de referência são criados usando construtores definidos para as classes. Essas variáveis são declaradas usando um tipo específico que não pode ser mudado. Por exemplo, Livro, Pessoa etc.
- O valor padrão de qualquer variável de referência é nula.
- A variável de referência pode ser usada para se referir a qualquer objeto do tipo declarado ou qualquer tipo compatível.
- Por exemplo: Animal = *new Animal ("girafa")*.

6) Operadores Aritméticos

Os operadores aritméticos são mostrados no Quadro 16. Para os exemplos, considere A = 10 e B = 20.

QUADRO 16 - OPERADORES ARITMÉTICOS DO JAVA

Operador	Descrição	Exemplo
+	Adição - Adiciona valores de cada lado do operador	A + B = 30
-	Subtração - Subtrai operando à direita do operando à esquerda	A - B = -10
*	Multiplicação - Multiplica valores de cada lado do operador	A * B = 200
/	Divisão - Divide operando esquerdo pelo operando direito	B / A = 2
%	Módulo - Divide operando esquerdo pelo operando direito e retorna restante	B% A = 0
++	Incremento - Aumentar o valor do operando por um	B ++ = 21
--	Decremento - Diminuir o valor do operando por um	B-- = 19

FONTE: Adaptado de: <http://www.tutorialspoint.com/java/java_quick_guide.htm>.

7) Operadores Lógicos

Os operadores lógicos são mostrados no Quadro 17. Para os exemplos, considere A como uma expressão booleana de valor Verdadeiro e B como uma expressão booleana de valor Falso.

QUADRO 17 - OPERADORES LÓGICOS DO JAVA

Operador	Descrição	Exemplo
<code>&&</code>	Chamado operador lógico <i>AND</i> . Se ambos os operandos são diferentes de zero, em seguida, a condição torna-se verdade.	(A <code>&&</code> B) é falsa.
<code> </code>	Chamado Lógico ou operador. Se qualquer um dos dois operandos são diferentes de zero, em seguida, a condição torna-se verdade.	(A <code> </code> B) é verdadeiro.
<code>!</code>	Chamado operador não lógico. Use a inverte o estado lógico do seu operando. Se a condição for verdadeira, então não lógico operador fará falsa.	<code>!</code> (A <code>&&</code> B) é verdadeira.

FONTE: Adaptado de: <http://www.tutorialspoint.com/java/java_quick_guide.htm>.

8) Operadores Relacionais

Os operadores aritméticos são mostrados no Quadro 18. Para os exemplos, considere A = 10 e B =20.

QUADRO 18 - OPERADORES RELACIONAIS DO JAVA

Operador	Descrição	Exemplo
<code>==</code>	Verifica se que o valor de dois operandos é igual ou não, se sim, então a condição se torna verdade.	(A <code>==</code> B) não é verdade.
<code>!=</code>	Verifica se o valor de dois operadores é igual ou não, se os valores não são iguais então a condição torna-se verdade.	(A <code>!=</code> B) é verdadeiro.
<code>></code>	Verifica se o valor do operando esquerdo for maior que o valor do operando direito, se sim, então a condição torna-se verdade.	(A <code>></code> B) não é verdade.
<code><</code>	Verifica se o valor do operando esquerdo for menor que o valor do operando direito, se sim, então condição torna-se verdade.	(A <code><</code> B) é verdadeiro.
<code>>=</code>	Verifica se o valor do operando esquerdo for maior ou igual ao valor do operando direito, se sim, então condição torna-se verdade.	(A <code>>=</code> B) não é verdade.
<code><=</code>	Verifica se o valor do operando esquerdo for menor ou igual ao valor do operando direito, se sim, então condição torna-se verdade.	(A <code><=</code> B) é verdadeiro.

FONTE: Adaptado de: <http://www.tutorialspoint.com/java/java_quick_guide.htm>.

9) Laços de Repetição

A linguagem Java possui três laços simples de repetição, conforme ilustrado no Quadro 19. O primeiro e o segundo são adequados para situações onde não se sabe o número de repetições, sendo que o segundo garante a execução do bloco de comandos pelo menos uma vez. O terceiro laço de repetição deve ser utilizado quando se sabe exatamente o número de repetições necessárias. O comando *break*, demonstrado no terceiro laço, serve para interromper de forma brusca a repetição em qualquer um dos três tipos de laço.

QUADRO 19 - TIPOS DE REPETIÇÃO

```

while(condicao==true){
    /* repete o que estiver dentro do bloco
       enquanto condicao tiver valor verdadeiro */
}

do{
    /* semelhante ao bloco anterior, com a ressalva
       de executar o comando pelo menos uma vez */
} while(condicao==true);

for (int i = 0; i < 10; i++) {
    /* executa um bloco de comandos
       até que o i receba o valor 10 */
    break;
}

```

FONTE: O autor

10) Desvios Condicionais

Um desvio condicional consiste de uma instrução IF que permite mudar o fluxo de execução do programa de acordo com o valor de uma expressão *booleana*. O Quadro 20 ilustra a utilização do IF em duas situações distintas. Na primeira delas, caso a expressão *booleana* seja verdadeira, procede-se com a execução dos comandos. Já na segunda, caso a expressão *booleana* seja verdadeira, procede-se com a execução dos comandos e caso seja falsa, existe um bloco específico de instruções para ser executado.

QUADRO 20 - DESVIO CONDICIONAL

```
if(condicao==true){  
    // executa bloco de comandos  
}  
  
if (condicao==true){  
    // executa bloco de comandos  
} else {  
    /* executa bloco específico para  
       condicao == false */  
}
```

FONTE: O autor

Por aqui encerramos a leitura complementar desta unidade. Lembramos você de que o objetivo desta leitura é fornecer subsídios básicos para que possa escrever seus programas em Java e de forma alguma temos a pretensão de esgotar o tema. Para mais informações e detalhes sobre a sintaxe da linguagem Java, sugerimos uma visita ao site da *Oracle* <<https://www.oracle.com/java/index.html>>, empresa que atualmente é proprietária da tecnologia.

Bons estudos!

RESUMO DO TÓPICO 3

Neste tópico vimos:

- As classes são compostas de um nome, um conjunto de atributos e um conjunto de métodos.
- Os atributos representam as informações que a classe contém dentro dela. Por exemplo, uma classe chamada Pessoa poderia possuir um atributo nome, outro CPF e outro sexo.
- Na linguagem de programação Java, cada atributo deve necessariamente ser tipado, pois a linguagem utiliza o conceito conhecido como tipagem forte.
- No exemplo da classe Pessoa, os atributos nome e sexo poderiam ser do tipo String e o atributo CPF poderia ser do tipo int.
- Na linguagem de programação Java, nomes de atributos e de variáveis locais começam geralmente com letra minúscula, por exemplo: nome, sexo.
- Caso um atributo ou variável tenha dois ou mais nomes, o primeiro nome começa com letra minúscula e os demais com letra maiúscula. Por exemplo: nome do dependente, contador de pontos, porcentagem aumento.
- Os métodos, também conhecidos como operações, representam funções que podem ser invocadas nos objetos.
- Na linguagem de programação Java, os métodos podem possuir 0 ou 1 retorno, sendo que este retorno pode ser de qualquer tipo, inclusive classes que você mesmo criou.
- Caso um método não retorne nada, a palavra void deve ser colocada.
- Por padrão, nomes de métodos começam com um verbo no infinitivo mais um substantivo. Por exemplo: calcularSalario(), validarCPF(), desenharFigura() são bons nomes para métodos.
- A nomenclatura dos métodos obedece ao mesmo padrão dos atributos, sendo que a primeira palavra começa com letra minúscula e as demais com letra maiúscula, vide exemplos do item 10.
- Os métodos podem receber parâmetros para executarem operações e validações. Estes parâmetros podem ser de qualquer tipo de dados já existente na linguagem ou mesmo uma classe que você tenha criado.

- As boas práticas de programação pregam que um método deve ter no máximo cinco parâmetros, caso contrário, seria melhor dividi-lo em métodos menores com menos parâmetros.
- Em situações específicas a regra definida no item 13 pode ser quebrada, tanto que isso ocorre diversas vezes nas próprias bibliotecas da linguagem de programação Java.
- A instanciação de uma classe é o momento em que ela se torna um objeto em memória e pode ser utilizada no programa.
- Somente a partir da instanciação é que valores podem ser alocados para os atributos e os métodos podem ser invocados, caso contrário, o compilador disparará uma exceção.
- Os valores dos atributos de um objeto são conhecidos como o estado deste objeto, enquanto os métodos são conhecidos como o comportamento deste objeto.

AUTOATIVIDADE



1 Avalie as afirmações a seguir, colocando V para as afirmações Verdadeiras e F para as afirmações Falsas:

- () Os atributos de uma classe definem o comportamento dela.
- () Todos os métodos devem retornar um tipo de dado após terminarem sua execução.
- () Os símbolos { e } representam o início e o fim de um bloco de código fonte, que pode ser uma classe, um método etc.
- () A declaração de uma variável ou um atributo utilizando como tipo uma classe recém-criada somente pode acontecer após a instanciação.
- () Gerar Relatório é um nome adequado para uma classe, uma vez que é significativo em relação ao significado.

Agora assinale a alternativa que contém a sequência CORRETA:

- a) F, F, V, F, F.
- b) F, V, V, F, F.
- c) V, F, V, F, F.
- d) F, F, V, F, V.

2 Utilizando os procedimentos mostrados neste tópico, crie no Eclipse um projeto com o nome Veterinário que contenha um pacote chamado canil e uma classe chamada Cachorro, contendo as seguintes informações:

- Atributos: nome, raça e peso.
- Métodos para coçar (sem implementação) e para latir, imprimindo na saída padrão a mensagem (“au...au...au”).

3 Utilizando o projeto do exercício anterior, crie dentro do pacote canil uma classe chamada de Testadora, através da qual você deve:

- Instanciar um objeto do tipo Cachorro.
- Atribuir valores para os atributos do Objeto.
- Invocar o método latir() do objeto instanciado e verificar o resultado na saída padrão.

4 Utilizando o projeto do exercício anterior, crie um novo pacote chamado de curral e dentro deste pacote, crie uma classe que represente um tipo de animal. Esta classe deve possuir atributos e métodos condizentes com o animal que ela representa.

ENCAPSULAMENTO, HERANÇA E POLIMORFISMO

OBJETIVOS DE APRENDIZAGEM

Ao final desta unidade, você será capaz de:

- utilizar a linguagem de programação Java para implementar os princípios fundamentais da Programação Orientada a Objetos;
- entender o conceito e a aplicabilidade do encapsulamento de suas classes;
- reutilizar código já escrito e testado através da herança;
- diferenciar situações onde a herança é ou não aplicável;
- tornar seu código-fonte mais flexível em tempo de execução através do polimorfismo.

PLANO DE ESTUDOS

Esta unidade de ensino está dividida em três tópicos e no final de cada um deles, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – ENCAPSULAMENTO

TÓPICO 2 – HERANÇA

TÓPICO 3 – POLIMORFISMO



ENCAPSULAMENTO

1 INTRODUÇÃO

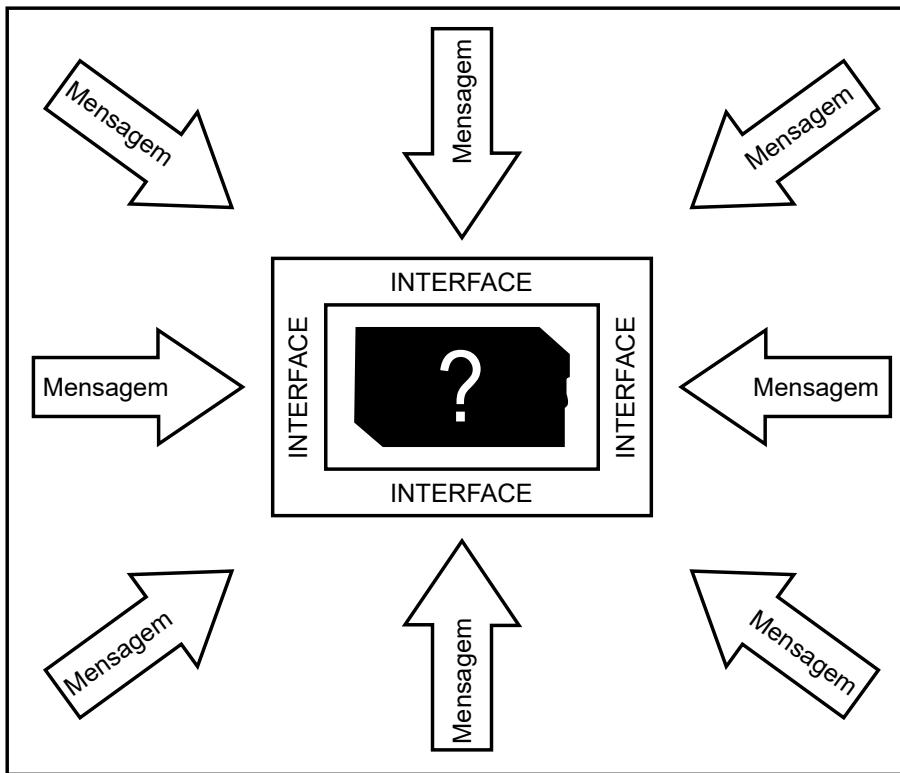
Adotar uma estratégia de programação orientada a objetos para desenvolvimento de *software* significa não sair diretamente programando, mas, sim, planejar de forma cuidadosa respeitando as teorias que embasam esse paradigma.

Sintes (2002) coloca que a teoria da programação orientada a objetos é fundamentada por muitos princípios e práticas, mas entre estes podemos destacar o encapsulamento, a herança e o polimorfismo.

Segundo o próprio Sintes (2002), encapsulamento é a característica da programação orientada a objetos que permite ocultar partes de seu código. Ao invés de ver um programa como uma única entidade grande e monolítica, o encapsulamento permite que você o divida em partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes, ocultando os detalhes internos de implementação.

Já Bates e Sierra (2010) reforçam a ideia de que o encapsulamento auxilia a proteger os dados internos de um objeto de valores inválidos, impedindo que alguma entidade externa os accesse diretamente. Qualquer acesso aos atributos que definem o estado de um objeto deve ser feito através de métodos. Além de proteger suas variáveis de valores indesejáveis, a utilização de métodos permite que internamente sejam feitas modificações em suas classes sem alterar quem interage com elas. Esses métodos que permitem que se interaja com o objeto são conhecidos como sua interface externa. Essa interface externa permite que se defina quais partes de seu objeto você deseja que fiquem visíveis ao mundo exterior e quais fiquem ocultas. Em suma, tudo que não fizer parte da interface externa do objeto fica oculto do mundo exterior, conforme ilustrado na Figura 29.

FIGURA 29 – INTERFACE EXTERNA DE UM OBJETO



FONTE: SINTES (2002)

O encapsulamento garante que você exponha somente o que é necessário para o mundo exterior e, consequentemente, oculte o desnecessário. Imaginemos uma classe Pessoa, com os atributos nome e CPF. Para garantir que os objetos instanciados a partir desta classe sejam “Bons cidadãos”, implementaremos um método que permita a validação de um número de CPF. Todos os números de CPF seguem um algoritmo para geração, o que permite que façamos uma validação dos mesmos. Agora uma pergunta: de que forma o CPF será validado é um conhecimento necessário para os demais objetos que utilizarão o objeto Pessoa? NÃO! Os demais objetos somente precisam saber se o CPF fornecido é válido ou não válido, para procederem com a correção, caso esta seja necessária. Este é um exemplo típico de encapsulamento, onde o método que valida o CPF ficaria oculto da interface externa do objeto e, portanto, inacessível às demais classes.



O termo “Bom cidadão” significa que precisamos manter o seu estado válido.

Um objeto com estado válido é aquele que possui o seu mínimo de atributos necessários com valores válidos.

Para fortalecer o seu entendimento, vamos exemplificar o encapsulamento em um exemplo que não está relacionado a desenvolvimento de *software*. Imagine que você está em frente à TV que mencionamos na última unidade, segurando o controle remoto e tentando mudar de canal. Quando você pressiona o botão para mudar de canal no controle remoto, um pulso elétrico é gerado na placa de circuito impresso no local da função específica relacionada à mudança de canal. O circuito impresso então aciona o *led* infravermelho que envia um sinal e alcança o sensor na TV. A TV por sua vez troca de canal e você pode ficar sentado tranquilamente no sofá. Apesar de simplificada, a explicação acima detalha a operação e a interação entre o controle remoto e a TV. Agora a pergunta: Você precisa saber desses detalhes para trocar de canal na TV? Novamente a resposta é NÃO! Bastaria apertar o botão, deixar os “objetos” trocarem mensagens entre si e realizar o trabalho. Neste caso, os botões do controle remoto representariam a interface externa do objeto, onde somente o que nos interessa é exposto, conforme Figura 30.

FIGURA 30 - PESSOA, TV E CONTROLE REMOTO



FONTE: Microsoft Cliparts, 2014

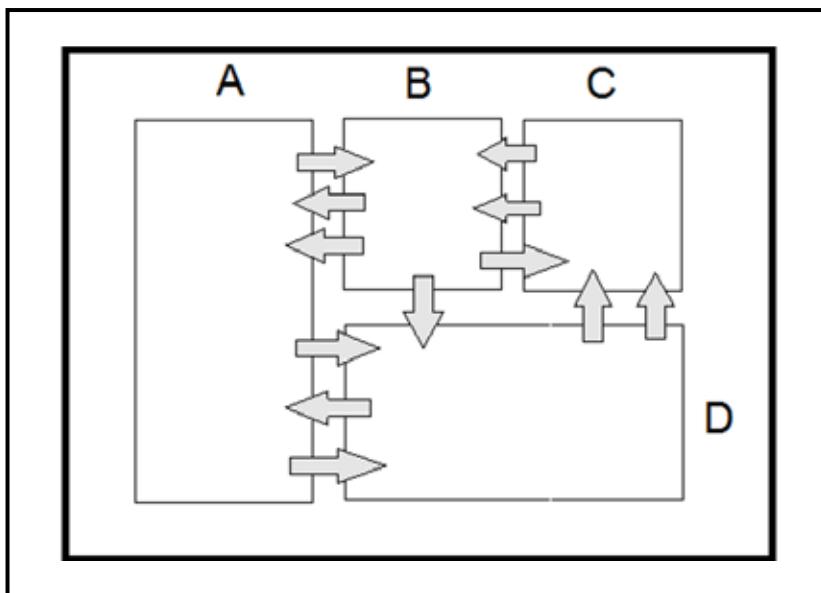
Na programação orientada a objetos devemos pensar exatamente da mesma forma, expondo somente o que é absolutamente necessário aos usuários de nossa classe. Lembrando que em geral, os usuários de nossa classe serão outros programadores, que precisarão das funcionalidades disponibilizadas. Ao ocultar detalhes desnecessários você reduz o acoplamento, e consequentemente a manutenção. Lembre-se de que qualquer interface exposta será eventualmente utilizada por outros programadores, o que impedirá você de ocultá-la depois, com o risco de quebrar funcionalidades que já estejam rodando.

2 ACOPLAGEMTO E COESÃO

Entender os conceitos de acoplamento e coesão é extremamente importante para entender corretamente os objetivos do encapsulamento de forma a aplicá-los de maneira correta.

Em termos de programação, acoplamento significa relacionamento, ou seja, quanto mais forte o acoplamento entre duas classes, maior o grau de relacionamento entre elas. À primeira vista isso parece algo positivo, entretanto, um grau maior de relacionamento implica um grau maior de conhecimento dos detalhes internos de implementação das classes com as quais nos relacionamos. Já vimos que quanto menos detalhes de implementação soubermos das demais classes do sistema, melhor, pois o que nos interessa é usar as funcionalidades. Classes fortemente acopladas frequentemente implicam manutenção conjunta, ou seja, caso uma classe A esteja fortemente acoplada com outra classe B, a probabilidade de que uma manutenção em A obrigue uma manutenção em B (e vice-versa) aumenta consideravelmente.

FIGURA 31 - ACOPLAGEMTO FORTE

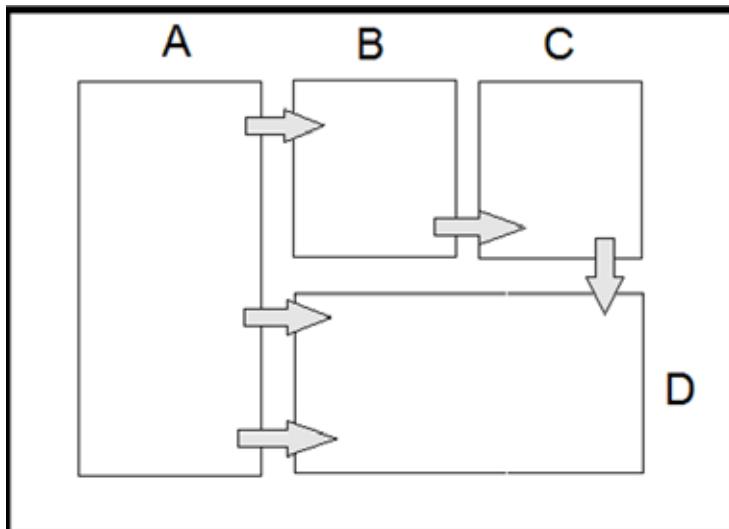


FONTE: O autor

Imagine um conjunto de classes representado pela Figura 31, onde A, B, C e D são as classes que interagem entre si para fornecer determinada funcionalidade. Os pontos ligados por setas são os locais onde as classes se relacionam entre si. Quanto maior o número de setas ligando duas classes, maior o grau de acoplamento. Esse acoplamento representa, em termos práticos, métodos ou atributos da interface pública destas classes. Quando ocorrer manutenção em qualquer lugar de A, B, C ou D que estiver conectado a outra classe através de uma seta, necessariamente haverá manutenção nos dois lados da conexão. Por este simples fato dizemos que o encapsulamento, quando feito corretamente, auxilia a manter o fraco acoplamento entre as classes. Logicamente, existem situações onde não é possível fugir do acoplamento forte, mas saber que estas situações devem ser evitadas ao máximo auxilia a projetar classes com maior qualidade.

Em contrapartida, a Figura 32 ilustra uma situação onde o mesmo conjunto de classes apresenta acoplamento fraco. Podemos perceber que o número de relacionamentos entre as classes é bem menor, o que por si só, diminui o número de locais que precisariam de manutenção em ambas as terminações da seta. É importante salientar que ALGUM tipo de acoplamento deve existir entre as classes, caso contrário, estas não conseguiram enviar mensagens entre si. O segredo é avaliar com cuidado se existe a necessidade de expor a funcionalidade ou não.

FIGURA 32 - ACOPLAMENTO FRACO



FONTE: O autor

Vamos exemplificar novamente o encapsulamento através de um exemplo feito em código-fonte? No Quadro 21 apresentamos uma classe que só tem uma funcionalidade: receber um valor inteiro e retornar o fatorial deste valor.

QUADRO 21 - CLASSE QUE FAZ CÁLCULO DO FATORIAL

```

1 package base;
2
3 public class Fatorial {
4
5     public int calcularFatorial(int valor){
6         if(valor==0)
7             return 1;
8         return valor*calcularFatorial(valor-1);
9     }
10
11 }
```

FONTE: O autor

Esta classe faz uso da recursividade para calcular o fatorial de um número, o que a torna pouco legível e razoavelmente complexa. Fazemos inclusive a validação do valor 0, que é matematicamente diferente de todos os demais fatoriais, resultando sem cálculo algum no valor 1. Entretanto, isso não faz diferença para quem a utiliza, visto que basta instanciar o objeto, chamar o método enviando o valor a ser calculado como parâmetro e aguardar o retorno. A implementação propriamente dita estará encapsulada e, caso algum dia venha a sofrer manutenção, não fará diferença, pois você interagirá somente com a chamada do método. Desta forma, o encapsulamento acaba também por auxiliar na reutilização de código, pois caso você ou outro programador de sua equipe no futuro precisem de um cálculo de fatorial, bastará adicionar a classe Fatorial ao projeto.

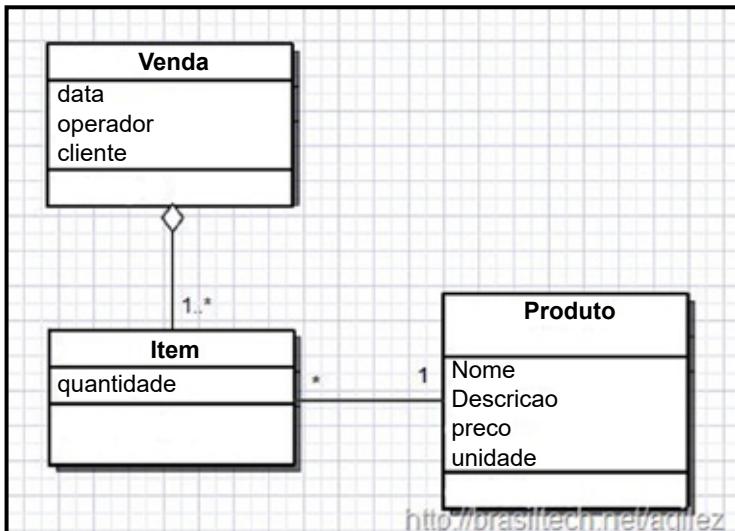


A técnica de recursividade, utilizada no exemplo do Quadro 1, consiste em criar uma lógica onde seja possível chamar uma função dentro dela mesma. Existem alguns problemas computacionais que somente podem ser resolvidos com a utilização da técnica. Para maiores detalhes sobre de que forma a recursividade funciona, sugerimos o site <<http://www.linhadecodigo.com.br/artigo/3316/recursividade-em-java.aspx>>.

Já a coesão é definida por Miller (2014) como a medida de proximidade no relacionamento entre todas as responsabilidades, os dados e os métodos de uma classe, ou seja, uma classe coesa é uma classe que tem uma função bem definida no sistema. Não é tão simples avaliar se uma classe é ou não coesa, mas um teste de coesão é examinar uma classe e decidir se todo o seu conteúdo está relacionado e é descrito pelo nome da classe. Por exemplo, digamos que você tenha as classes EstacaoDeTrem, Trem, Maquinista e HorarioDosTrens em um sistema e existam funções envolvendo horários nas classes EstacaoDeTrem e Trem. Você já criou a classe HorarioDosTrens para lidar com essas questões de horário, então para manter a coesão e garantir que cada classe tenha somente uma responsabilidade no sistema, estas funções deveriam ser movidas para lá. Logicamente, classes com nomes mais genéricos acabam por ter mais de uma responsabilidade, mas este tipo de situação deve ser evitado sempre que possível.

Mais um exemplo de coesão pode ser visto na Figura 33. Neste exemplo temos três classes relacionadas entre si através de um diagrama de classes da UML. A classe Venda tem um relacionamento do tipo agregação com a classe Item e a classe Item tem um relacionamento de associação com a classe Produto. A pergunta é a seguinte: se quiséssemos saber o total da venda, de que forma colocaríamos os métodos nas classes, respeitando os princípios da alta coesão e baixo acoplamento?

FIGURA 33 - DIAGRAMA DE CLASSES



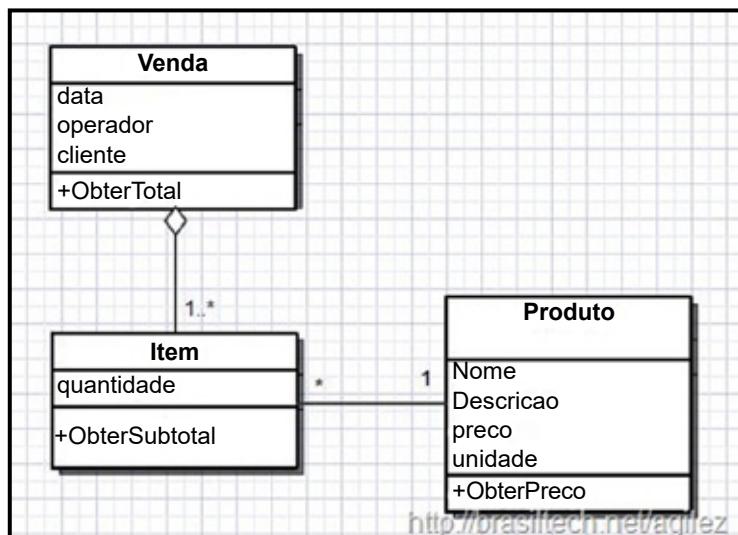
FONTE: CUNHA, (2009)



A UML (Unified Modeling Language) é uma linguagem para modelagem de sistemas orientadas a objeto utilizada em todo o mundo. Você a conhecerá em detalhes em outras disciplinas, mas caso queira se aprofundar imediatamente, sugerimos o site: <<http://www.uml.org>>.

Para calcular o total da venda precisamos das informações do preço e da quantidade de cada item comprado. O primeiro passo seria criar uma interface na classe Produto que permitisse o acesso externo ao preço de cada objeto deste tipo. O segundo passo seria criar uma interface externa na classe Item que calculasse o subtotal, com base no preço fornecido pela classe Produto e na quantidade do próprio Item. Finalmente, o método que faz o cálculo final seria colocado na classe Venda, visto que o total deve ser conhecido somente por ela. Neste método, a classe faria uma iteração nos itens, solicitando o subtotal de cada um e acumulando o total, conforme Figura 34.

FIGURA 34 - DIAGRAMA DE CLASSES COM MÉTODOS



FONTE: CUNHA, (2009)

3 IMPLEMENTAÇÃO EM JAVA

Uma vez que abordamos a parte teórica, vamos demonstrar agora em um exemplo prático de implementação na linguagem de programação Java.

Conforme colocamos anteriormente, encapsular basicamente significa ocultar. No caso específico do Java, marcamos com modificadores de visibilidade os atributos, métodos ou classes que desejamos encapsular. São quatro os modificadores de visibilidade da linguagem de programação Java:

- 1) ***private***: fornece acesso somente dentro da classe onde estiver declarado, no caso de atributos ou métodos. Classes privadas somente fazem sentido se forem internas a outra classe;
- 2) ***public***: fornece acesso dentro da aplicação onde estiver declarado, no caso de classes, atributos ou métodos. Quando marcamos uma classe como *public*, queremos dizer que esta será visível a todas as demais classes da aplicação, independentemente do pacote onde estiver. Logicamente, existe a necessidade da referência da classe que se deseja utilizar, caso esta esteja em outro pacote. Veremos de que forma essa referência é feita no exemplo de implementação;
- 3) ***default***: o modificador *default* é implementado simplesmente não colocando nenhum modificador na frente da classe, atributo ou método. Ele indica que existe visibilidade dentro do pacote onde você estiver. Comumente esse modificador é conhecido como *private* de pacote;
- 4) ***protected***: este modificador indica que o atributo ou método será visível somente na subclasse de um relacionamento de herança. Abordaremos herança em maiores detalhes no Tópico 2 desta unidade.



Classes internas são classes cuja definição ocorre dentro de outra classe já existente. Em geral sua utilização não é uma boa prática.

Para saber mais acesse: <<http://blog.caelum.com.br/classes-aninhadas-o-que-sao-e-quando-usar/>>.

Trazemos um exemplo utilizando uma classe chamada de ContaCorrente, que possui os atributos correntista, número e saldo. Inicialmente esta classe não respeitará os conceitos de coesão e acoplamento, mas faremos modificações de forma a torná-la mais aderente a tais conceitos. O Quadro 22 traz o código-fonte da primeira versão da classe. A primeira novidade está entre as linhas 9 e 12, onde é declarado um método construtor. O objetivo do método construtor é construir o objeto já inicializando os atributos com valores vindos na instanciação. Perceba que neste caso, recebemos um valor inteiro, que depois é atribuído para a variável numero e uma String, que depois é atribuída para correntista. O método construtor pode também ser utilizado para garantir que um objeto terá pelo menos alguns valores mínimos quando for criado. No nosso caso, essa configuração de construtor não permitirá que se crie um objeto ContaCorrente sem um número e um correntista, o que convenhamos, faz todo o sentido. Uma classe pode ter quantos construtores quisermos, desde que a ordem ou o tipo dos parâmetros seja diferente. Para criar um método construtor basta colocar o modificador public (por enquanto veremos somente construtores públicos), nenhum tipo de retorno e o nome da própria classe.

QUADRO 22 - CLASSE CONTACORRENTE

```

2
3 public class ContaCorrente {
4
5     String correntista;
6     double saldo;
7     int numero;
8
9     public ContaCorrente(int nro, String corr){
10         numero = nro;
11         correntista = corr;
12     }
13
14 }
```

FONTE: O autor

No Quadro 23 podemos observar a instanciação de um objeto ContaCorrente e em sequência a modificação dos valores de seus atributos. Perceba que na instanciação simulamos uma entrada de dados do usuário, já os colocando de forma literal no construtor. A partir deste momento, o método construtor da classe ContaCorrente é chamado e executa os comandos que estiverem nele. Neste caso, recebe os dois parâmetros e coloca seus valores nos atributos correspondentes. Lembre-se de que o construtor é um método como qualquer outro e que podemos inclusive chamar outros métodos dentro de um construtor.

Observe atentamente e tente dizer o que não está coerente no exemplo do Quadro 3. Se formos considerar os conceitos de encapsulamento e coesão, além do mecanismo de funcionamento de uma Conta Corrente, existem várias incoerências, que destacaremos a seguir:

Criamos o método construtor justamente para garantir que toda Conta Corrente tivesse no mínimo um numero e um correntista, entretanto, nas linhas 10 e 11 alteramos estes valores. Isto é possível em uma Conta Corrente real? A resposta é não... Tente chegar a seu banco e dizer que não quer mais o número que você usa para sua conta e que quer passar a titularidade desta conta para outra pessoa. Estas são operações que não podem ser realizadas, ou seja, o número de sua conta e o titular são imutáveis. Caso você deseja outra conta corrente, deveria encerrar a que está usando atualmente e somente então criar outra.

Na linha 9 fazemos uma alteração direta ao saldo de sua conta, simplesmente atribuindo o valor 500. Novamente pergunto a você, isso é possível em uma Conta Corrente real? A resposta é Não, para alterar o saldo, você deve realizar operações de saque e depósito/transferência.

Não queremos que o correntista seja modificado depois que o objeto foi criado, entretanto, é possível que este correntista mude de nome? Sim, essa é uma situação bastante comum, como por exemplo, caso uma mulher adquira matrimônio e adote o sobrenome do marido.

QUADRO 23 - TESTADOR DE CONTACORRENTE

```

2
3 public class Testador {
4
5     public static void main(String[] args) {
6
7         ContaCorrente cc = new ContaCorrente(12333, "Bill Joy");
8
9         cc.saldo = 500;
10        cc.numero = 3432;
11        cc.correntista = "Outro correntista";
12
13    }
14
15 }
```

FONTE: O autor

Para corrigirmos todos estes problemas, iniciaremos encapsulando o acesso ao número da conta, ao saldo e ao correntista, marcando-os com o modificador private, conforme Quadro 24. Ao realizarmos essa modificação, automaticamente as linhas 10 e 11 da classe Testador param de funcionar, pois impedimos o acesso a estes atributos em qualquer outra classe fora de ContaCorrente.

QUADRO 24 - MODIFICADOR PRIVATE

```

3 public class ContaCorrente {
4
5     private String correntista;
6     private double saldo;
7     private int numero;
8
9     public ContaCorrente(int nro, String corr){
10         numero = nro;
11         correntista = corr;
12     }
13
14 }
```

FONTE: O autor

Nosso encapsulamento não pode simplesmente ocultar os atributos, mas sim fornecer acesso a eles de forma adequada. O número da conta corrente é absolutamente imutável, então forneceremos acesso a ele através de um método Getter (Quadro 25). O objetivo de um método Getter é fornecer acesso somente de leitura a um atributo que esteja privado na Classe, exatamente como no caso da Conta Corrente. O padrão para a criação de um método Getter pode ser visto na linha 14, onde colocamos o modificador public, o tipo do retorno do método, que deve ser exatamente o mesmo do atributo que se quer fornecer acesso e finalmente o nome do método, que na linguagem Java é a palavra *get* concatenada com o nome do atributo. Em nosso caso, a assinatura do método fica *public int getNumero()*.



Existem exceções à regra, mas neste início de aprendizado, você pode marcar TODOS os atributos de suas classes com o modificador private e fornecer métodos de acesso aos mesmos, quando necessário.

No caso do saldo, a questão é um pouco mais complicada. Sem dúvida, desejamos ter acesso ao saldo da conta corrente, o que nos leva a criar também um Getter para o Saldo, conforme Quadro 5. E se quisermos mudar o valor do saldo? Em nenhuma conta corrente real é possível alterar diretamente o valor do saldo. Para mantermos o encapsulamento e a coesão, criaremos dois métodos, um que permite sacar um valor da Conta Corrente e outro que permite depositar um valor, acrescendo o saldo. Ambos os métodos podem ser vistos no Quadro 6. O primeiro método recebe um parâmetro valor, referente ao valor que deve ser retirado do saldo. Perceba que fazemos uma validação, verificando se existe saldo suficiente antes de realizar o saque. Já o segundo método, simplesmente adiciona o valor recebido como parâmetro ao saldo do objeto. Nosso objetivo é deixar o código o mais coeso e fracamente acoplado possível. Perceba a verificação do saldo existente no método sacar. Você acredita que é responsabilidade do método sacar saber quanto existe de saldo? A resposta é outra vez NÃO. Encapsularemos esta funcionalidade criando um método chamado de existeSaldo() (Figura 26), que fará a verificação e retornará um *booleano*. Esta refatoração inclusive auxiliará as manutenções futuras, caso outro tipo de verificação seja adicionada. Um detalhe interessante sobre este método específico pode ser observado no modificador de visibilidade do mesmo. Adicionamos o modificador *private*, pois não é necessário que qualquer outra classe fora de conta corrente tenha acesso a essa informação. Caso qualquer outra classe deseje saber qual o saldo de conta corrente, ela deverá chamar o acessor respectivo, o método getSaldo().

QUADRO 25 – MÉTODOS GETTERS

Quadro	5.	Métodos	Getters
<pre> 3 public class ContaCorrente { 4 5 private String correntista; 6 private double saldo; 7 private int numero; 8 9 public ContaCorrente(int nro, String corr){ 10 numero = nro; 11 correntista = corr; 12 } 13 14 public double getSaldo(){ 15 return saldo; 16 } 17 18 public int getNumero(){ 19 return numero; 20 } 21 }</pre>	5.		

FONTE: O autor

QUADRO 26 – MÉTODOS QUE INTERAGEM COM O SALDO

```

36  public void depositar(double valor){
37      saldo = saldo + valor;
38  }
39
40  public void sacar(double valor){
41      if (existeSaldo(valor) == true)
42          saldo = saldo - valor;
43  }
44
45  private boolean existeSaldo(double valor){
46      if (valor <= saldo)
47          return true;
48      else
49          return false;
50  }

```

FONTE: O autor

Existe ainda uma incoerência em nossas classes relacionada ao correntista. Implementamos simplesmente uma *String* para armazenarmos o nome do proprietário da conta corrente. A primeira vista, essa implementação parece suficiente, mas e se quiséssemos armazenar também o telefone do correntista? Adicionaríamos o atributo telefone na classe conta corrente. Em uma futura manutenção surge também a necessidade de envirmos correspondência ao correntista e inserirmos o atributo endereço na classe conta corrente. Agora a pergunta é: Por que essa inserção de informações é incoerente com o conceito de coesão e acoplamento?

Se precisássemos de mais informações sobre o correntista e estas fossem inseridas dentro da classe ContaCorrente, estariam violando o conceito de coesão, pois a classe ContaCorrente sabe muitos detalhes sobre o correntista. Nesta situação, o ideal seria criarmos uma classe Correntista e a relacionarmos com a classe ContaCorrente através de um relacionamento de associação. O Quadro 27 ilustra a classe Correntista com os atributos nome e endereço.

QUADRO 27 - CLASSE CORRENTISTA

```

3 public class Correntista {
4
5     private String nome, endereco;
6
7     public Correntista (String corr, String end){
8         nome = corr;
9         endereco = end;
10    }
11
12    public String getCorrentista(){
13        return nome;
14    }
15
16    public void setEndereco(String end){
17        endereco = end;
18    }
19
20    public String getEndereco(){
21        return endereco;
22    }

```

FONTE: O autor

Perceba que criamos um construtor para a classe Correntista e consequentemente obrigamos qualquer instância a ter no mínimo nome e endereço. Criamos também *Getters* para o nome e o endereço e ainda um método que permite alterar o endereço de um correntista, afinal nada impede que este altere sua residência. A novidade aqui é a utilização de um método conhecido como *Setter*. Enquanto a função do *Getter* é fornecer acesso de leitura para atributos privados, o *Setter* recebe um parâmetro e o atribui para o que se pretende alterar. A configuração de um método *Setter* pode ser vista nas linhas 16 a 18. O método é público, retorna um *void* e é sempre nomeado com a palavra *set* concatenada com o nome do atributo. No nosso caso, o método chama-se *setEndereco*. O nome do parâmetro que o método receberá é indiferente, desde que o tipo seja o mesmo do atributo. Por hora manteremos a classe Correntista assim, simples, lembrando que o importante é que consigamos mudá-la sem interferir com ContaCorrente, mantendo assim o acoplamento fraco.

Para mantermos essa independência entre as classes, faz-se necessária uma última alteração na classe ContaCorrente. Em todos os lugares que fazíamos referência a uma *String* para o correntista, faremos agora referência a um objeto do tipo Correntista. Lembre-se de que, a partir do momento em que criamos uma classe, podemos interagir com um novo tipo de dados. O Quadro 28 mostra as alterações necessárias na classe ContaCorrente.

QUADRO 28 - CLASSE CONTACORRENTE ALTERADA

```

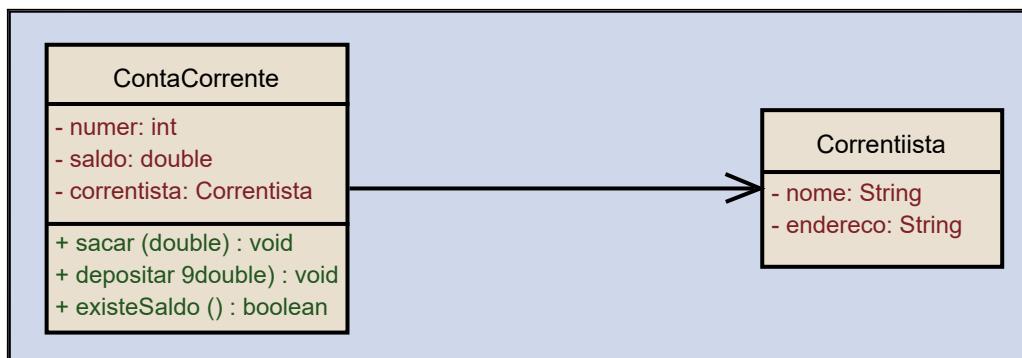
2
3 public class ContaCorrente {
4
5     private Correntista correntista;
6     private double saldo;
7     private int numero;
8
9     public ContaCorrente(int nro, Correntista corr){
10         numero = nro;
11         correntista = corr;
12     }
13
14     public Correntista getCorrentista(){
15         return correntista;
16     }
17

```

FONTE: O autor

Perceba que mantemos a obrigatoriedade da instanciação da conta corrente já com o correntista e a impossibilidade de alterá-lo, uma vez que não há o Setter para o correntista. Com essa alteração, qualquer manutenção feita na classe Correntista não refletirá na classe ContaCorrente, visto que esta já recebe um objeto pronto do tipo Correntista. Dizemos que a classe Correntista está associada à classe ContaCorrente (Figura 35). A seta sai da classe ContaCorrente e aponta na direção da classe Correntista, indicando a visibilidade e acessibilidade do relacionamento. Pela figura, deduzimos que a ContaCorrente tem acesso e pode ver Correntista, enquanto o contrário não é verdade.

FIGURA 35 - RELACIONAMENTO DE ASSOCIAÇÃO



FONTE: O autor

Mas como procedemos com a criação da ContaCorrente? A criação de uma ContaCorrente pressupõe a existência de um Correntista, conforme demonstrado nas linhas 13 e 14 do Quadro 29. Uma vez que dispomos de um objeto do tipo Correntista, procedemos com a criação da conta corrente. Na linha 16 instanciamos o objeto, referenciado pela variável cc, passando como parâmetro o número da conta e o correntista criado anteriormente. Na linha 17 fazemos a invocação do método depositar. Neste exemplo podemos perceber que cada classe é responsável por resolver parte do problema e que o relacionamento entre elas ocorre somente onde é indispensável que haja comunicação. Nem sempre é tão simples descobrir quais são as responsabilidades exatas das classes que compõem o seu sistema, entretanto, dadas as vantagens que a alta coesão e o baixo acoplamento apresentam quando houver (e com certeza haverá) manutenções, vale a pena pensar um pouco a respeito quando as estiver projetando.

QUADRO 29 - INSTANCIACÃO DA CONTACORRENTE

```

11*   public static void main(String[] args) {
12
13     Correntista c = new Correntista("Edsger Dijkstra",
14                               "Universidade Stanford");
15
16     ContaCorrente cc = new ContaCorrente(23331, c);
17     cc.depositar(450.90);
18
19 }
```

FONTE: O autor

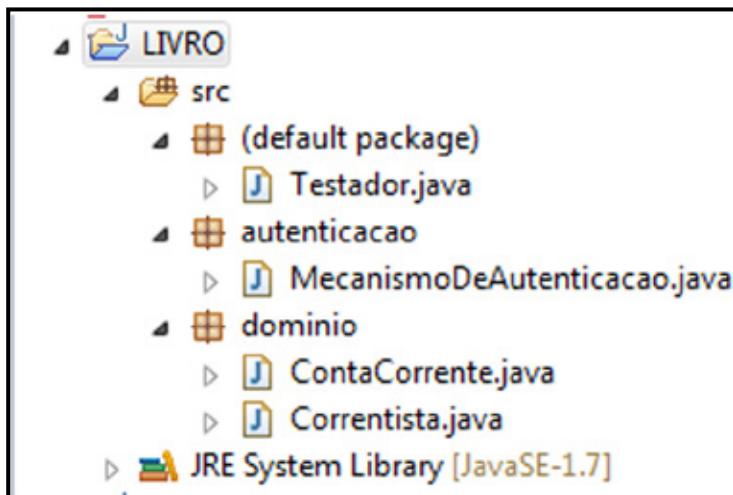
3.1 VISIBILIDADE ENTRE PACOTES

Até aqui aprendemos maneiras de utilizar os modificadores *public* e *private* para ocultar ou fornecer acesso a métodos ou atributos de nossas classes. Mas e quanto a esses modificadores aplicados a nossas classes?

O modificador *private* somente faz sentido em uma classe caso esta seja interna, mas como esta é uma situação relativamente rara, vamos nos concentrar na questão dos modificadores *public* e *default*.

Ao marcarmos uma classe com o modificador *public*, estamos dizendo que esta classe será visível em todo nosso projeto, entretanto, essa visibilidade não ocorre de forma transparente ao desenvolvedor. Vejamos o exemplo do quadro 10, onde três classes estão em pacotes diferentes. Podemos perceber que existem duas classes no pacote domínio, uma classe no pacote autenticação e uma classe no pacote default, que é o mesmo que dizer que ela não está em pacote nenhum. Vamos tentar utilizar o Testador para instanciar o mesmo objeto ContaCorrente dos exemplos anteriores, só que agora as classes que o Testador utilizará estão em pacotes diferentes.

QUADRO 30 - ESTRUTURA DE PACOTES



FONTE: O autor

Ao tentar instanciar os objetos, o Eclipse nos avisa de um erro, informando que Correntista e ContaCorrente não são tipos de dados, como se essas classes não existissem. O problema é que, pelo menos por enquanto, para o compilador do Java elas realmente não existem, pois não estão no pacote corrente. Elas são públicas e, portanto visíveis em todo o projeto, entretanto não antes de realizarmos uma pequena operação. No Quadro 31 podemos perceber de que forma o Eclipse destaca o erro.

QUADRO 31 - ERRO DE COMPILAÇÃO DESTACADO NO ECLIPSE

```

1
2 public class Testador {
3
4     public static void main(String[] args) {
5
6         Correntista c = new Correntista("Joao da Silva",
7                                         "Rua XV de no 100");
8
9         ContaCorrente cc = no 100;
10    }
11
12 }
```

A screenshot of the Eclipse IDE code editor showing a compilation error. On line 6, the variable 'c' is assigned an object of type 'Correntista'. The word 'Correntista' is underlined with a red squiggly line, indicating it cannot be resolved to a type. A tooltip appears over the underlined text with the message 'Correntista cannot be resolved to a type' and '3 quick fixes available'. The quick fix options listed are: 'Import 'Correntista' (dominio)' (with a blue arrow icon), 'Create class 'Correntista'' (with a green circle icon), and 'Fix project setup...' (with a yellow triangle icon). The status bar at the bottom right says 'Press F2 for focus'.

FONTE: O autor

Na linha 6, ao colocarmos o cursor sobre a palavra Correntista, o Eclipse inicialmente nos mostra a mensagem de erro do compilador e em seguida apresenta três correções possíveis, no entendimento dele. Esta correção rápida apresentada pelo Eclipse é uma ferramenta extremamente poderosa, e neste caso acertou na mosca. Para que possamos “ver” as classes Correntista e ContaCorrente, precisamos

antes fazer a importação das mesmas. O comando Import, colocado sempre abaixo do nome do pacote, traz a possibilidade de acessarmos classes públicas oriundas de outros pacotes. TODAS as classes do próprio Java que não fazem parte do *core* são acessadas exatamente da mesma forma, ou seja, caso você queira se conectar a um banco de dados, a uma rede, desenhar uma tela, fazer cálculos matemáticos complexos, entre outras operações usando classes do Java, você deverá antes usar o comando Import. O Quadro 32 mostra nas linhas 1 e 2 de que forma fica a classe após a colocação dos imports. Lembrando que sempre existe a opção de utilizar o nome completo da classe (*full qualified name*), bastando colocar o nome do pacote na frente do nome da classe. Ao invés de fazer os imports, você sempre escreverá domínio. Correntista e domínio.ContaCorrente, prática que funciona embora prejudique a legibilidade do código.

QUADRO 32 - CLASSE APÓS A COLOCAÇÃO DOS IMPORTS

```

1 import dominio.ContaCorrente;
2 import dominio.Correntista;
3
4
5 public class Testador {
6
7     public static void main(String[] args) {
8
9         Correntista c = new Correntista("Joao da Silva",
10             "Rua XV de novembro");
11
12         ContaCorrente cc = new ContaCorrente(2331, c);
13
14         cc.depositar(4599);
15     }
16
17 }
```

FONTE: O autor

Para exemplificarmos o modificador *default*, marcamos a classe MecanismoDeAutenticacao com esse modificador (Quadro 33). Na verdade, a visibilidade *default* pode ser utilizada ainda para métodos e atributos, embora faça mais sentido com classes. O *default* implica visibilidade de pacote, ou seja, uma classe com modificador *default* somente é visível dentro do pacote onde ela estiver. No caso de atributos e métodos, o princípio é o mesmo. Dentro desta classe criamos um atributo e um método com o mesmo modificador, lembrando que a ausência de modificador indica visibilidade *default*.

É preciso atentar para o modificador *default* em métodos e atributos, visto que o modificador da classe tem prioridade. Por exemplo, uma classe pública com métodos *default* poderá ser vista em toda a aplicação, mas seus métodos somente dentro do mesmo pacote. Por outro lado, uma classe *default* com métodos públicos somente dará acesso a estes métodos dentro do pacote, visto que para invocar um método é preciso criar o objeto antes.

QUADRO 33 - CLASSE COM MODIFICADOR DEFAULT

```

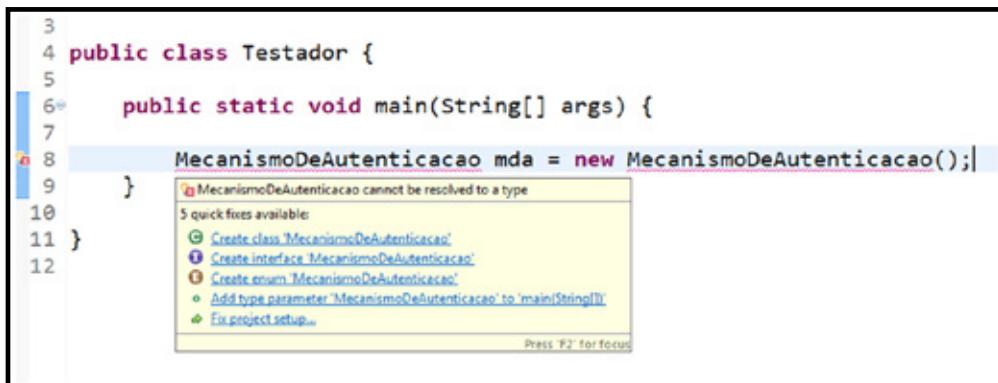
1 package autenticacao;
2
3 class MecanismoDeAutenticacao {
4
5     String nivelDeSeguranca;
6
7     void setNivelDeSeguranca(String nivel){
8         nivelDeSeguranca = nivel;
9     }
10
11 }

```

FONTE: O autor

Caso queiramos instanciar um objeto do tipo `MecanismoDeSeguranca` na classe `Testador`, temos o mesmo erro indicado pelo Eclipse no Quadro 31, entretanto, na sugestão de correção não existe a opção de importação. Por que isto ocorre? Ao marcarmos a classe com o modificador *default*, restringimos sua visibilidade ao pacote, ou seja, ela NÃO ESTÁ DISPONÍVEL para utilização em outros pacotes, mesmo com importação.

QUADRO 34 - INSTANCIACÃO DA CLASSE MECANISMOAUTENTICACAO



FONTE: O autor

É preciso atentar para a colocação dos modificadores em classes. Da mesma forma que uma classe que deve ser usada em outros pacotes deve ser marcada com o modificador *public*, uma classe que somente será usada dentro de um pacote deve ser marcada com o *default*, respeitando, desta forma, o princípio de baixo acoplamento e restringindo o acesso à mesma.

RESUMO DO TÓPICO 1

Neste tópico vimos:

- Os três princípios fundamentais da Programação Orientada a Objetos são: encapsulamento, herança e polimorfismo.
- O encapsulamento significa ocultar dados e funções, de forma que não sejam acessíveis externamente.
- O encapsulamento é importante, pois possibilita que reutilizemos código sem nos preocuparmos com detalhes de implementação.
- A camada que permite acesso a um objeto é conhecida como interface externa deste objeto.
- Coesão e acoplamento são conceitos geralmente relacionados ao encapsulamento.
- Alta coesão significa projetar as classes com uma e somente uma responsabilidade principal.
- Baixo acoplamento significa reduzir ao máximo os pontos de comunicação entre as classes. Essa prática é positiva, pois em caso de manutenção de uma classe, quanto mais desacopladas forem as demais, menor a probabilidade de que estas tenham que ser mantidas em cascata.
- Na linguagem de programação Java, o encapsulamento é implementado através da colocação de modificadores na frente de classes, atributos ou métodos.
- Os modificadores de visibilidade disponíveis no Java são: private, public, protected e default.
- A visibilidade private, quando aplicada a atributos ou métodos, restringe o acesso dos mesmos para somente dentro da classe onde foram definidos. O modificador private só é aplicado em classes quando criamos classes internas.
- O modificador public garante o acesso a classes, métodos ou atributos em toda a aplicação.
- O modificador protected é utilizado somente quando implementamos herança.
- O modificador default, que restringe o acesso a classes, métodos ou atributos para somente dentro do pacote onde estiverem definidos, é implementado através da não colocação de modificadores.

- O acesso a atributos privados dentro de uma classe deve ser feito através de métodos acessores (Getters ou Setters).
- Quando colocamos um atributo de um tipo de classe dentro de outra classe, fazemos um relacionamento de associação.
- Classes públicas são visíveis em toda a aplicação, embora sua utilização em pacotes diferentes exige a utilização do comando import.

AUTOATIVIDADE



1 Assinale a alternativa CORRETA:

- a) O encapsulamento não pode ser definido em nível de classe, pois não pode existir uma classe private.
- b) Existem três modificadores de visibilidade no Java:
 - public libera o acesso do item em questão (classe, método, atributo) a TODAS as demais classes da aplicação;
 - implemented libera o acesso do item em questão (classe, método, atributo) somente a classes que forem derivadas (filhas) da classe onde o modificador foi aplicado;
 - private libera o acesso do item em questão (método, atributo) somente a classe onde ele está inserido.
- c) Um mesmo arquivo fonte escrito em Java não pode ser compilado em diferentes sistemas operacionais, cada um com sua máquina virtual específica.
- d) Encapsulamento consiste em ocultar parte de seus dados do resto de sua aplicação e limitar a possibilidade de outras partes de seu código acessarem esses dados. Ao invés de ter um programa como uma entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes, facilitando manutenções futuras.

2 Descreva de que forma a alta coesão e o baixo acoplamento contribuem para que seja mais fácil fazer a manutenção de programas de computador.

3 Assinale a alternativa CORRETA:

- a) O encapsulamento é implementado através de modificadores de visibilidade. Ao identificar uma classe, método ou campo com eles, define-se o que será oculto e o que será visível as demais classes da aplicação.
- b) Uma classe pode conter somente um método construtor.
- c) A linguagem de programação Java é considerada multiplataforma pelo fato de podermos escrever programas Java tanto em um editor de texto comum quanto em IDEs mais elaboradas, como por exemplo Eclipse ou Netbeans.
- d) Um mesmo arquivo .class Java que foi compilado e transformado em *bytecode* no Linux pode ser simplesmente copiado e executado em qualquer outro sistema operacional, mesmo sem máquina virtual (JVM).

4 Observe as afirmações a seguir classificando-as em verdadeiras (V) ou falsas (F):

- () Uma classe somente pode ser constituída por um nome e por um conjunto de métodos que definem o seu comportamento.
- () Uma classe não pode conter mais de um método construtor, pois no momento da instanciação o compilador não saberia qual construtor chamar.
- () Uma classe pode possuir uma variedade especial de métodos que são chamados de construtores. A principal funcionalidade desses métodos é construir a classe, atribuindo valores aos campos etc.
- () Encapsulamento consiste em colocar TODOS os seus atributos e métodos como *private*, impedindo o acesso a estes por outras partes de sua aplicação. Ao invés de ter um programa como uma entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes, facilitando manutenções futuras.
- () Uma variável local em Java sempre deve ser inicializada e pode conter qualquer modificador (*private*, *public*, *protected*) pois seu escopo se alterará de acordo com o modificador aplicado.

5 Explique detalhadamente porque não faz sentido colocar o modificador *public* em um método dentro de uma classe com visibilidade *default*.



HERANÇA

1 INTRODUÇÃO

No tópico anterior estudamos o primeiro dos três pilares que dão sustentação para a programação orientada a objetos, o encapsulamento.

No Tópico 2 abordaremos a herança, prática extremamente poderosa no que se refere à reutilização de código, mas que deve ser usado com parcimônia, sob o risco de aumentar o acoplamento entre as classes que fazem parte deste tipo de relacionamento. Veremos aqui os principais critérios adotados para julgar se uma utilização é ou não passível de herança, bem como as armadilhas que a utilização excessiva da prática ocasiona no código.

2 HERANÇA

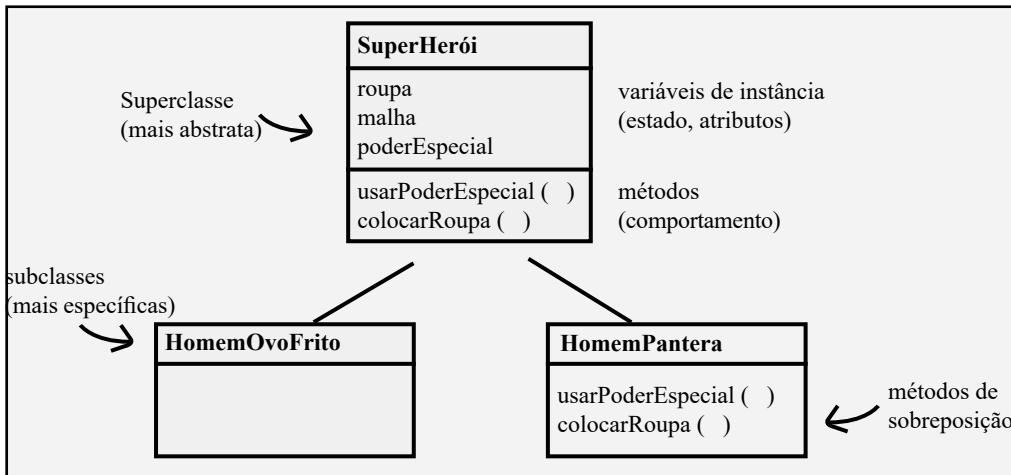
Sintes (2002) coloca que a herança permite que você reproveite o código já existente em uma classe previamente escrita. Usando a herança, sua nova classe pode herdar atributos e comportamentos que forem interessantes para ela, permitindo que sua interface externa seja exatamente igual à da classe que foi herdada. Conforme veremos no Tópico 3 desta unidade, a herança permite à classe que está herdando redefinir qualquer comportamento que não seja adequado. Este comportamento permite que você altere seu *software* com o mínimo de esforço, quando seus requisitos mudarem. Caso você precise fazer uma alteração, basta escrever uma classe que herde a antiga funcionalidade e então a sobrepor, o que permite que se altere a maneira como uma classe se comporte sem tocar na classe original. O interessante é que você pode até mesmo reproveitar o comportamento de uma classe que você não tenha o código-fonte.



A sobreposição é o mecanismo da programação orientada a objetos, que envolve herança e polimorfismo que permitem que se redefina o comportamento de um método herdado, sem alterar o comportamento da classe original.

A Figura 36, extraída de Bates e Sierra (2010), ilustra como ficaria uma hierarquia de herança de classes.

FIGURA 36 - REPRESENTAÇÃO DA HERANÇA



FONTE: Bates e Sierra, 2010

Na figura podemos perceber que a Classe SuperHerói é chamada de superclasse, enquanto as classes HomemOvoFrito e HomemPantera são chamadas de subclasses. Outra nomenclatura comum é chamar a superclasse de classe mãe e as subclasses de classes filhas. Quanto mais acima na hierarquia, mais genérica é a definição da classe e quanto mais abaixo, mais específica. Por esse motivo, o relacionamento de herança é frequentemente denominado por generalização – especialização.

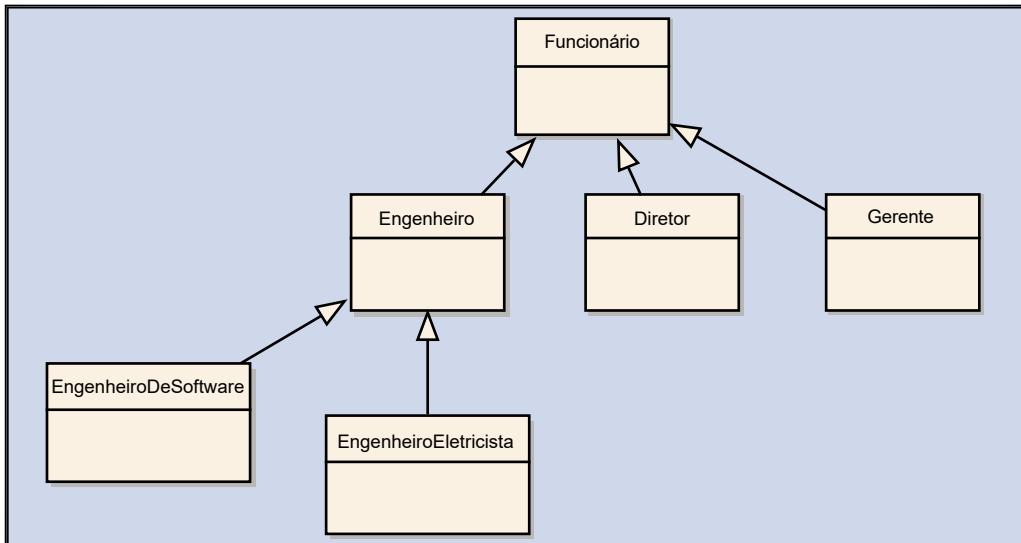
Neste exemplo, o HomemOvoFrito tem os mesmos atributos e métodos de super-herói (genérico), não sobrepondo nenhum deles. Já o HomemPantera apresenta uma implementação específica para usarPoderEspecial() e colocarRoupa(). Não existe sobreposição em variáveis de instância (atributos), mas ambas as subclasses poderiam adicionar outras variáveis, caso desejassem.

Na linguagem de programação Java não existe implementação de herança múltipla, o que por conseguinte, determina:

- Uma classe mãe pode ter quantas filhas quiser, seja na hierarquia vertical ou horizontal.
- Uma classe filha pode ter uma e somente uma classe mãe.

A não implementação da herança múltipla na linguagem de programação Java foi uma decisão de projeto, visando evitar o conhecido problema do diamante, que se resume à seguinte situação: caso uma classe filha possa herdar de duas classes mães e estas classes mães tenham um método com o mesmo nome, qual dos dois métodos será implementado pela classe filha? A Figura 37 ilustra um diagrama de classes com herança em níveis verticais e horizontais.

FIGURA 37 - DIAGRAMA DE CLASSES MOSTRANDO HERNAÇA



FONTE: O autor

Na Figura 37, a classe **Funcionario** é a Superclasse da relação, tendo três filhas diretas horizontalmente, a classe **Engenheiro**, a classe **Diretor** e a classe **Gerente**. Já a classe **Engenheiro** acaba por possuir também duas filhas, **EngenheiroDeSoftware** e **EngenheiroEletricista**. Nesta situação específica, **EngenheiroDeSoftware** e **EngenheiroEletricista** herdam os atributos e os métodos de **Engenheiro** e de **Funcionario**, respectivamente. Em geral, as superclasses não são instanciadas, pois não existe um **Engenheiro**, simplesmente, mas sim um engenheiro especialista em alguma área do conhecimento.

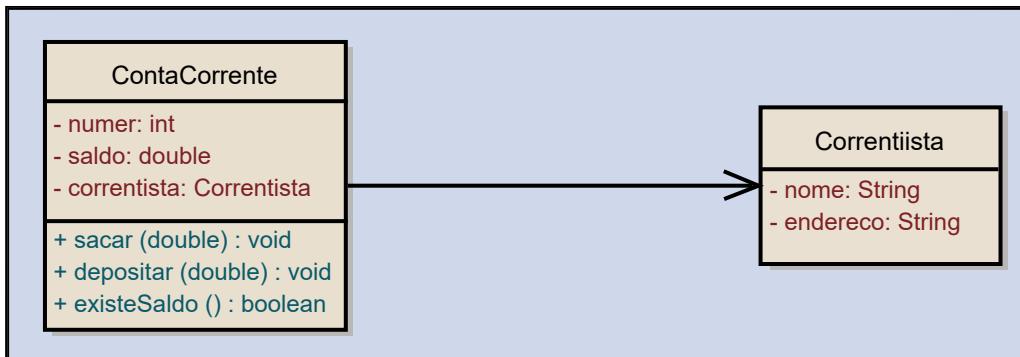


Mais à frente abordaremos com maiores detalhes o caso acima citado.

Mas como posso determinar se o problema computacional deve ser representado em meu sistema através de uma hierarquia de herança? Existe uma regra que funciona na maioria das situações, especialmente enquanto estamos iniciando o aprendizado de programação orientada a objetos. Sintes (2002) afirma que, para descobrir se o relacionamento entre duas classes pode ser representado através da herança, basta perguntar se a classe filha é do tipo da classe mãe. Por exemplo, HomemOvoFrito é um tipo de super-herói? SIM, então você PODE usar herança (embora isso não signifique que DEVA), colocando HomemOvoFrito como subclasse e SuperHerói como superclasse. EngenheiroEletricista é um tipo de Engenheiro? SIM, então existe possibilidade de usar herança, da mesma forma que no relacionamento entre EngenheiroEletricista e Engenheiro.

Outro relacionamento bastante comum entre duas classes é o relacionamento de associação, onde uma classe TEM UM atributo de outra classe. Esse relacionamento é caracterizado exatamente por essa pergunta: TEM UM? Como exemplo, podemos citar o relacionamento ilustrado no Tópico 1 onde existe uma instância de Correntista dentro de ContaCorrente, explicitando que ContaCorrente TEM UM Correntista (Figura 38).

FIGURA 38 - RELACIONAMENTO DE ASSOCIAÇÃO



FONTE: O autor

Antes de efetivamente abrir, o Eclipse perguntará a você qual o local onde deseja colocar o *workspace*, conforme Figura 17. O *workspace* é o local onde seus projetos serão armazenados com todos os códigos fontes pelo Eclipse. Sugerimos fortemente que você selecione um local para o *workspace* que seja de fácil acesso e *backup* constante. Em nosso exemplo, o *workspace* foi colocado dentro da pasta do usuário crfranco, que é o mesmo local onde o Windows armazena os documentos, imagens, vídeos etc. Caso você deseja que esta tela de configuração não apareça mais, basta clicar no *checkbox* da parte inferior da mesma, onde está escrito: *Use this as the default and do not ask again*. Ao selecionar esta opção, o Eclipse entende que sempre que abrir, o *workspace* será aquele que já estiver configurado.

2.1 MECÂNICA DA HERANÇA

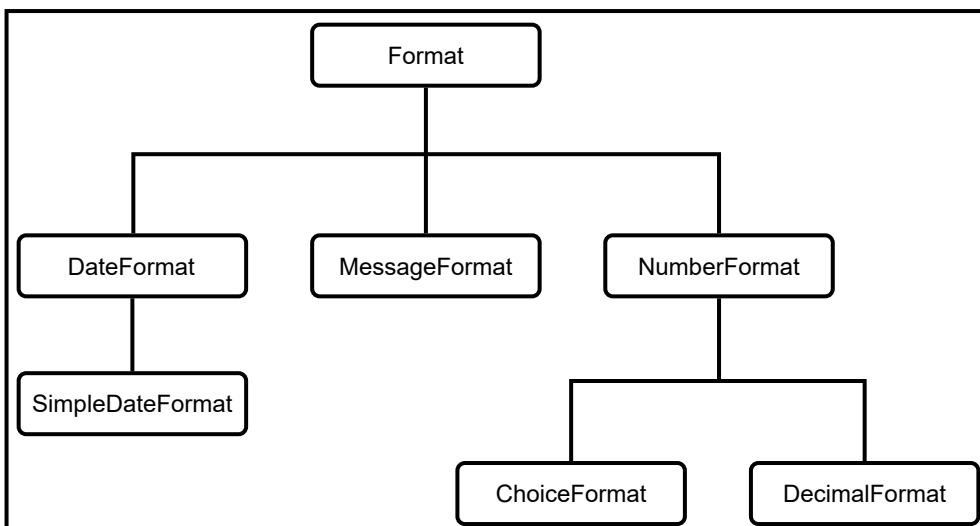
Quando uma classe herda de outra, ela herda implementação, atributos e comportamento. Isso significa que todos os métodos e atributos disponíveis na interface externa da classe mãe estarão também na interface externa da filha (SINTES, 2002). Uma classe construída através de herança pode ter três tipos importantes de métodos e atributos:

1. Sobreposto: a nova classe herda o método da progenitora, mas fornece uma nova definição.
2. Novo: a nova classe adiciona um método ou atributo completamente novo.
3. Recursivo: a nova classe simplesmente herda um método ou atributo da classe mãe.

FONTE: Disponível em: <http://www/usr.inf.ufsm.br/~rose/curso3/cafe/cap3_Heranca.pdf>. Acesso em: 28 out. 2014.

A própria API do Java implementa a herança em diversos locais, como pode ser visto na figura a seguir:

FIGURA 39 – API do Java



FONTE: Adaptado de: Sintes (2002)



Se você verificar que uma classe filha precisa remover funcionalidades ou atributos, isso é um forte sinal de que sua hierarquia de herança deve ser revista.

Conforme Sintes (2002), existem três principais maneiras para se utilizar a herança:

- 1) Reutilização de implementação: neste tipo de herança buscamos simplesmente reutilizar alguma funcionalidade já existente, sem precisar copiar e colar código-fonte. O cuidado a ser tomado com a herança de reutilização é com a tipagem, uma vez que ao herdar de uma classe mãe, a classe filha automaticamente pode ser tipada como a classe mãe. Isso significa que simplesmente reutilizar uma funcionalidade não é motivo suficiente para estabelecer uma hierarquia de herança. Lembre-se da pergunta: Classe filha é um tipo de Classe mãe?
- 2) Diferença: na programação por diferença simplesmente adicionamos atributos e comportamentos complementares em uma Classe filha. Essa prática gera código menor e mais fácil de gerenciar e manter. Ao escrever somente atributos e métodos complementares sem alterar o código existente para classes já testadas, você acaba por escrever menos código e consequentemente, comete menos erro.
- 3) Substituição de tipo: Na herança por substituição, podemos substituir um tipo referenciado por uma subclasse por qualquer uma de suas subclasses. Por exemplo: em uma hierarquia de classes contendo uma Superclasse Funcionario e duas subclasses FuncionarioHorista e FuncionarioComissionado, qualquer referência feita a Funcionario permite sua substituição pelas subclasses FuncionarioComissionado, FuncionarioHorista e até mesmo qualquer outro tipo de Funcionario que venha a ser criado como subclasse. Esse comportamento é poderoso e dá base para a implementação de código-fonte flexível e fácil de ser mantido.

2.2 DICAS PARA FAZER UMA HERANÇA EFICAZ

Sintes (2002) fornece algumas dicas para que você possa utilizar esse mecanismo da programação orientada a objetos de forma mais eficaz:

- 1) Em geral, use herança para reutilização de interface e para definir relacionamentos de substituição.
- 2) Sempre use a regra “é um”.
- 3) Como regra geral, mantenha suas hierarquias de classe relativamente rasas.

- 4) Projete cuidadosamente sua hierarquia de herança para evitar ao máximo a quebra de encapsulamento.
- 5) As classes frequentemente compartilham código comum. Não há sentido em ter várias cópias de código. Você deve remover o código comum e isolá-lo em uma única classe mãe logo acima na hierarquia.
- 6) Simplesmente não é possível planejar suas hierarquias completamente. As características comuns não aparecerão tão facilmente assim, mas quando aparecerem, não tenha medo de reescrever suas classes. O trabalho se pagará em um momento de manutenção.
- 7) Nunca se esqueça de que a substituição é o objetivo principal. Mesmo que um objeto deva intuitivamente aparecer em uma hierarquia, isso não quer dizer que ele deve aparecer.
- 8) Programe pela diferença para manter o código mais fácil de gerenciar e manter.

Como já dissemos, “a herança é um mecanismo extremamente poderoso, mas que como todo poder, traz junto uma grande responsabilidade” (BEN, FILME DO HOMEM ARANHA).



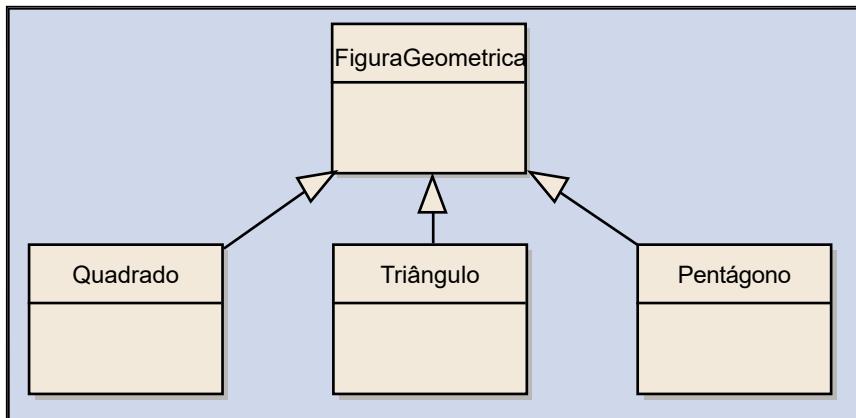
Mais à frente aprenderemos a utilizar uma alternativa bastante recomendável a alguns casos de herança, conhecida como composição.

2.3 IMPLEMENTAÇÃO DA HERANÇA NA LINGUAGEM DE PROGRAMAÇÃO JAVA

Da mesma forma que o encapsulamento, a herança é implementada na linguagem de programação Java através de modificadores e palavras-chave. Imaginemos um relacionamento de herança conforme o da Figura 12. Neste relacionamento, a classe FiguraGeométrica é a superclasse ou classe mãe e as classes Quadrado, Triângulo e Pentágono são subclasses ou classes filhas. Podemos perceber que todas elas passam no teste da pergunta “é um tipo de” e que faz sentido o relacionamento de herança, pois todas elas compartilham características em comum.

Programaticamente, fazemos uso da palavra reservada `extends`, para indicar que uma classe é uma extensão de outra, ou seja, que herda estado e comportamento da interface pública de uma superclasse. Os quadros 15 e 16 mostram essa situação na prática.

FIGURA 40 – RELACIONAMENTO DE HERANÇA



FONTE: Adaptado de: Sintes (2002)

QUADRO 35 – SUPERCLASSE

```
1  
2 public class FiguraGeometrica {  
3  
4 }  
5
```

FONTE: O autor

Perceba que não há diferença na classe mãe, pois a marcação é feita nas classes filhas através da palavra reservada extends, destacada no Quadro 16. A partir deste momento, o relacionamento de herança está estabelecido entre FiguraGeometrica e Quadrado.

QUADRO 36 - SUBCLASSE

```
1  
2 public class Quadrado extends FiguraGeometrica {  
3  
4 }  
5
```

FONTE: O autor

Para entendermos de que forma a herança se comporta em relação aos atributos e métodos, vamos implementar um exemplo um pouco mais complexo. Inicialmente vamos utilizar uma classe Aluno, que tem como atributos a matrícula, o nome e o curso, além de métodos getters para todos e setters somente para o nome e o curso. Você consegue pensar em algum motivo para não colocarmos o acessor na matrícula? Se considerarmos que a matrícula funcionará como um identificador para o aluno, mais ou menos como um CPF, entenderemos que ela não deve sofrer alterações e que deve ser atribuída somente no momento da matrícula do aluno, que no nosso caso é a criação do objeto. Para atender esse requisito, criamos um construtor que recebe os três atributos como parâmetro.

Essa classe será nossa superclasse e terá duas classes filhas, uma representando um aluno de graduação e outra representando um aluno de pós-graduação. O código-fonte parcial da classe mãe pode ser visualizado no Quadro 37.

QUADRO 37 - CLASSE ALUNO ABREVIADA

```

4 public class Aluno {
5
6     private int matricula;
7     private String nome;
8     private String curso;
9
10    public Aluno(int matricula, String nome, String curso) {
11
12        this.matricula = matricula;
13        this.nome = nome;
14        this.curso = curso;
15    }
16
17    public String getNome() {
18    public void setNome(String nome) {
19    public String getCurso() {
20    public void setCurso(String curso) {
21    public int getMatricula() {
22
23
24
25
26
27
28
29
30
31
32

```

FONTE: O autor

O relacionamento de herança dá visibilidade para os métodos e atributos que estiverem na interface pública do objeto, o que permite aos filhos acesso aos atributos somente através de seus acessores. Vamos demonstrar essa característica da herança no Quadro 38, mas antes abordaremos os construtores em relacionamentos de herança.

Uma subclasse é, em essência, uma especialização de uma superclasse, o que implica um compartilhamento e reutilização direta de código-fonte. Essa facilidade, entretanto, traz algumas restrições. Por exemplo, se a superclasse tiver um ou mais construtores, a subclasse é obrigada a fornecer uma nova implementação de pelo menos um destes e neste, chamar o construtor da superclasse explicitamente. Na linguagem de programação Java, a não obediência desta restrição gera um erro de compilação.

No Quadro 38 está a classe `AlunoDeGraduacao`, que é subclasse de `Aluno`. Na linha 4 podemos ver que ela fornece uma implementação particular para o construtor de `Aluno`. O primeiro comando dentro deste construtor deve ser a chamada para o construtor da superclasse, representada pela palavra **super**, destacada na linha 5. Perceba que a linha 5 parece realmente uma chamada de método, onde os parâmetros `matricula`, `nome` e `curso` são repassados para o construtor da superclasse. Lembre-se de que a não realização destes passos resulta em erro de compilação.

Ainda no Quadro 38, tentamos fazer uma chamada fictícia aos métodos e atributos da superclasse (linha 6) para demonstrar o relacionamento de herança. Perceba que NÃO temos acesso aos atributos da classe, pois estes não fazem parte da interface pública de `Aluno`. As variáveis `matricula`, `nome` e `curso` que vemos são as variáveis locais que vieram como parâmetro do método. Por outro lado, os métodos *getters* estão completamente visíveis. Agora você pergunta: mas de onde o `AlunoDeGraduacao` está buscando esses métodos? A resposta é: diretamente de `aluno`, reutilizando o código sem reescrever uma linha sequer. Esse é o poder que a herança traz para nossos códigos.

Existe um modificador de acesso que permite fazer um controle de encapsulamento mais fino dentro de nossas hierarquias de herança. Ele é representado pela palavra-chave `protected`. Ao colocarmos o modificador `protected` em um método ou atributo, isso implica que o mesmo somente será visto por seus descendentes e que lá, o acesso a estes é *private*. Isso permite que se tenha um maior controle sobre as operações com os atributos, visto que se os mantivermos como *private*, seu acesso se restringe aos métodos acessores. Tal situação é demonstrada no Quadro 39, onde fazemos exatamente a mesma experiência do Quadro 38, só que desta vez com os atributos da classe mãe utilizando `protected` ao invés de `private`.

QUADRO 38 - CONSTRUTOR NA CLASSE FILHA

```

2 public class AlunoDeGraduacao extends Aluno {
3
4     public AlunoDeGraduacao(int matricula, String nome, String curso) {
5         super(matricula, nome, curso);
6     }
7
8 }
9
10

```

FONTE: O autor

QUADRO 39 - ACESSO AOS ATRIBUTOS PROTEGIDOS

The screenshot shows a Java code editor with the following code:

```

2 public class AlunoDeGraduacao extends Aluno {
3
4     public AlunoDeGraduacao(int matricula, String nome, String curso) {
5         super(matricula, nome, curso);
6
7     }
8
9 }
10

```

A code completion dropdown is open at the end of the constructor call. It lists several methods and fields from the `Aluno` interface, including `getNome()`, `setNome(String nome)`, `getMatricula()`, `setMatricula(int matricula)`, `getCurso()`, and `setCurso(String curso)`. Below these, it shows the constructor and the `this` reference pointing to the current object.

FONTE: O autor

Agora temos acesso a todos os métodos da interface pública de `Aluno` e ainda poderemos acessar diretamente seus atributos, tendo em mente que estes são privados em `AlunoDeGraduacao`. A demonstração desta característica pode ser observada no Quadro 20, onde utilizamos um testador para instanciar um `AlunoDeGraduacao` e tentar chamar seus métodos.

QUADRO 40 - TESTE DA SUBCLASSE

The screenshot shows a Java code editor with the following code:

```

6 public class Testador {
7
8     public static void main(String[] args) {
9
10         Aluno a = new AlunoDeGraduacao(1233, "Peter Parker", "Biologia");
11
12     }
13
14 }
15
16

```

A code completion dropdown is open at the end of the assignment statement. It lists methods from the `Object` class and the `Aluno` interface, such as `equals(Object arg0)`, `getClass()`, `getNome()`, `getMatricula()`, `getCurso()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()`, and `wait()`. The `a` reference is highlighted in red.

FONTE: O autor

Na linha 10 criamos um objeto `AlunoDeGraduacao` através de seu construtor e na linha 12 tentamos invocar o que estiver na interface pública do objeto. Perceba que os atributos não estão mais visíveis, pois como foram marcados como protegidos na classe `Aluno`, eles se tornam privados em qualquer classe que herde de `Aluno`.

Dando continuidade ao exemplo, vamos agora definir a classe AlunoDePosGraduacao, diferenciando-o dos demais pela inserção de um atributo que indique qual é a instituição de graduação do aluno.

QUADRO 41 - HERANÇA POR DIFERENÇA

```

2
3 public class AlunoDePosGraduacao extends Aluno{
4
5     private String instituicaoDeGraduacao;
6
7     public AlunoDePosGraduacao(int matricula, String nome, String curso,
8         String inst) {
9         super(matricula, nome, curso);
10
11         instituicaoDeGraduacao = inst;
12     }
13
14     public String getCursoDeGraduacao() {
15         return instituicaoDeGraduacao;
16     }
17
18 }
```

FONTE: O autor

Na linha 5 declaramos o atributo privado instituicaoDeGraduacao e nas linhas 7 e 8 colocamos este atributo no construtor da classe. A colocação deste atributo indica que, obrigatoriamente um aluno de pós-graduação deverá fornecer o nome da instituição onde se graduou. Destacamos que esta é uma opção nossa para ilustrar o uso dos construtores na herança e que a utilização de acessores para atribuir tal valor também seria válida. Nas linhas 14 a 16 está o *getter* do atributo recém-colocado.

A primeira coisa a ser feita no construtor de uma classe filha é a chamada para o construtor da classe mãe através da instrução *super* e do envio dos parâmetros necessários (linha 9). Em seguida, perceba que continuamos com a função normal do construtor, que é receber o parâmetro do método e o atribuir à variável de instância instituicaoDeGraduacao, neste caso. A chamada *super* desvia o fluxo de execução para a superclasse que, quando encerrar suas inicializações, retorna para a subclasse.

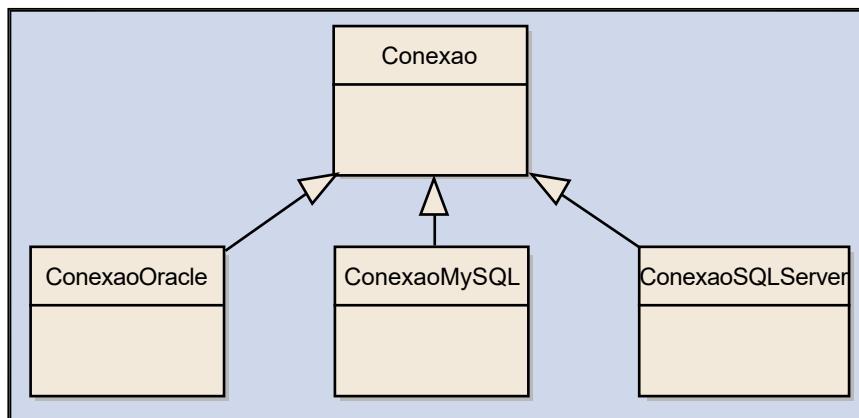
A utilização do modificador *protected* ao invés de *private* vai depender da sua necessidade de acessar diretamente os atributos na classe filha. Conforme vimos no tópico 1, acessar atributos de outra classe, mesmo que seja sua classe mãe, em geral não é uma boa ideia, então sugerimos fortemente a marcação dos atributos como *private* até que outra opção seja absolutamente necessária. Esse é um dos motivos pelos quais se diz que a herança viola o encapsulamento. A classe filha acaba por saber muitos detalhes da classe mãe. Um detalhe importante a ser destacado é que, na plataforma Java, a visibilidade *protected* também permite acesso por outras classes no mesmo pacote.



Na linha 10 do Quadro 40, você deve ter percebido que o tipo do objeto criado é Aluno, enquanto a referência é do tipo AlunoDeGraduacao. Esta é uma boa prática definida pelo Princípio de Substituição de Liskov, através do qual toda subclasse deve poder ser substituída por sua superclasse na tipagem, de modo a respeitar a hierarquia de herança. Para saber mais sobre esse assunto, acesse: <<http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple>>.

A preocupação com detalhes é extremamente importante quando se projeta uma hierarquia de herança, pois ao criar uma superclasse, você abre uma porta que é muito difícil de ser fechada. Por exemplo: digamos que você criou uma classe chamada Conexao, que estabeleça conexões com o banco de dados e que esta classe foi projetada para permitir que suas filhas façam reutilização do código necessário para a conexão. Cada banco de dados distinto exige a implementação de uma nova classe filha (Figura 41) que, logicamente, reutilizará o código-fonte da classe mãe.

FIGURA 41 – EXEMPLO DE HERANÇA PARA CONEXÃO COM BANCO DE DADOS



FONTE: O autor

A solução demonstrada no parágrafo anterior é viável e pode muito bem ser utilizada em um projeto de *software*. O objetivo do exemplo é demonstrar a dependência que os sistemas terão em relação a esta classe. Toda e qualquer operação que envolva banco de dados fará uso da mesma, e como você pode imaginar, a grande maioria dos *softwares* interage intensamente com bancos de dados. Praticamente todos os programadores de sua equipe farão uso de sua classe, direta ou indiretamente.

Percebeu o problema? A partir deste momento, você perde a liberdade de alteração de sua classe, pois muitas outras funcionalidades dependem dela. Toda e qualquer alteração deverá ser feita com extremo cuidado e ser seguida por uma bateria intensa de testes para não quebrar a compatibilidade.

Joshua Bloch em seu livro *Effective Java 2* (BLOCH, 2008), vai além, ao sugerir que todas as classes de sua aplicação que não são projetadas para serem superclasses em uma hierarquia de herança sejam impedidas de realizar tal papel através do modificador **final** (Quadro 42). Ao colocar este modificador na classe Quadrado, caso alguém tente estendê-la obterá um erro de compilação.

QUADRO 42 – MODIFICADOR FINAL

```

1
2 public final class Quadrado extends FiguraGeometrica {
3
4 }
5

```

FONTE: O autor

A utilização desta prática é vista por muitos como radical, entretanto, dependendo do tamanho da equipe de desenvolvedores, é muito fácil perder o controle sobre o que é ou não utilizado pelos outros. Com o tempo, você aprenderá a discernir o quanto defensivo deve ser seu código, mas inicialmente, enquanto estamos aprendendo, é seguro dizer que quanto mais defensivo melhor.

2.4 CLASSES ABSTRATAS

A hierarquia de classes representada pela Figura 40 apresenta uma característica existente em alguns relacionamentos de herança. Vemos uma figura geométrica genérica fornecer características para um quadrado, um triângulo e um pentágono. Mas o que é essa figura geométrica? Ela efetivamente existe? Quais são suas características?

Não existe figura geométrica genérica, como demonstrado na hierarquia. O que existe são quadrados, triângulos, pentágonos etc. A classe FiguraGeometrica somente faz parte da hierarquia como um agregador de características e comportamentos, não sendo efetivamente um objeto real dentro do sistema. O mesmo ocorre com a Figura 41, onde para existir, uma conexão deve obrigatoriamente estar associada a um banco de dados.

Quando encontramos este tipo de situação, dizemos que a superclasse da hierarquia é abstrata, ou seja, NÃO PODE SER INSTANCIADA. Mas para que serve uma classe que não pode ser instanciada em um objeto? Neste tipo de situação, exatamente para agregar estados e comportamentos comuns a uma família de objetos. Perceba que não poder instanciar a FiguraGeometrica sem definir exatamente o que ela é garante a integridade conceitual do sistema, afinal, como já mostramos, não existe uma FiguraGeometrica, e sim especializações da mesma.



As classes abstratas têm ainda uma importante função, que é permitir a utilização de métodos abstratos. Os métodos abstratos são uma das formas de se implementar o polimorfismo e daremos mais destaque a essas questões no Tópico 3 desta unidade.

Para implementarmos uma classe abstrata, basta colocar o modificador `abstract` antes da palavra reservada `class`, conforme Quadro 43. A partir deste momento, essa classe se torna impossível de instanciar, gerando um erro de compilação. Conceitualmente falando, é como se o modificador `abstract` fosse o contrário do modificador `final`, pois enquanto o `final` impede que se estenda uma classe, o `abstract` praticamente exige, pois caso contrário, sua utilização é absolutamente inútil. Para as classes filhas, a utilização do `abstract` como modificador para a classe é absolutamente transparente. A influência sobre as filhas ocorre quando utilizamos o modificador `abstract` em métodos.

QUADRO 43 - DEFINIÇÃO DE CLASSE ABSTRATA

```

1
2 public abstract class FiguraGeometrica {
3
4 }
5

```

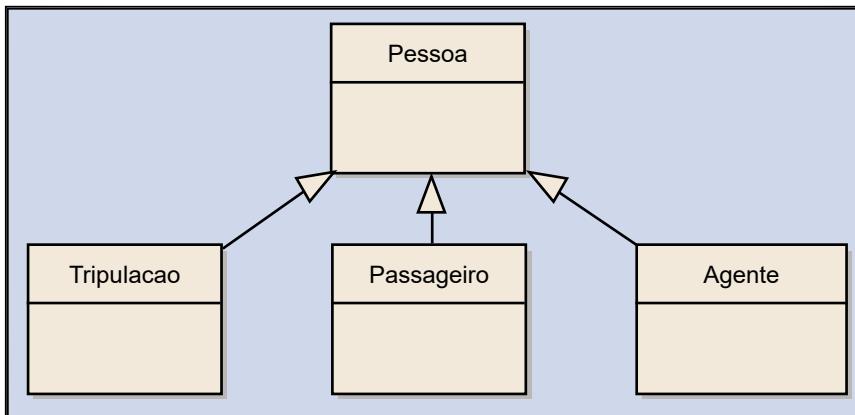
FONTE: O autor

2.5 COMPOSIÇÃO AO INVÉS DE HERANÇA

Conforme Bloch (2008), o relacionamento de herança acaba por violar o encapsulamento, pois a subclasse sempre tem que saber detalhes demais da subclasse, diminuindo a coesão de cada classe e fortalecendo o acoplamento entre ambas. Uma das soluções para evitar estes problemas da herança e ainda assim fazer reutilização de código é a técnica conhecida como composição.

Na composição reutilizamos código de outras classes não através da extensão, mas sim através da delegação. A classe que é composta por outra delega a realização de determinado trabalho para a classe que a compõe. Para exemplificarmos, considere uma hierarquia de herança conforme a demonstrada na Figura 42.

FIGURA 42 – HIERARQUIA DE HERANÇA



FONTE: Adaptado de: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/pat/herancavscomposicao.htm>>. Acesso em: 20 out. 2014.

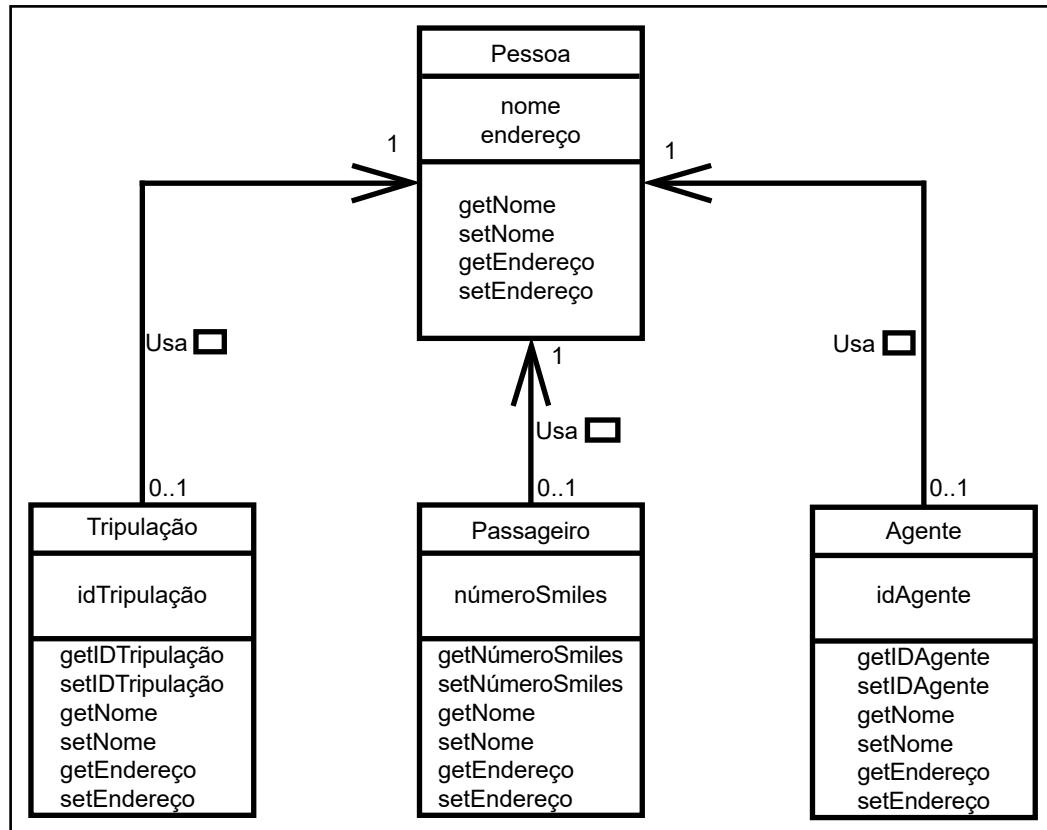
Nesta hierarquia, temos a superclasse Pessoa e as subclasses Tripulacao, Passageiro e Agente. Ao fazermos a pergunta “é um tipo de” tudo parece correto, o que indica que a herança é uma boa alternativa. O problema é que, nesta situação específica, uma Pessoa pode acabar trocando de papéis ou mesmo assumindo mais de um papel, por exemplo, quando um Agente viaja como Passageiro. Neste caso, qual das classes utilizamos? Além disso, continuamos com o problema da quebra do encapsulamento, pois tanto Tripulacao, quanto Passageiro, quanto Agente conhecem detalhes demais da classe Pessoa.

Para solucionar este problema utilizando a composição, faríamos o seguinte:

- As classes Tripulacao, Passageiro e Agente existiriam de forma independente.
- As três classes referenciariam um objeto Pessoa por associação.
- Para acessar os atributos e métodos da classe Pessoa, as três classes fariam uso da delegação. Por exemplo, digamos que o objeto tripulação precisa saber seu nome. A classe Tripulacao não possui tal atributo, entretanto, é possível fazer um método chamado getName e delegar a esse método o trabalho de buscar o nome do objeto Pessoa, retornando-o.
- Através dessa prática, o comportamento do objeto pode ser escolhido em tempo de execução ao invés de em tempo de compilação, como ocorre na herança.

O resultado da mudança pode ser visto na Figura 43. Como resultado da composição, temos o reaproveitamento do código já escrito em Pessoa sem os problemas típicos da herança. Perceba que mantivemos a coesão, pois nenhuma das três classes conhece detalhes da classe Pessoa e, quando precisamos reutilizar algo de pessoa, o fazemos via delegação. Consequentemente, as classes estão com baixo acoplamento e com o encapsulamento intacto. Uma prova disso é que, caso Pessoa sofra manutenção, a consequência disso em Tripulacao, Agente ou Passageiro é praticamente nula.

FIGURA 43 - COMPOSIÇÃO



FONTE: Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/pat/heranca-composicao.htm>>. Acesso em: 20 out. 2014.

Para ilustrarmos de outra forma, o código-fonte necessário para a composição está nos quadros 44 e 45.

QUADRO 44 - CLASSE PESSOA

```
2 public class Pessoa {  
3  
4     private String nome;  
5     private String endereco;  
6  
7     public String getNome() {  
8         return nome;  
9     }  
10    public void setNome(String nome) {  
11        this.nome = nome;  
12    }  
13    public String getEndereco() {  
14        return endereco;  
15    }  
16    public void setEndereco(String endereco) {  
17        this.endereco = endereco;  
18    }  
19 }
```

FONTE: O autor

No Quadro 44 podemos perceber a classe Pessoa, criada com os atributos nome e endereço e acessores para ambos. Já no Quadro 45, a classe Passageiro tem dois atributos, um numeroSmiles (linha 3) e uma Pessoa (linha 4). Aí é que observamos a composição. A classe Passageiro é composta por um objeto do tipo Pessoa. Perceba que um objeto Passageiro não tem nome diretamente, mas através da composição, reutilizamos o atributo nome da classe Pessoa. Os métodos setNome e getNome da classe Passageiro **delegam** a responsabilidade para os métodos homônimos do objeto pessoa, ou seja, se o passageiro precisa de um nome, ele chama o método getNome do objeto pessoa, adicionado por composição.

QUADRO 45 - CLASSE PASSAGEIRO

```

2 public class Passageiro {
3     private int numeroSimles;
4     private Pessoa pessoa;
5
6     public Passageiro(Pessoa p){
7         pessoa=p;
8     }
9
10    public void setNome(String nome){
11        pessoa.setNome(nome);
12    }
13
14    public String getNome(){
15        return pessoa.getNome();
16    }
17
18    public int getNumeroSimles() {□
19    public void setNumeroSimles(int numeroSimles) {□
20
21
22
23
24 }
```

FONTE: O autor

Um detalhe importantíssimo é a forma pela qual a classe terá acesso ao objeto que a comporá. No Quadro 45 podemos perceber que a classe Passageiro recebe um objeto do tipo Pessoa dentro do construtor, ou seja, injetamos um objeto pessoa dentro do objeto passageiro e a partir daí a composição está pronta para uso. Outra maneira bastante comum é a injeção através de um método *setter*. No exemplo teríamos um método setPessoa, que receberia um objeto pessoa como parâmetro. Você consegue perceber a diferença entre as duas abordagens? Na primeira, forçamos o desenvolvedor a fornecer um objeto pessoa já na construção de passageiro, enquanto na segunda, o desenvolvedor fornecerá um objeto pessoa somente se utilizar o método *setter*.

Para encerrarmos este tópico, novamente destacamos que a herança é um recurso poderoso, mas envolto de questões que podem prejudicar outros aspectos importantes do *software*, como coesão e acoplamento, por exemplo. É importante, neste momento de aprendizado, que você pratique a utilização da herança constantemente, mas em sua vida profissional, você aprenderá a discernir problemas que parecem obter vantagem com a herança daqueles que efetivamente a obterão.

RESUMO DO TÓPICO 2

Neste tópico vimos:

- A herança é um dos pilares da programação orientada a objetos.
- Através da herança é possível reutilizar atributos e métodos já implementados em outras classes sem fazer cópia de código-fonte.
- A herança permite que se estabeleça um relacionamento entre duas ou mais classes, onde uma, conhecida como superclasse, é classe que fornece o estado e comportamento a ser herdado.
- As classes inferiores no relacionamento são conhecidas como superclasses ou subclasses.
- Uma classe mãe pode ter quantas classes filhas quiser, entretanto, cada classe filha somente pode estar relacionada a uma classe mãe.
- Em geral, hierarquias de herança devem ser mantidas rasas, com no máximo 2 ou 3 níveis.
- Não existe herança múltipla na linguagem de programação Java.
- O modificador `protected` pode ser utilizado para marcar atributos ou métodos que se deseja passar em relacionamentos de herança para as classes filhas, mas que fiquem privados para outros pacotes.
- A herança é um mecanismo poderoso, entretanto apresenta diversos problemas, entre eles a quebra do encapsulamento.
- Uma alternativa viável para os relacionamentos de herança é a técnica conhecida como composição, onde uma classe possui um atributo do tipo da classe que se deseja reutilizar.
- A composição permite a reutilização de código-fonte de outras classes sem incorrer na violação de encapsulamento geralmente ocasionada pela herança.

AUTOATIVIDADE



- 1 Crie uma hierarquia de herança fictícia, desenhando o diagrama da UML e descreva textualmente quais atributos e métodos da superclasse serão herdados pelas subclasses.
- 2 Crie um projeto no Eclipse chamado de HERANCA e implemente todas as classes que foram definidas no exercício 1, incluindo seus atributos e métodos. Crie ainda uma classe testadora para testar a criação dos objetos e seus relacionamentos.
- 3 Com relação ao relacionamento de herança, assinale V para as alternativas VERDADEIRAS e F para as alternativas FALSAS:
 - () O relacionamento de herança é semelhante ao relacionamento de associação.
 - () A superclasse ou classe filha é a classe que fornece o estado ou comportamento a ser herdado pelas subclasses.
 - () Para se definir o relacionamento de herança, basta fazer a pergunta: "é um tipo de?"
 - () Uma subclasse pode estar relacionada a uma e somente uma superclasse.
 - () A Herança é uma prática que pode interferir com o encapsulamento entre as classes.
- 4 Altere o diagrama feito no exercício 1 de forma a eliminar a herança e incluir a composição no relacionamento entre as classes.
- 5 Crie um projeto no Eclipse chamado de COMPOSICAO e implemente todas as classes que foram definidas no exercício 4, incluindo seus atributos e métodos. Crie ainda uma classe testadora para testar a criação dos objetos e seus relacionamentos.
- 6 Descreva detalhadamente de que forma a composição elimina parte dos problemas causados pela herança.



POLIMORFISMO

1 INTRODUÇÃO

Até aqui aprendemos os dois pilares da programação orientada a objetos: o encapsulamento, que permite a construção de componentes de *softwares* independentes e coesos e a herança, que permite a reutilização e extensão destes componentes.

Existe um ditado na área de desenvolvimento de *software* que diz o seguinte: “as três certezas na vida de um desenvolvedor são: a morte, os impostos e a mudança dos requisitos”. Requisitos mudam, aceite, celebre, abrace esse fato como uma certeza da vida do desenvolvedor e, se possível, projete seu *software* já pensando em algumas mudanças que você consegue antecipar.

O polimorfismo existe como uma ferramenta para possibilitar que você escreva seus componentes de *software* de forma mais flexível, fazendo manutenções futuras sem alterar praticamente nada do que já está funcionando. Nossa objetivo é deixar o *software* o mais “a prova de futuro” possível, mas como você verá nas próximas páginas, esta não é uma tarefa fácil.

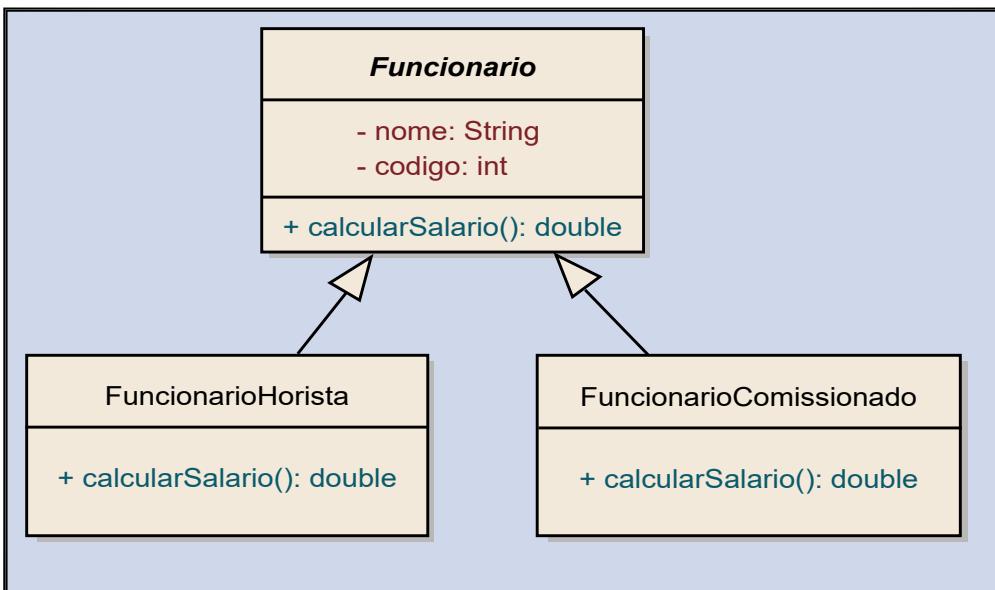
2 POLIMORFISMO

Sem o encapsulamento e a herança, o polimorfismo não seria possível, e sem o polimorfismo, a própria programação orientada a objetos não seria eficiente. Sintes (2002) afirma que é no polimorfismo que a programação orientada a objetos realmente brilha e que seu domínio é absolutamente necessário para que se possa construir programas realmente orientados a objeto.

Polimorfismo significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou método represente um código diferente, selecionado por algum tipo de mecanismo automático. Deste modo, um nome pode assumir muitas formas e como pode representar código diferente, o mesmo nome pode representar muitos comportamentos diferentes (SINTES, 2002).

Vamos partir para um exemplo prático, conforme demonstrado na Figura 44. Neste diagrama simplificado de classe, vemos uma superclasse abstrata chamada de Funcionario e duas subclasses concretas (não abstratas) chamadas de FuncionarioHorista e FuncionarioComissionado. Perceba que as duas subclasses apresentam o método calcularSalario(), aparentemente herdado da superclasse.

FIGURA 44 - POLIMORFISMO



FONTE: Adaptado de: Sintes (2002)



Na notação do diagrama de classes da UML, quando o nome de uma classe ou de um método está em itálico, isso significa que a classe e o método são abstratos. Na Figura 44, podemos observar isso na classe Funcionario e no método calcularSalario().

Lembre-se de que como a classe Funcionario é abstrata, na prática isso significa que ela não pode ser instanciada, ou seja, em nosso sistema não existe um funcionário, mas sim funcionários horistas e comissionados. O que diferencia essas duas classes é a maneira pela qual se faz o cálculo do salário. No caso do funcionário comissionado, o valor do salário seria calculado pela soma do salário fixo com a comissão, enquanto no salário horista, multiplicar-se-ia o valor da hora pela quantidade de horas trabalhadas no mês.

Agora a pergunta é: de que forma isso implementa o polimorfismo? Perceba que, caso existisse uma classe chamada CalculadoraDeSalario, bastaria você passar para ela um objeto de qualquer subtipo de funcionário e pedir que ele mesmo faça o cálculo, ou seja, o mesmo nome de objetos ou métodos representaria duas implementações diferentes, uma para horista e outra para comissionado, selecionada automaticamente em tempo de execução.

Outra vantagem é a possibilidade de se adicionar quantos novos tipos de funcionários fossem necessários, cada um com uma fórmula distinta para calcular o salário, sem alterar NADA do que já está funcionando. Mas como podemos garantir que a calculadora faça o cálculo corretamente? Aí é que está a beleza do polimorfismo... a calculadora não faria nenhum cálculo e sim DELEGARIA esse cálculo para o próprio objeto que está sendo recebido. Digamos que o método que faça a delegação receba um parâmetro do tipo Funcionario. Isso por si só garantiria que somente funcionários fossem recebidos e consequentemente tivessem que fazer o cálculo. Ao tipar o parâmetro deste método é como se dissessemos: "Permitida a entrada somente de funcionários".



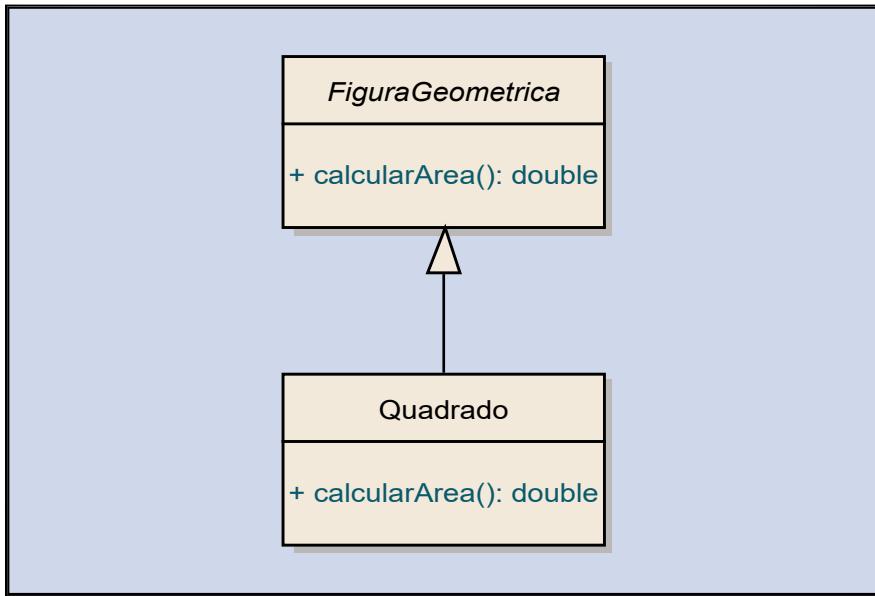
Na linguagem de programação Java, ao declararmos um tipo como uma superclasse de uma relação de herança, esse tipo pode ser instanciado como qualquer um de seus subtipos. O mesmo ocorre em vetores, matrizes ou parâmetros de métodos. Maiores detalhes sobre essa característica do Java serão vistos na Unidade 3.

Vamos exemplificar o polimorfismo de outra forma, com uma hierarquia de herança já utilizada no tópico anterior. Na Figura 45, podemos ver que a classe abstrata FiguraGeometrica tem uma filha, com uma implementação específica para cálculo de área. Como a subclasse é um quadrado, a fórmula para o cálculo de área é lado x lado.

Digamos que em um momento futuro haja a necessidade de se construir uma classe Triangulo, e que conseguimos perceber a semelhança entre suas características de FiguraGeometrica. 1. Bastaria criarmos a nova classe e, em seu método polimórfico calcularArea(), colocar a fórmula $(base \times altura) / 2$. 2. Perceba que essa manutenção não incorre em alteração nenhuma nas demais classes que já estão em funcionamento, economizando tempo e trabalho.

Esse tipo de manutenção simples ocorre somente por causa das facilidades que o polimorfismo fornece. Quando bem empregado, o polimorfismo praticamente permite "prever o futuro" de determinadas partes de nossa aplicação.

FIGURA 45 - POLIMORFISMO DE FIGURAS GEOMÉTRICAS



FONTE: O autor

Esse tipo de manutenção ocorre somente por causa das facilidades que o polimorfismo fornece. Quando bem empregado, o polimorfismo praticamente permite “prever o futuro” de determinadas partes de nossa aplicação.

Apresentaremos aqui os quatro tipos de polimorfismo definidos por Sintes (2002):

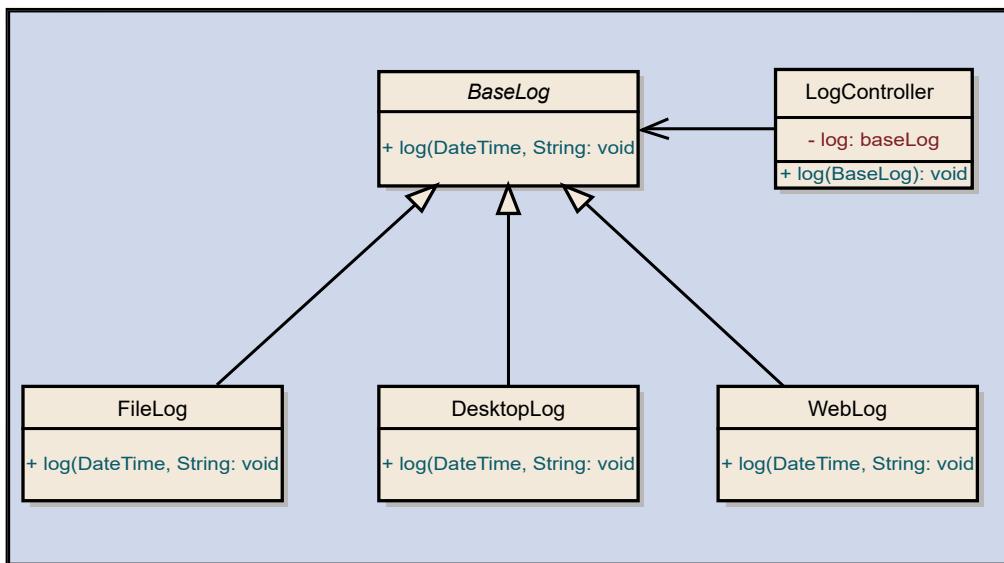
1. Polimorfismo de inclusão.
2. Polimorfismo paramétrico.
3. Sobreposição.
4. Sobrecarga.

2.1 POLIMORFISMO DE INCLUSÃO

O polimorfismo de inclusão, também conhecido como polimorfismo puro, é aquele que demonstramos nos exemplos do funcionário e da figura geométrica. Ele é chamado de polimorfismo de inclusão por que torna mais fácil incluir novos tipos com comportamento semelhante em seu *software*, minimizando o trabalho de manutenção. Este tipo de polimorfismo é o motivo de você não associar automaticamente reutilização de código com herança. Ao invés disso, você deve utilizar herança principalmente para permitir um comportamento polimórfico através de relacionamentos com capacidade de substituição (SINTES, 2002).

Um dos aspectos mais importantes do polimorfismo é a criação de algum mecanismo que faça automaticamente a seleção de qual implementação será escolhida em tempo de execução. Por exemplo, no caso do funcionário deveria haver uma classe com um método que fizesse a delegação do cálculo do salário para um objeto recebido como parâmetro. Esse parâmetro teria que ser do tipo da superclasse da hierarquia, no sentido de permitir que futuras implementações se adaptem automaticamente ao mecanismo. A Figura 46 ilustra um exemplo.

FIGURA 46 - HIERARQUIA DE LOG



FONTE: Adaptado de: Sintes (2002)

Aqui temos uma classe abstrata chamada de `BaseLog`, que representa a base de todos os comportamentos de *log* disponíveis até então. Perceba que existem três implementações específicas, uma para fazer *log* em arquivos, uma para fazer *log* em sistemas *desktop* e outra para fazer *log* em sistemas *web*. A primeira classe gera um arquivo com as informações necessárias, enquanto a segunda exibe na tela uma mensagem e a terceira envia um *e-mail* ao administrador do *website*. Cada uma delas tem uma implementação específica, representando o polimorfismo.

Na classe LogController podemos destacar a existência de duas características básicas para que o mecanismo faça a seleção do tipo de *log* em tempo de execução. A primeira delas é a existência de um atributo do tipo BaseLog e a segunda é o próprio método *log*, que receberá uma instância específica de BaseLog. Dentro deste método, basta chamar o método *log* de qualquer objeto recebido como parâmetro para que a escolha e consequente execução ocorram de forma automática. Caso no futuro quiséssemos gerar um *log* em XML ou mesmo um *log* do tipo sms, enviado para um celular, bastaria criarmos as implementações específicas.

2.2 POLIMORFISMO PARAMÉTRICO

No polimorfismo paramétrico, um método ou tipo de dado pode ser escrito genericamente para que possa suportar valores sem depender de seus tipos específicos, sendo estes definidos somente em tempo de execução. O Quadro 46 ilustra um exemplo de código-fonte escrito de forma a utilizar o polimorfismo genérico. O objetivo desta classe é, em essência, copiar o conteúdo de um vetor para outro vetor, independentemente do tipo dos vetores.



O polimorfismo paramétrico é altamente dependente da linguagem de programação, tornando sua implementação impossível em muitas delas. O Java, desde sua versão 1.5, permite a implementação de polimorfismo paramétrico através do que é conhecido como Generics.

QUADRO 46 – POLIMORFISMO PARAMÉTRICO

```

1
2 public class Copier <T>{
3
4     public void copy(T a[], T b[], int n){
5         for(int i=0; i< n; i++)
6             a[i]=b[i];
7     }
8
9 }
```

FONTE: Adaptado de Zanoni (2014)

Podemos perceber a utilização do tipo <T>, definido como parâmetro para a classe Copier (linha 2) e posteriormente tipando os vetores que serão copiados (linhas 4). O quadro 47 ilustra de que forma faríamos uso dos *generics* para o polimorfismo paramétrico, onde instanciamos dois objetos distintos, cada um recebendo tipos diferentes como parâmetro para o método copy.

QUADRO 47 - TESTE DA CLASSE PARAMÉTRICA

```

1
2 public class Testador {
3
4     public static void main(String[] args) {
5
6         Double f1[] = new Double[50];
7         Double f2[] = new Double[50];
8         Copier<Double> cpDouble = new Copier<>();
9         cpDouble.copy(f1, f2, 10);
10
11        String s1[] = new String[20];
12        String s2[] = new String[40];
13        Copier<String> cpString = new Copier<>();
14        cpString.copy(s1, s2, 15);
15
16    }
17
18 }
```

FONTE: O autor

Perceba que com essa implementação, a classe Copier fica completamente polimórfica, dependendo unicamente do tipo de parâmetro definido para o método copy. Um exemplo bastante comum de utilização de polimorfismo paramétrico é a criação de classes para trabalhar com mapeamento objeto relacional, onde se define uma classe base tipada por <T> e substituída por implementações específicas de entidades a serem persistidas no banco de dados.



Mapeamento objeto relacional é um recurso para conseguir compatibilizar sistemas programados de forma orientada a objetos com bancos de dados relacionais. Abordaremos esse tema quando falarmos de acesso a bancos de dados com Java, na Unidade 3.

2.3 SOBREPOSIÇÃO

Já vimos a utilização de sobreposição nos exemplos anteriores de funcionário e figura geométrica, onde cada subclasse possuía uma implementação particular de um método, de forma a sobrescrever o comportamento já definido na classe base. Aqui é que perceberemos a verdadeira função de uma classe abstrata.

Aprendemos que a classe abstrata não pode ser instanciada, servindo basicamente como um aglutinador de comportamentos e atributos comuns a subclasses. O detalhe é que a linguagem de programação Java possui um mecanismo que obriga o desenvolvedor a fornecer uma nova implementação que sobreponha a implementação original.

O Quadro 48 mostra a classe Funcionario, reescrita agora com um método abstrato para calcularSalario (linha 10). Um método abstrato é um método que, quando escrito em uma superclasse, obriga TODAS as subclasses a fornecer sua própria implementação particular. Dessa forma, em nosso exemplo, o funcionário horista precisa implementar de alguma forma o código para realizar o cálculo do salário, assim como toda e qualquer classe que herdar de funcionário.

QUADRO 48 - CLASSE ABSTRATA

```

1
2 public abstract class Funcionario {
3
4     private String nome;
5
6     public Funcionario(String nome) {
7         this.nome = nome;
8     }
9
10    public abstract double calcularSalario();
11
12 }
```

FONTE: O autor

Alguns detalhes devem ser observados sobre os métodos abstratos:

- Um método abstrato não tem corpo, somente sua assinatura (linha10).
- Métodos abstratos somente podem ser definidos em classes abstratas.
- Uma classe abstrata pode ter quantos métodos concretos (não abstratos) e abstratos forem necessários.
- Nas subclasses, a marcação de que determinado método está sendo sobreescrito ou sobreposto ocorre através de uma Annotation.

- Caso haja uma hierarquia de herança onde uma classe abstrata herde de outra classe abstrata, não há a necessidade de se fornecer uma implementação para os métodos abstratos. A necessidade ocorre somente na primeira classe concreta da hierarquia.

QUADRO 49 - ERRO DE COMPILAÇÃO NA SUBCLASSE

```

1
2 public class FuncionarioHorista extends Funcionario{
3
4     private d
5     private d
6
7     public FuncionarioHorista(String nome, double valorHora,
8         double horasTrabalhadas) {
9             super(nome);
10            this.valorHora = valorHora;
11            this.horasTrabalhadas = horasTrabalhadas;
12        }
13    }
14
15
16 }

```

FONTE: O autor

Caso a implementação não seja feita, ocorrerá um erro de compilação, conforme demonstrado no Quadro 49, linha 2. Ao fornecermos a implementação para o método (Quadro 30), o erro de compilação desaparece e a marcação @Override (linha 13) é adicionada para o método, indicando que estamos sobrescrevendo ou sobrepondo uma implementação da superclasse.

QUADRO 50 - IMPLEMENTAÇÃO DO MÉTODO ABSTRATO

```

1 public class FuncionarioHorista extends Funcionario {
2
3     private double valorHora;
4     private double horasTrabalhadas;
5
6     public FuncionarioHorista(String nome, double val
7         double horasTrabalhadas) {
8             super(nome);
9             this.valorHora = valorHora;
10            this.horasTrabalhadas = horasTrabalhadas;
11        }
12
13    @Override
14    public double calcularSalario() {
15        return valorHora * horasTrabalhadas;
16    }
17
18 }

```

FONTE: O autor

2.4 SOBRECARGA

Conforme Sintes (2002), a sobrecarga permite que você utilize o mesmo nome de método para muitos métodos diferentes, cada um com um número e tipos de parâmetros distintos. Uma implementação bastante comum da sobrecarga ocorre nos construtores de classes conhecidos como telescópicos. Imagine uma classe Email, onde um construtor mínimo exige o destinatário, o remetente, o assunto e o texto do e-mail. Para facilitar a vida do desenvolvedor, existe ainda outro construtor que exige o destinatário, o remetente, o assunto, o texto do *e-mail* e um arquivo para ser anexado. Neste exemplo fizemos a sobrecarga dos construtores da classe, afinal o nome do método é o mesmo, embora seus parâmetros sejam diferentes.

A sobrecarga é considerada um tipo de polimorfismo, pois obtemos comportamento distinto de um mesmo método. A sobrecarga é útil quando um método não é definido por seus argumentos e sim um conceito independente dos parâmetros. Para exemplificar essa situação, observe o código existente no Quadro 51.

QUADRO 51 – SOBRECARGA DE MÉTODOS

```

1 public class Maximo {
2
3     public static int max(int a, int b) {
4         if (a > b)
5             return a;
6         else if (b > a)
7             return b;
8         return 0;
9     }
10
11    public static double max(double a, double b, double c) {
12        return 0;
13    }
14
15 }
```

FONTE: O autor

Nessa classe podemos perceber que existem dois métodos com exatamente o mesmo nome, diferenciando-se somente pelo número e tipo dos parâmetros. Observe que não implementamos nenhum código para retornar o maior valor no segundo método max, pois o objetivo do exemplo é simplesmente demonstrar a sobrecarga. O primeiro método recebe dois parâmetros do tipo int, enquanto o segundo recebe três parâmetros do tipo double. Perceba que, para que a sobrecarga aconteça, basta que se diferencie o número de parâmetros ou o tipo de parâmetros, não havendo necessidade de fazer os dois. Na prática, isso quer dizer que um método max que recebesse três int como parâmetro ou dois double, já estaria fazendo polimorfismo de sobrecarga.



O modificador static que vemos na assinatura dos métodos da classe Maximo, significa que podemos chamá-los sem criar uma instância da classe antes. Maiores detalhes sobre esse modificador serão vistos na Unidade 3 deste caderno.

2.5 INTERFACES

As interfaces são estruturas utilizadas na linguagem de programação Java para fornecer uma abstração de implementação sem entretanto fornecer a implementação propriamente dita. A ideia é permitir que se adicione comportamento à determinada classe por um mecanismo que seja diferente e mais flexível do que a herança.

Quando estudamos herança aprendemos que para existir um relacionamento de herança deve-se responder sim para a pergunta “é um tipo de”. A partir daí, é possível compreender que a herança diz respeito a algo que você é. Além disso, a herança é única, ou seja, ao herdarmos de uma classe, esgotamos todas as possibilidades de herança e caso seja necessário alterar ou mesmo criar um novo comportamento polimórfico, há a necessidade de se reavaliar toda a hierarquia de herança.

A utilização de interfaces nos livra das restrições que a herança traz, pois para adicionarmos um novo comportamento polimórfico, basta dizer que determinada classe implementa tal interface. Nesse sentido, as interfaces funcionam mais ou menos como as classes abstratas, onde existem métodos abstratos que são obrigatoriamente implementados nas classes concretas que participam da herança. Quando uma classe faz a implementação de uma interface, ela assume o compromisso de fornecer implementação para todos os métodos. Por esse motivo é comum referir-se às interfaces como contratos assinados pelas classes. Outra vantagem é que uma classe pode implementar n interfaces diferentes, simulando uma espécie de “herança múltipla”.

A interface diz respeito ao que uma classe “FAZ” e por este motivo são indicadas para as situações onde se deseja somente comportamento polimórfico em detrimento da reutilização de código.

Na Figura 47, ilustramos um exemplo adaptado de Barth (2003), onde se pode utilizar interfaces. Perceba que o RadioRelogio é um rádio e também é um relógio, mas como não podemos contar com herança múltipla, modificamos a relação para funcionar da seguinte forma: um RadioRelógio faz o que um rádio faz e também faz o que um relógio faz. Com essa simples alteração, podemos deixar o código mais flexível e ainda torná-lo passível de aplicação de polimorfismo através de interfaces.

FIGURA 47 – POLIMORFISMO COM RÁDIO RELÓGIO



FONTE: O autor

Os quadros 52 e 53 demonstram de que forma esse relacionamento seria representado na linguagem de programação Java.

QUADRO 52 - INTERFACES RADIO E RELOGIO

```

1
2 public interface Radio {
3
4     void sintonizarEstacao();
5 }

1
2 public interface Relogio {
3
4     void getHora();
5 }

```

FONTE: O autor

No Quadro 52 estão listados os códigos-fonte das interfaces Radio e Relogio, onde cada uma tem um método. Perceba que os métodos não apresentam corpo e encerram logo depois dos parênteses, diferentemente dos métodos de outras classes. O motivo é que TODOS os métodos existentes em uma interface são implicitamente *public* e *abstract*, da mesma forma que os métodos que criamos nas classes abstratas do tópico anterior. Por serem abstratos, estes métodos exigirão implementação por alguma classe.

QUADRO 53 – CLASSE IMPLEMENTANDO AS INTERFACES

```

1
2 public class RadioRelogio implements Radio, Relogio{
3
4     @Override
5     public void getHora() {
6         System.out.println("São 12 Horas!");
7     }
8
9
10    @Override
11    public void sintonizarEstacao() {
12        System.out.println("Sintonizando 94.2 FM");
13    }
14
15
16 }

```

FONTE: O autor

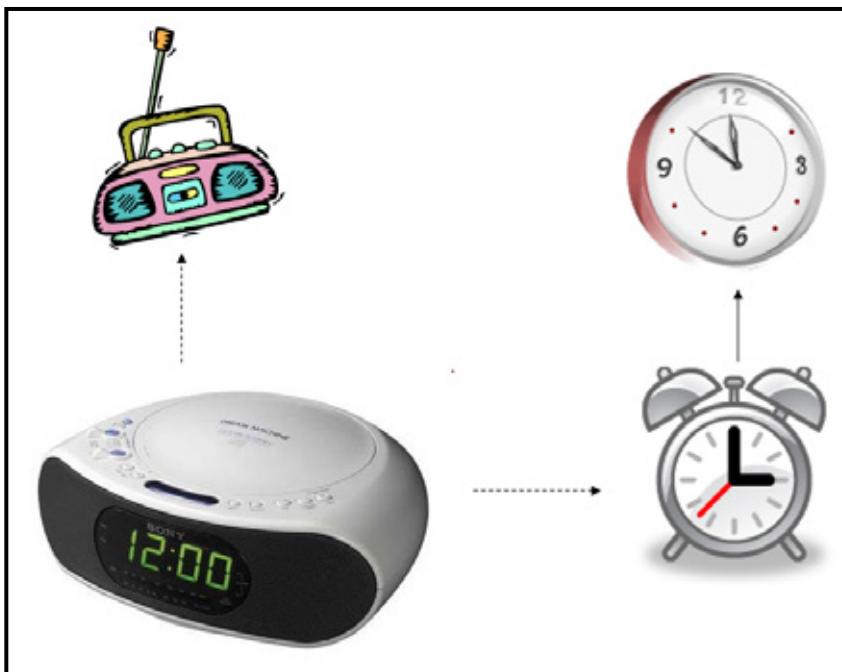
Logo depois da declaração do nome da classe existe a cláusula implements, seguida pelas interfaces Radio e Relogio. Isso significa que assinamos contrato com Radio e com Relogio e somos obrigados a implementar todos os comportamentos existentes nestas interfaces. A implementação ocorre da mesma forma que nas classes abstratas, onde a colocação da anotação @Override (linhas 4 e 10) indica a sobreposição de um método existente em uma classe abstrata ou uma interface.

Vamos agora melhorar o nosso exemplo do RadioRelogio adicionando uma funcionalidade de programação do despertador para determinado horário. Em qual interface deveríamos colocar este método? Considerando que, nem Radio e nem Relogio naturalmente possuem despertador, a opção mais viável é a criação de uma terceira Interface chamada de Despertador. Mas espere, um despertador deve ter alguma função para ver a hora, afinal um despertador é um tipo de relógio. Você consegue pensar em alguma solução para este problema?

Na linguagem de programação Java, uma interface não pode implementar outra interface, entretanto, um relacionamento de herança é perfeitamente permitido. Como definimos anteriormente, um despertador é um tipo de relógio, então definimos a hierarquia de herança conforme a Figura 20.

Utilizando o relacionamento de herança, fazemos com que a interface Despertador herde da interface Relogio. Como Despertador também é uma interface, ele não precisa fornecer a implementação para o método getHora() e o delega para a primeira classe concreta que usá-lo, em nosso caso, a classe RadioRelogio.

FIGURA 48 - RADIORELOGIO VERSÃO 2



FONTE: O autor

Vamos avaliar de que forma essa alteração modificou o código-fonte de nossa implementação através dos quadros 54 e 55. Criamos uma interface Despertador com o método programarAlarme() e a fizemos herdar de Relógio, conforme o relacionamento mostrado na Figura 48. A partir daí, toda classe que implementar Despertador, implicitamente também implementa Relogio.

Essa situação pode ser vista no Quadro 35, onde a classe RadioRelogio implementa os métodos existentes nas três interfaces.

QUADRO 54 - INTERFACE DESPERTADOR

```

1
2 public interface Despertador extends Relogio{
3     void programarAlarme();
4 }
```

FONTE: O autor

QUADRO 55 - RADIORELOGIO 2.0

```

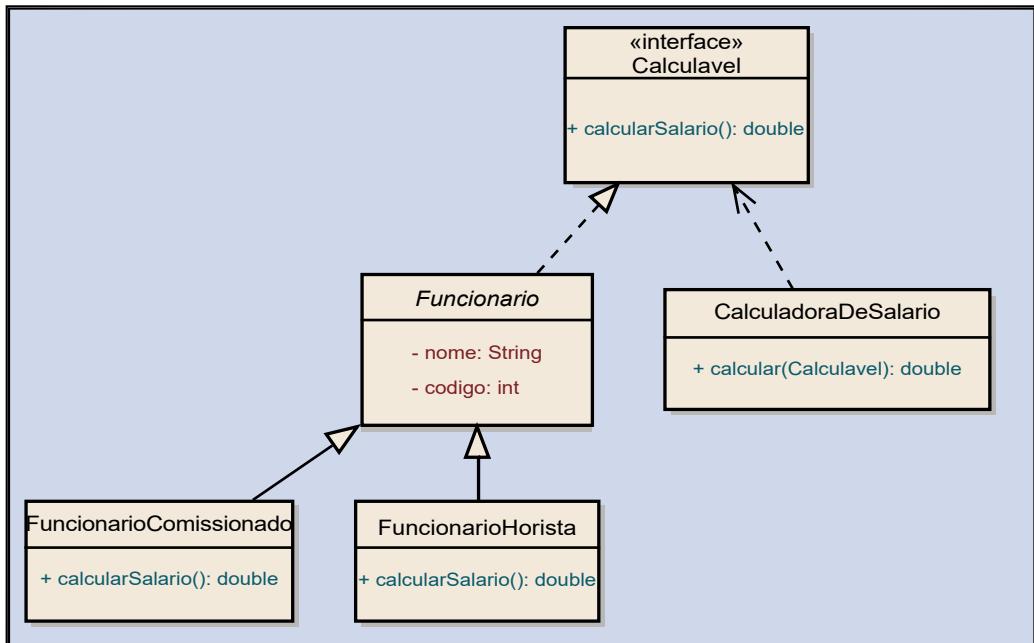
2 public class RadioRelogio implements Radio, Despertador{
3
4     @Override
5     public void getHora() {
6         System.out.println("São 12 Horas!");
7     }
8
9     @Override
10    public void sintonizarEstacao() {
11        System.out.println("Sintonizando 94.2 FM");
12    }
13
14    @Override
15    public void programarAlarme() {
16        System.out.println("Programando alarme para as 7:00 da manha.");
17    }
18 }
```

FONTE: O autor

Mas de que forma a utilização de interfaces auxilia o polimorfismo? Vamos considerar o exemplo do funcionário horista e funcionário comissionado, vistos no início deste tópico. Naquele caso tínhamos uma classe abstrata Funcionario com um método abstrato calcularSalario() e cogitamos a hipótese de criação de uma classe CalculadoraDeSalario que recebesse um objeto do tipo Funcionario e delegasse o cálculo em tempo de execução.

De forma a deixar o exemplo mais flexível, o método `calcularSalario()` poderia ser colocado em uma interface implementada pela classe abstrata `Funcionario`. Caso quiséssemos mais métodos, bastaria adicionar na interface ou mesmo criar outra interface, fazendo com que `Funcionario` a implementasse. Nesta situação específica percebemos o poder das interfaces. A método `calcularSalario()` da classe `CalculadoraDeSalario` poderia receber um objeto do tipo da interface que está implementando, permitindo a delegação da mesma forma. Esta situação é ilustrada na Figura 49.

FIGURA 49 - POLIMORFISMO COM INTERFACES



FONTE: O autor

Perceba que a classe `CalculadoraDeSalario` recebe um parâmetro do tipo `Calculavel`, o que implica que qualquer classe que implementar essa interface pode ser enviada para esse método. O código da classe `CalculadoraDeSalario` pode ser visualizado no Quadro 36. Na linha 5, simplesmente delegamos a chamada ao método `calcularSalario` do objeto que veio como parâmetro. Essa implementação é semelhante àquela demonstrada anteriormente neste tópico, entretanto, não temos as restrições de uma hierarquia de herança. Por exemplo, digamos que exista um consultor contratado pela empresa e que tenha que participar deste processo de cálculo de salário. Você cria uma classe `Consultor`, mas não pode enquadrá-la na herança, afinal ela não representa um tipo de funcionário da empresa e provavelmente não utilizaria todos os atributos herdados. Para resolver este problema, bastaria colocá-lo a implementar a interface `Calculavel` e a partir daí enviá-lo sem problemas para a calculadora de salário. Em resumo, sempre que possível prefira a utilização de interfaces a detrimento às classes abstratas.

QUADRO 56 - CLASSE CALCULADORADESALARIO

```

1
2 public class CalculadoraDeSalario {
3
4     public double calcularSalario(Calculavel calculavel){
5         return calculavel.calcularSalario();
6     }
7
8 }
```

FONTE: O autor

2.6 POLIMORFISMO EFICAZ

Para finalizar este tópico, apresentamos algumas dicas extraídas de Sintes (2002) sobre o polimorfismo eficaz:

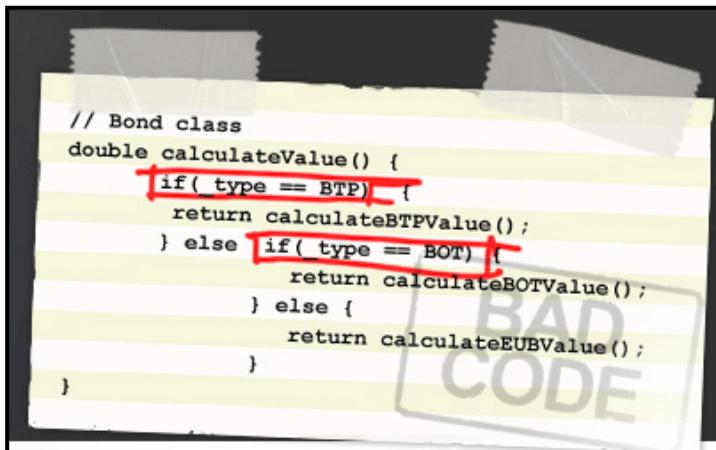
- Planeje com cuidado o encapsulamento e as hierarquias de classe de seus programas.
- Sempre programe para a interface e não para a implementação. Programando para uma interface, você define especificamente quais tipos de objetos podem participar de seu programa, o que faz com que o polimorfismo garanta que esses objetos participem corretamente.
- Pense e programe genericamente. Deixe o polimorfismo se preocupar com os detalhes específicos, dessa forma escrevendo menos código.
- Defina a base do polimorfismo estabelecendo e usando relacionamentos com capacidade de substituição, dessa forma você poderá adicionar novos subtipos e ter certeza de que o código correto será executado.
- Se sua linguagem de programação fornece uma maneira de separar a interface da implementação, favoreça esse mecanismo em detrimento da herança, conforme vimos no item 2.5, separar os dois permite mais flexibilidade no polimorfismo.
- Todas as classes que não estiverem no nível mais baixo de sua hierarquia de herança devem ser abstratas.

LEITURA COMPLEMENTAR

Como não aprender orientação a objetos: o excesso de ifs – Disponível originalmente em: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>. Adaptado livremente pelo autor.

Aglomerados de ifs aparecem com frequência, e chegam até a ter um aspecto engraçado. Em alguns casos podem dar a impressão de que estamos usando orientação a objetos, já que cada cláusula costuma envolver a invocação de um método, dependendo do tipo do objeto. Infelizmente, essa sensação é falsa, e chegou até a gerar o conhecido movimento *anti if campaign* na internet.

FIGURA 1 – ANTI IF CAMPAIGN



FONTE: Disponível em: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

O exemplo a seguir mostra uma sequência de ifs que indicam condições distintas de execução:

QUADRO 1 – SEQUÊNCIA DE IFS

```

35* public double calculaBonus(Funcionario f) {
36     if (f.getCargo().equals("gerente")) {
37         return f.getVendasDoMes() * 0.05 + getSalario() * 0.1;
38     } else if (f.getCargo().equals("diretor")) {
39         return f.getVendasDoMes() * 0.05 + getSalario() * 0.2
40             + (Today.isDecember() ? getSalario() : 0);
41     } else {
42         return f.getVendasDoMes() * 0.01;
43     }
44 }

```

FONTE: Adaptado de: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

Esse tipo de condicional pode ser retrabalhado em objetos (ou funções dependendo da linguagem) que definam o comportamento a ser executado em cada caso. Herança e poliformismo podem (e, na maioria dos casos, devem) ser usados de maneira controlada para **alterar um comportamento**.

QUADRO 2 – HERANÇA E POLIMORFISMO

```

3 class Funcionario {
4
5     protected double vendasDoMes;
6     // outros atributos e metodos aqui
7     public double getBonus() {
8         return vendasDoMes * 0.01;
9     }
10 }
11
12 class Gerente extends Funcionario {
13     // outros atributos e metodos aqui
14     public double getBonus() {
15         return vendasDoMes * 0.05 + getSalario() * 0.1;
16     }
17 }
```

FONTE: Adaptado de: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

Nesse caso, os responsáveis pelo comportamento são encontrados de acordo com uma condição: o tipo que o objeto representa, sendo que cada subtipo pode redefinir o comportamento do método. Não haverá necessidade de ifs, já que essa descoberta de qual método executar é feita pela máquina virtual: `funcionario.getBonus()`.

Esses ifs não costumam aparecer sozinhos. Assim como o comportamento do bônus, surgem em breve outros comportamentos com os mesmos ifs em outras partes do sistema.

QUADRO 3 - REPETIÇÃO DE IFS

```

12*   public boolean liberaVerba(Funcionario f, Produto produto) {
13       if (f.getCargo().equals("gerente")) {
14           return produto.getValor() < 5000
15               || produto.getTipo().equals(Tipo.URGENTE);
16       } else if (f.getCargo().equals("diretor")) {
17           return (int) produto.getValor() < 10000;
18       } else {
19           return produto.getValor() < 1000 ||
20           produto.getTipo().equals(Tipo.USO_DIARIO);
21       }
22   }
```

FONTE: Adaptado de: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

Esse tipo de código fica então muito instável: uma mudança de comportamento deve ser alterada em diversos lugares distintos e, pior ainda, é muito fácil esquecer um deles ao criar um novo comportamento: alguns dos pontos de múltiplos ifs são esquecidos. Ao escrever o mesmo código pela segunda vez, seguimos a prática sugerida de não nos repetir (*DRY – don't repeat yourself*).

Mas o uso da herança é delicado, e o desenvolvedor deve estar ciente de que ela pode trazer um acoplamento indesejado e suas alternativas. O uso de interfaces se encaixaria aqui com perfeição. Outros tipos de condições podem determinar qual ação deve ser tomada, como por exemplo, o valor de um parâmetro, resultando em uma abordagem que utiliza um mapa. Note como, nesse caso, novamente, nenhum *switch* ou sequências de ifs precisam ser feitos: ifs são substituídos por polimorfismo.

QUADRO 4 – POLIMORFISMO

```

29     private Map<String, Aplicador> taxas = new HashMap<String, Aplicador>();
30
31
32     public void processa(String taxa, double juros) {
33         impostosRecolhidos += taxas.get(taxa).aplicaComJuros(juros);
34     }

```

FONTE: Adaptado de: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

De todas as variações, a solução do registro (em geral um Map) e a chave é a abordagem mais simples e capaz de separar responsabilidades distintas. Uma outra variação, que remove a necessidade de registrar os itens no mapa, mas abusa de *reflection*, muitos casos desnecessariamente, é a seguinte:

QUADRO 5 – VARIAÇÃO

```

34     interface AplicadorDeTaxa {
35         double aplicaComJuros(double valor);
36     }
37
38     public void processa(String taxa, double juros)
39             throws Exception {
40         Object instancia = Class.forName("br.com.caelum.taxas." + taxa)
41             .newInstance();
42         AplicadorDeTaxa aplicador = (AplicadorDeTaxa) instancia;
43         impostosRecolhidos += aplicador.aplicaComJuros(juros);
44     }

```

FONTE: Adaptado de: <<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs/>>.

A invocação a `Class.forName("br.com.caelum.taxas." + taxa).newInstance()`; pode ainda ser encapsulada em uma Factory, que em vez de buscar por um nome de classe, consultaria anotações ou um arquivo de configuração.

Esses problemas com o if surgem também em outros paradigmas. Em linguagens funcionais é possível atingir o mesmo resultado usando lambdas, ou ainda em procedurais é possível passar ponteiros de funções com *abstract data types*. Além dos casos em que ifs e condicionais podem ser trocados pelo bom uso de polimorfismo, podemos seguir as boas práticas e evitar condicionais complicados e muito aninhados.

RESUMO DO TÓPICO 3

Neste tópico vimos:

- Polimorfismo significa ter muitas formas, em que um único nome permite representar código diferente.
- Existem basicamente quatro tipos de polimorfismos: polimorfismo de inclusão, polimorfismo paramétrico, polimorfismo de sobreposição e polimorfismo de sobrecarga.
- Embora na comunidade acadêmica exista um certo desacordo entre os tipos de polimorfismo, sabendo os quatro que mostramos nesta unidade, você já terá um bom embasamento em polimorfismo.
- O polimorfismo de inclusão permite que um objeto expresse muitos comportamentos diferentes em tempo de execução.
- O polimorfismo paramétrico permitirá que você defina um objeto ou método que operará com vários tipos diferentes de parâmetros.
- A sobreposição, também conhecida como sobrescrição, permite que você sobreponha um método e deixe para o polimorfismo a escolha de qual método executará.
- Na linguagem de programação Java, a annotation @Override indica que um método está sendo sobreposto ou sobrescrito.
- Uma classe abstrata não pode ser instanciada, entretanto, pode ter tanto métodos concretos quanto abstratos.
- Um método abstrato é um método que EXIGE uma implementação por parte da primeira classe concreta da hierarquia de herança.
- Na prática isso significa que, uma superclasse abstrata A, que contenha um método abstrato a() forçará qualquer um de seus filhos concretos a fornecerem uma implementação para a().
- As interfaces são estruturas que definem uma espécie de contrato que as classes assinam. Ao implementar uma interface, uma classe concreta (não abstrata) tem obrigação de fornecer implementação para TODOS os métodos daquela interface.
- Em uma interface, somente podemos ter atributos do tipo static.

- Todos os métodos contidos em uma interface são implicitamente public abstract.
- Uma interface pode herdar de outra interface.
- Sempre que possível deve-se preferir a utilização de interfaces, ao invés de classes abstratas, pois estas garantem uma maior flexibilidade ao código.
- Enquanto a herança diz respeito ao que você É, a utilização de interfaces diz respeito ao que você FAZ.
- Uma classe pode implementar quantas interfaces forem necessárias.
- A sobreposição permite que você declare um método com o mesmo nome várias vezes, alterando simplesmente o número ou o tipo dos parâmetros. O polimorfismo escolhe qual dos métodos chamar com base nos parâmetros.
- O objetivo principal do polimorfismo é permitir que você escreva código-fonte “a prova de futuro”, em que as alterações a serem feitas se tornam mais fáceis de serem incluídas.

AUTOATIVIDADE



1 Avalie as afirmações a seguir, colocando V para as afirmações Verdadeiras e F para as Falsas:

- () Polimorfismo significa colocar diversos comportamentos sob um mesmo nome, escolhidos em tempo de compilação.
- () Para que o polimorfismo aconteça, existe a obrigatoriedade da herança.
- () Uma classe abstrata é uma classe que obrigatoriamente possui métodos abstratos.
- () Uma interface não pode conter métodos concretos.
- () Quando uma classe implementa uma interface, a partir deste momento podemos tipá-la através desta interface.

Agora assinale a alternativa que contém a sequência CORRETA:

- a) () F – F – F – V – V.
- b) () F – V – V – F – F.
- c) () V – F – V – F – F.
- d) () F – F – V – F – V.

2 Utilizando as classes criadas no exemplo da Figura Geométrica visto neste tópico, refatore o código de modo a remover o método abstrato calcularArea() da classe FiguraGeometrica e o coloque dentro de uma interface. Deve ser criada uma classe testadora que permita o teste de todas as figuras.

3 Qual é a principal utilidade das interfaces quando em comparação com as classes abstratas? De exemplos:

4 Crie um novo projeto no eclipse e desenvolva um exemplo contendo classes para demonstrar de que forma o polimorfismo pode auxiliar na manutenção do código-fonte de uma aplicação. O exemplo deve ser diferente daqueles mostrados nesta unidade.

TÓPICOS AVANÇADOS DA LINGUAGEM E BOAS PRÁTICAS

OBJETIVOS DE APRENDIZAGEM

Ao final desta unidade, você será capaz de:

- utilizar conceitos avançados da linguagem de programação Java para implementar os princípios fundamentais da Programação Orientada a Objetos;
- compreender de que forma as coleções podem auxiliar seus programas a manterem a coesão e o encapsulamento;
- conhecer a API de acesso a banco de dados JDBC;
- conhecer e utilizar os conceitos de mapeamento objeto relacional em seu código;
- tornar seu código fonte mais claro, flexível e confiável através da utilização de boas práticas.

PLANO DE ESTUDOS

Esta unidade de ensino está dividida em três tópicos, sendo que no final de cada um deles, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – COLEÇÕES E SORT

TÓPICO 2 – STATIC E TRATAMENTO DE EXCEÇÕES

TÓPICO 3 – JDBC E DAO



COLEÇÕES E SORT

1 INTRODUÇÃO

Nas duas unidades anteriores abordamos os principais pontos que caracterizam o paradigma da programação orientada a objetos. Você aprendeu qual é a diferença entre classes e objetos, qual é o motivo da coesão e acoplamento estarem diretamente relacionados ao encapsulamento e ainda percebeu que a herança pode contribuir com o polimorfismo, embora não seja a única maneira de aplicá-lo.

Todos os tópicos conceituais foram acompanhados de exemplos escritos em código fonte, afinal, na prática, a implementação é que demonstra a aplicabilidade e utilidade da programação orientada a objetos.

“A verdade está no código”, como diz Eric Evans em seu livro Domain Driven Design (EVANS, 2003) e, por este motivo, um profundo conhecimento sobre a linguagem de programação que se está utilizando é necessário por parte do programador. Conhecer adequadamente a ferramenta de trabalho é essencial para que se possa resolver cada problema com a solução adequada. A linguagem de programação Java, especialmente, possui uma API muito rica, que já ataca boa parte dos problemas que os programadores enfrentam no dia a dia, permitindo que estes se concentrem em resolver os problemas do domínio onde atuam. Entretanto, para que se faça uso desta API, é necessário conhecê-la.

Este é nosso objetivo nesta unidade: fornecer um entendimento mais aprofundado de algumas características da Linguagem de Programação Java que a tornam tão atraente para o desenvolvimento de aplicações. Da mesma forma que nas unidades anteriores, sugerimos fortemente que você estude este caderno com o Eclipse aberto e digitando todos (eu disse TODOS) os códigos fontes dos exemplos, mesmo que o conceito que esteja sendo demonstrado tenha sido entendido por você. Escrever código fonte dará a você fluência na linguagem, além de possibilitar a convivência com os erros e mensagens do compilador. Isso acabará por tornar você um programador melhor.

Salientamos que nem todas as ferramentas da linguagem de programação Java que demonstraremos aqui estão relacionadas com a Programação Orientada a Objetos. A finalidade de algumas destas ferramentas é simplesmente permitir que você consiga escrever seus programas de forma menos complexa.

Você já venceu duas unidades e com certeza já adquiriu bastante conhecimento sobre a plataforma Java e sobre a programação orientada a objetos. Não desanime, pois nesta última unidade aprofundaremos ainda mais a linguagem. Conto com você para vencermos mais esta etapa. Bons estudos!

2 COLEÇÕES E SORT

Conforme Caelum (2014), a manipulação de arrays é bastante trabalhosa e não permite redimensionamento, o que exige o conhecimento do número total de elementos ANTES de efetivamente criá-la. Outra dificuldade ao se trabalhar com arrays é que não se pode buscar diretamente por um elemento cujo índice não se conheça.

Todas estas funcionalidades estão disponíveis no conjunto de classes e interfaces conhecido como Collections Framework, disponível desde o Java 1.2.

Mas efetivamente, o que é uma coleção? Embora nem todas as coleções obedeçam a este tipo de estrutura, você pode pensar em uma coleção como uma espécie de vetor dinâmico, onde o tamanho do vetor não precisa ser conhecido no momento de sua criação. Isso significa que o mesmo que você consegue fazer em um vetor, você consegue fazer em uma coleção, tendo a vantagem da existência de diversos métodos utilitários.

2.1 ARRAYLIST

Vamos começar com o tipo de coleção que obedece praticamente à mesma estrutura de um vetor, conhecida como Lista. No Quadro 57 demonstramos a criação e utilização de uma lista na linguagem de programação Java. Na unidade anterior demonstramos a utilização de Generics para o polimorfismo paramétrico. Aqui o Generics é utilizado para tipar a coleção, ou seja, permitir que somente um tipo de dados seja colocado na coleção, neste caso, o tipo String.



A utilização de generics em Java apresenta inúmeras finalidades além daquelas demonstradas neste Caderno de Estudos. Para maiores detalhes sobre a utilização de Generics em Java, acesse: <<http://tutorials.jenkov.com/java-generics/index.html>>.

QUADRO 57 – CRIAÇÃO DE UMA LISTA

```

6 public class Testador {
7
8     public static void main(String[] args) {
9
10         List<String> lista = new ArrayList<>();
11
12         lista.add("antonio");
13         lista.add("juliana");
14         lista.add("cesar");
15         lista.add("carlos");
16
17         lista.remove("juliana");
18     }
19     for(String s:lista)
20         System.out.println(s);
21 }
22 }
```

FONTE: O autor

Na linha 10 vemos a declaração e instanciação da lista. No Java, o tipo específico é o ArrayList. Inicialmente o ArrayList reserva 10 posições de memória para armazenar os objetos e quando necessário aloca mais 10 de cada vez. Este tipo de coleção pode ser ordenado e permite duplicatas, ou seja, se você quiser adicionar mais de uma vez o mesmo objeto, não há problemas. Nas linhas 12 a 15 adicionamos as Strings à lista, através do método add. Na linha 17 fazemos a remoção do objeto da lista.

Nas linhas 19 e 20 utilizamos a instrução conhecida como enhanced for, que se lê da seguinte forma: para cada String s da lista, faça algo. No nosso caso, repetiremos três vezes a impressão de uma String no console, de acordo com o número de objetos na lista (Figura 50).

FIGURA 50 – IMPRESSÃO DA COLEÇÃO

```
<terminated> Testador (11) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (03/11/2014 13:46:46)
antonio
cesar
carlos
```

FONTE: O autor

Um detalhe importante a ser salientado, é que a coleção pode conter **QUALQUER** tipo de dados. Para exemplificar, vamos criar uma classe chamada de Pessoa, com os atributos CPF e nome (Quadro 58). Os getters e setters foram omitidos por motivos de simplificação.

No Quadro 59, utilizamos uma classe testadora para instanciar quatro objetos da classe Pessoa e em seguida fazer o mesmo procedimento que fizemos com as Strings no Quadro 57.

QUADRO 58 – IMPLEMENTAÇÃO DA CLASSE PESSOA

```
3 class Pessoa {
4
5     private String cpf;
6     private String nome;
7
8     public Pessoa(String cpf, String nome) {
9         super();
10        this.cpf = cpf;
11        this.nome = nome;
12    }
13
14*    public String getNome() {..}
15
16*    public void setNome(String nome) {..}
17
18*    public String getCpf() {..}
19
20
21
22*    public String toString() {..}
23
24
25
26 }
```

FONTE: O autor

QUADRO 59 – INSERÇÃO DAS PESSOAS NA LISTA

```

5 class Testador {
6
7     public static void main(String[] args) {
8
9         List<Pessoa> lista = new ArrayList<>();
10
11        Pessoa p1 = new Pessoa("8373635","Juliana");
12        Pessoa p2 = new Pessoa("8555535","Carlos");
13        Pessoa p3 = new Pessoa("2223345","Karol");
14        Pessoa p4 = new Pessoa("5578999","Cesar");
15
16        lista.add(p1); lista.add(p2);lista.add(p3);lista.add(p4);
17
18        lista.remove(p1);
19
20        for(Pessoa p:lista)
21            System.out.println(p.getNome());
22    }
23 }
```

FONTE: O autor

Perceba que estruturalmente praticamente não houve mudança entre os exemplos demonstrados nos quadros 58 e 59, entretanto, algumas adequações tiveram que ser feitas devido ao fato do objeto a ser armazenado pela lista ser diferente. Na linha 9 do Quadro 59, o tipo de dados dentro do operador diamante (`<>`) agora é Pessoa, ao invés de String. Isso indica que nossa lista somente poderá conter objetos do tipo Pessoa. Na linha 20 alteramos o tipo de dado a ser listado no enhanced for, onde para cada Pessoa existente na lista, imprimiremos seu nome.



O mesmo princípio utilizado para vetores e parâmetros de métodos também é válido para o operador diamante. Qualquer subtipo de Pessoa também poderia estar na lista.

Um ArrayList comporta-se exatamente como um vetor, então é possível referenciar objetos através de índices e mesmo percorrer a lista através de um for tradicional, indexando uma variável do tipo inteira `i` (Quadro 60). A ordem de armazenamento dos elementos dentro de uma lista é exatamente a mesma da inserção. A diferença básica é que no for tradicional, precisamos pegar um elemento da lista, atribuí-lo a um objeto e aí sim fazer a impressão do nome daquele objeto.

QUADRO 60 – INDEXAÇÃO DE UM ARRAYLIST

```

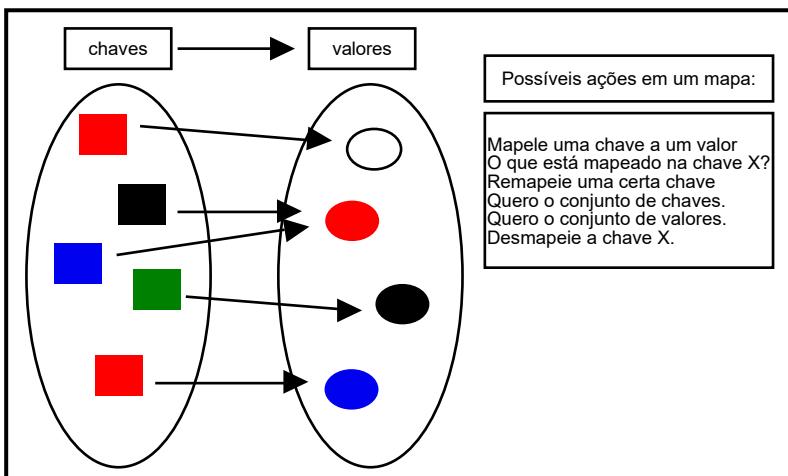
6 class Testador {
7
8 public static void main(String[] args) {
9     List<Pessoa> lista = new ArrayList<>();
10
11     Pessoa p1 = new Pessoa("8373635","Juliana");
12     Pessoa p2 = new Pessoa("8555535","Carlos");
13     Pessoa p3 = new Pessoa("2223345","Karol");
14     Pessoa p4 = new Pessoa("5578999","Cesar");
15
16     lista.add(p1); lista.add(p2);lista.add(p3);lista.add(p4);
17     lista.remove(p1);
18
19     for(int i=0; i<lista.size();i++){
20         Pessoa p = lista.get(i);
21         System.out.println(p.getNome());
22     }
23 }
24 }
```

FONTE: O autor

2.2 HASHMAP

Uma coleção do tipo Mapa tem um comportamento bastante peculiar. Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa, que é composto por um conjunto de associações entre um objeto-chave a um objeto-valor (Figura 51). É equivalente ao conceito de dicionário, usado em várias linguagens (CAELUM, 2014).

FIGURA 51 – COLEÇÕES DO TIPO MAPA



FONTE: Caelum (2014)

Vamos exemplificar a utilização de mapas através do mesmo objeto Pessoa utilizado anteriormente. No Quadro 61, podemos observar a criação e inserção de 4 objetos do tipo pessoa no Map, sendo indexadas pelo CPF. Nas linhas 17 e 18 utilizamos o método put, colocando como parâmetro a String e o objeto Pessoa propriamente dito. Nas linhas 20 a 22 fazemos a iteração no mapa, imprimindo sempre o índice (key) e depois um atributo do objeto (value). Os generics determinam que os Keys serão sempre Strings e os value serão sempre objetos do tipo Pessoa. É importante salientar que o HashMap aceita qualquer tipo de dados como key e qualquer tipo de dados como value.

Ao executarmos esse código, perceberíamos que a ordem de armazenamento dos elementos não necessariamente é a mesma de inserção, ou seja, ao contrário das listas, a ordem não é importante para o HashMap. Caso tentássemos fazer a inserção de um mesmo objeto pessoa duas vezes, não haveria problema desde que seu índice (key) fosse diferente. Ao inserirmos dois objetos distintos com uma mesma chave, a primeira ocorrência do objeto é sobreescrita pela segunda. O HashMap permite duplicatas desde que sejam referenciadas por índices distintos. Em contrapartida, é perfeitamente permitido que um objeto seja referenciado por mais de uma chave.

QUADRO 61 – EXEMPLO DE MAP

```

7 class Testador {
8     public static void main(String[] args) {
9
10         Pessoa p1 = new Pessoa("8373635", "Juliana");
11         Pessoa p2 = new Pessoa("8223635", "Karol");
12         Pessoa p3 = new Pessoa("2345677", "Cecilia");
13         Pessoa p4 = new Pessoa("2349999", "Anne");
14
15         Map<String, Pessoa> mapa = new HashMap<>();
16
17         mapa.put(p1.getCpf(), p1); mapa.put(p2.getCpf(), p2);
18         mapa.put(p3.getCpf(), p3); mapa.put(p4.getCpf(), p4);
19
20         for (Map.Entry<String, Pessoa> elemento: mapa.entrySet()) {
21             System.out.println(elemento.getKey());
22             System.out.println(elemento.getValue().getNome());
23         }
24     }
25 }
```

FONTE: O autor

Existem métodos utilitários para a busca de elementos com base na chave, afinal este é o objetivo da utilização de um par chave/valor. No Quadro 6 repetimos a inserção feita no Quadro 61, mas ao invés de listar os elementos, fazemos a busca por um CPF que, caso encontrado, retornará o objeto Pessoa relacionado ao mesmo (linhas 22 e 23). Perceba que ao retornarmos uma pessoa, podemos chamar o método getNome() sem maiores problemas e obter a String “Cecilia” como resultado na saída padrão.

QUADRO 62 – BUSCA DE UM ELEMENTO PELA CHAVE

```

7 class Testador {
8     public static void main(String[] args) {
9
10         Pessoa p1 = new Pessoa("8373635", "Juliana");
11        Pessoa p2 = new Pessoa("8223635", "Karol");
12        Pessoa p3 = new Pessoa("2345677", "Cecilia");
13        Pessoa p4 = new Pessoa("2349999", "Anne");
14
15        Map<String, Pessoa> mapa = new HashMap<>();
16
17        mapa.put(p1.getCpf(), p1); mapa.put(p2.getCpf(), p2);
18        mapa.put(p3.getCpf(), p3); mapa.put(p4.getCpf(), p4);
19
20        String cpf = "2345677";
21
22        if(mapa.containsKey(cpf))
23            System.out.println(mapa.get(cpf).getNome());
24    }
25 }
```

FONTE: O autor

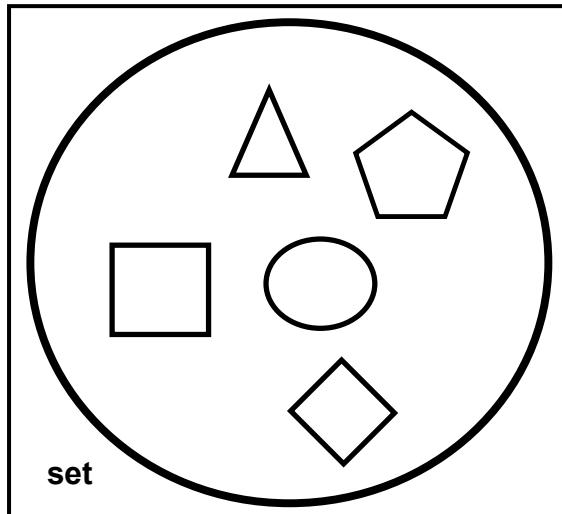
2.3 HASHSET

Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados. Outra característica fundamental é que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto (CAELUM, 2014).

Imaginando que o círculo dentro da Figura 51 representa um conjunto de figuras geométricas, seguem algumas operações possíveis em um conjunto:

- O pentágono está no conjunto?
- Remova o círculo.
- Adicione um paralelogramo.
- Limpe o conjunto.

FIGURA 53 – CONJUNTO



FONTE: Adaptado de Celum (2014)

Perceba que, ao criarmos uma coleção do tipo conjunto, não é possível a adição de elementos duplicados e sua ordem não é conhecida. Então qual seria a vantagem de utilizar um Set ao invés de uma lista? A verificação da existência de um elemento em um conjunto é muito mais rápida do que em uma lista. Sempre que a ordem dos elementos não for importante, considere a utilização de um conjunto.

O código fonte do Quadro 63 ilustra a criação e utilização de um conjunto através da implementação HashSet. Perceba que a declaração do Set (linha 17) é muito semelhante a do ArrayList, inclusive pela tipagem da coleção através dos generics. Neste exemplo, fazemos uso novamente do enhanced for para listar os elementos do conjunto (linha 20).

QUADRO 63 – IMPLEMENTAÇÃO DO HASHSET

```

8 class Testador {
9
10 public static void main(String[] args) {
11
12     String p1 = "Juliana";
13     String p2 = "Karol";
14     String p3 = "Scarlett";
15     String p4 = "Juliana";
16
17     Set<String> conjunto = new HashSet<>();
18     conjunto.add(p1);conjunto.add(p2);
19     conjunto.add(p3);conjunto.add(p4);
20     for (String string : conjunto) {
21         System.out.println(string);
22     }
23 }
24 }
```

FONTE: O autor

Mas agora, qual será a saída do código acima? A Figura 52 mostra o que aconteceu com a execução do Testador. Note que a primeira String a ser impressa é Juliana, enquanto a segunda é Scarlett e a terceira é Karol, ou seja, ele não obedeceu à ordem de inserção das Strings e, além disso, não permitiu a colocação da String p4 no conjunto, afinal as repetições não são permitidas.

FIGURA 52 – LISTAGEM DOS ELEMENTOS DO CONJUNTO

```
<terminated> Testador (11) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (05/11/2014 12:32:54)
Juliana
Scarlett
Karol
```

FONTE: O autor

Um exemplo da utilização dos métodos utilitários dos conjuntos pode ser visto no Quadro 64. Na linha 18 fizemos a remoção do objeto Juliana, utilizando o método remove. Na linha 20 perguntamos se a String “Karol” existe no conjunto, através do método contains. Finalmente, na linha 25 limpamos o conjunto, excluindo todos os seus elementos.

QUADRO 64 – ALGUNS MÉTODOS DA INTERFACE SET

```
8 class Testador {
9     public static void main(String[] args) {
10         String p1 = "Juliana";
11         String p2 = "Karol";
12         String p3 = "Scarlett";
13         String p4 = "Juliana";
14
15         Set<String> conjunto = new HashSet<>();
16         conjunto.add(p1);conjunto.add(p2);
17         conjunto.add(p3);conjunto.add(p4);
18         conjunto.remove("Juliana");
19
20         if(conjunto.contains("Karol"))
21             System.out.println("Karol está no conjunto");
22         else
23             System.out.println("Karol não está no conjunto");
24
25         conjunto.clear();
26     }
27 }
```

FONTE: O autor



Existem outros tipos de coleções no Collections Framework que não foram abordados neste caderno. Para um aprofundamento maior no tema, sugerimos a leitura do material disponível em Caelum (2014).

2.4 IGUALDADE ENTRE OBJETOS

Observe o seguinte exemplo de código fonte (Quadro 62), onde utilizamos a classe Pessoa, definida no Quadro 58. Perceba que na linha 14 perguntamos se o objeto p1 é igual ao objeto p2 e mandamos o console imprimir uma mensagem de acordo com a resposta a esta expressão booleana. O que você acha? Qual será a resposta?

QUADRO 65 – EXEMPLO DE IGUALDADE ENTRE OBJETOS

```

6 class Testador {
7
8  public static void main(String[] args) {
9
10
11      Pessoa p1 = new Pessoa("8373635", "Juliana");
12      Pessoa p2 = new Pessoa("8555535", "Carlos");
13
14      if(p1==p2)
15          System.out.println("São iguais");
16      else
17          System.out.println("São diferentes");
18  }
19 }
```

FONTE: O autor

Se você respondeu “são diferentes”, acertou. Agora vejamos um novo exemplo, definido no Quadro 6. Neste exemplo, atribuímos EXATAMENTE os mesmos valores para o CPF e o nome dos objetos p1 e p2. E agora? Qual será a resposta?

QUADRO 66 – IGUALDADE ENTRE OBJETOS, EXEMPLO 2

```

6 class Testador {
7
8 public static void main(String[] args) {
9
10
11     Pessoa p1 = new Pessoa("8373635", "Juliana");
12     Pessoa p2 = new Pessoa("8373635", "Juliana");
13
14     if(p1==p2)
15         System.out.println("São iguais");
16     else
17         System.out.println("São diferentes");
18 }
19 }
```

FONTE: O autor

Todos que responderam “são diferentes”, acertaram. Mas espere aí... no Quadro 61 tudo bem, afinal os valores dos atributos são efetivamente diferentes. Já no exemplo do Quadro 62, colocamos exatamente os mesmos valores para os atributos e ainda assim obtivemos como resposta que os objetos são diferentes. Você consegue imaginar o motivo deste comportamento?

A resposta reside no objetivo do comparador “==” quando utilizado com referências de objetos ao invés de tipos primitivos. Com tipos primitivos (e até mesmo com Strings) o comportamento é o que se espera (Quadro 67). Neste exemplo, o resultado seria “são diferentes”, mas se eu alterar o valor de p2 para 5, imediatamente obteríamos na execução a mensagem “são iguais”.

QUADRO 67 – COMPARAÇÃO ENTRE TIPOS PRIMITIVOS

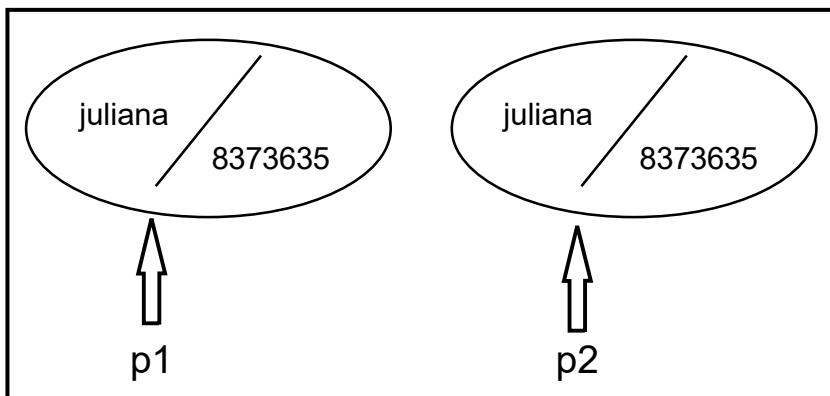
```

6 class Testador {
7
8 public static void main(String[] args) {
9
10
11     int p1 = 5;
12     int p2 = 3;
13
14     if(p1==p2)
15         System.out.println("São iguais");
16     else
17         System.out.println("São diferentes");
18 }
19 }
```

FONTE: O autor

Com referências, a linguagem de programação Java atua de maneira diferente. Quando perguntamos se o objeto `p1==p2`, na verdade estamos perguntando se os dois referenciam o mesmo endereço de memória. Observe a Figura 54 para um melhor entendimento.

FIGURA 54 – REFERÊNCIAS DOS OBJETOS



FONTE: O autor

Na figura os objetos `p1` e `p2`, depois de instanciados, possuem exatamente os mesmos valores para os atributos, entretanto, os locais referenciados pelas variáveis `p1` e `p2` na memória são diferentes, o que resulta na resposta falso para a pergunta `p1==p2`. Agora observe alteração na linha 13 do Quadro 68.

QUADRO 68 – ATRIBUIÇÃO DA MESMA REFERÊNCIA AO OBJETO P2

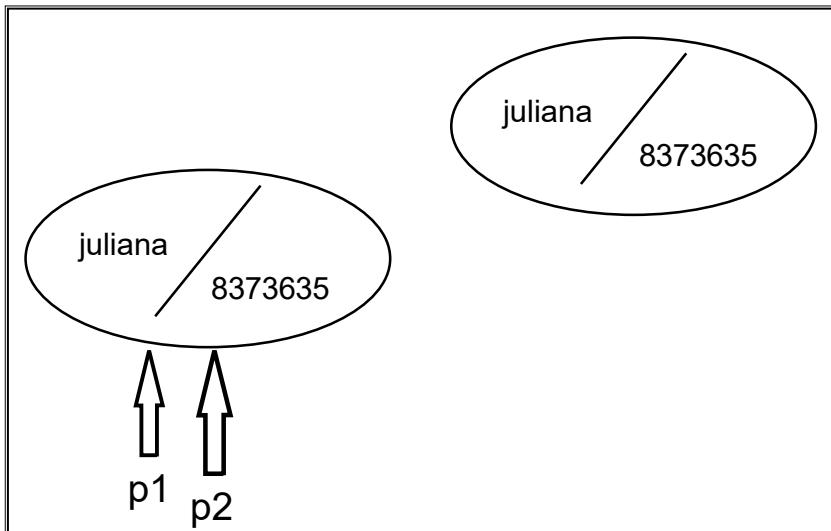
```

6 class Testador {
7
8     public static void main(String[] args) {
9
10        Pessoa p1 = new Pessoa("8373635", "Juliana");
11        Pessoa p2 = new Pessoa("8373635", "Juliana");
12
13        p2 = p1;
14        if (p1 == p2)
15            System.out.println("São iguais");
16        else
17            System.out.println("São diferentes");
18    }
19 }
```

FONTE: O autor

Ao executar o programa, agora obtemos a mensagem “são iguais”. Isso ocorre porque na linha 13 dizemos basicamente o seguinte: “`p2`, a partir de agora você referencia o mesmo endereço de `p1`”. A Figura 6 ilustra o ocorrido.

FIGURA 55 – DIAGRAMA DE CLASSES COM MÉTODOS



FONTE: Cunha, 2009

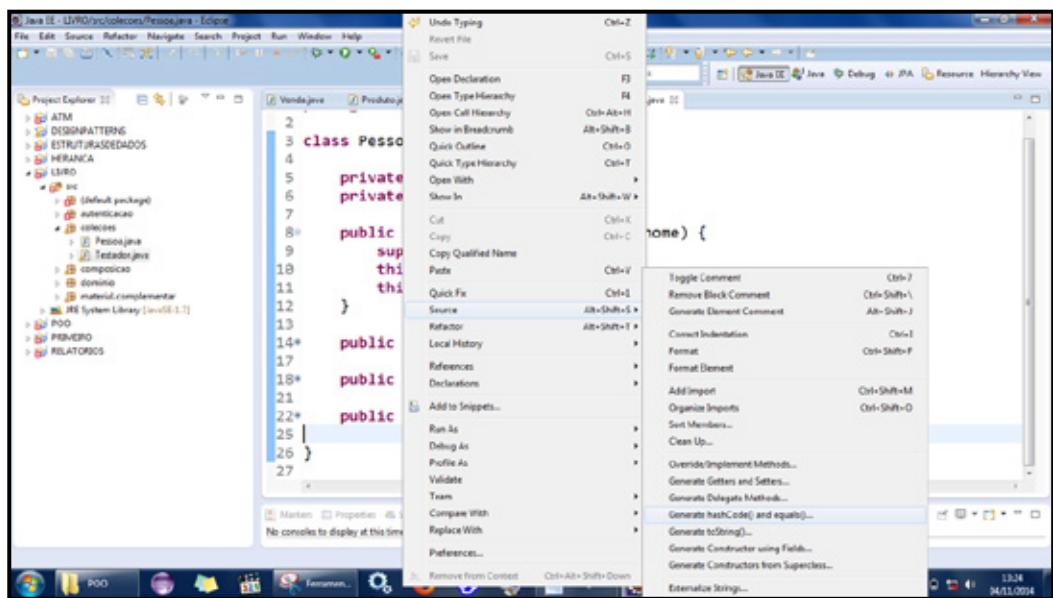
Podemos perceber que as duas variáveis referenciam o mesmo endereço de memória, enquanto o objeto anteriormente referenciado por p2 agora é um candidato ao mecanismo de coleta de lixo (garbage collector) do Java.



O coletor de lixo é um dos mecanismos mais importantes para o bom funcionamento da plataforma Java. Maiores detalhes sobre o Garbage Collector podem ser obtidos no link: <<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>>.

Mas se você quiser efetivamente um mecanismo que permita a comparação entre objetos do mesmo tipo de acordo com algum atributo ou conjunto de atributos? A linguagem de programação Java permite a implementação deste tipo de comportamento através do polimorfismo. A classe Object, que é classe mãe de TODAS as classes do Java, inclusive daquelas que você acabou de criar, possui um método chamado equals. O objetivo deste método é fornecer um mecanismo de comparação entre dois objetos do mesmo tipo de acordo com critérios que podem ser definidos pelo implementador. Por exemplo: nossa classe Pessoa tem dois atributos, o nome e o CPF. Podem existir duas pessoas com exatamente o mesmo nome, mas não com o mesmo CPF. Com base nesta regra, poderíamos dizer que, se duas pessoas possuem o mesmo CPF, é porque são efetivamente a mesma pessoa. Observe na Figura 56 como fazer essa implementação no Eclipse e no Quadro 69 o resultado obtido.

FIGURA 56 – CRIAÇÃO DO MÉTODO EQUALS NO ECLIPSE



FONTE: O autor

Ao clicar com o botão direito do mouse dentro da classe Pessoa, deve-se selecionar as opções Source -> Generate Hashcode() and Equals(). A próxima tela permite que você escolha quais os atributos farão parte do método. No nosso caso, queremos que o único atributo que determine igualdade entre objetos seja o CPF, então o deixamos selecionado.

No Quadro 69, podemos perceber o código gerado pelo Eclipse. Por hora, basta sabermos que este método retornará true se e somente se os dois objetos do tipo pessoa tiverem exatamente o mesmo CPF. Na linha 35 podemos perceber o uso do polimorfismo através da annotation @Override. Já no Quadro 70, fazemos o teste da igualdade utilizando o mesmo exemplo anterior, somente trocando o operador “==” pelo método que acabamos de criar.

QUADRO 69 – MÉTODO EQUALS GERADO PELO ECLIPSE

```

35@override
36 public boolean equals(Object obj) {
37     if (this == obj)
38         return true;
39     if (obj == null)
40         return false;
41     if (getClass() != obj.getClass())
42         return false;
43     Pessoa other = (Pessoa) obj;
44     if (cpf == null) {
45         if (other.cpf != null)
46             return false;
47     } else if (!cpf.equals(other.cpf))
48         return false;
49     return true;
50 }
```

FONTE: O autor

QUADRO 70 – COMPARAÇÃO ENTRE OBJETOS

```

6 class Testador {
7
8     public static void main(String[] args) {
9
10        Pessoa p1 = new Pessoa("8373635", "Juliana");
11        Pessoa p2 = new Pessoa("8373635", "Juliana");
12
13
14        if (p1.equals(p2))
15            System.out.println("São iguais");
16        else
17            System.out.println("São diferentes");
18    }
19 }
```

FONTE: O autor

Perceba que na linha 14 fazemos a pergunta: p1 e p2 são iguais de acordo com o que foi definido no método equals()? Se a resposta for afirmativa, o método retornará true, caso contrário, retornará false. No caso do exemplo, os objetos são iguais perante o método. Esse mecanismo torna bastante fácil a comparação entre quaisquer objetos que você crie, bastando utilizar o polimorfismo através do método equals(). Nos tópicos onde colocamos que determinadas coleções (HashSet) não permite repetições, é exatamente através do método equals() que a coleção determina a igualdade dos objetos.

2.5 HASHCODE

Muitas das coleções da linguagem de programação Java guardam seus objetos dentro de tabelas de espalhamento (hash). Para que este mecanismo funcione, cada objeto é classificado através de seu hashCode e agrupado com outros que possuírem o mesmo valor.

Uma boa prática em Java é implementar o hashCode de forma que, se dois objetos A e B retornarem true através de uma comparação feita com o equals, obrigatoriamente o hashCode de A e B deverão ser iguais. Caelum (2014) afirma que implementar hashCode de forma que resultados distintos retornem para objetos considerados iguais pelo equals resulta em coleções que usam espalhamento (HashMap e HashSet) de forma incorreta. O Quadro 71 traz a implementação do método hashCode da classe Pessoa.

QUADRO 71 – IMPLEMENTAÇÃO DE HASHCODE

```

26  @Override
27  public int hashCode() {
28      final int prime = 31;
29      int result = 1;
30      result = prime * result + ((cpf == null) ? 0 : cpf.hashCode());
31      return result;
32  }
33

```

FONTE: O autor

2.6 ORDENAÇÃO

O collections framework traz diversas coleções onde a ordem é relevante e cabe ao implementador efetivamente determinar a ordenação dos objetos por algum critério específico. No âmbito deste caderno, a única coleção ordenada que abordamos é o ArrayList e por este motivo faremos a ordenação em exemplos com este tipo.

Sempre que falamos em ordenação, há a necessidade de se definir um critério para tal. Por exemplo, em nossa classe Pessoa, poderíamos utilizar uma ordenação alfabética através do nome do objeto. Uma vez que o critério foi determinado, existem duas maneiras distintas para proceder à ordenação da coleção:

- 1) O Java deve poder comparar dois objetos do mesmo tipo de modo a determinar se, segundo nosso critério de ordenação, um objeto A é maior, menor ou igual a um objeto B. Para obtermos tal funcionalidade, basta fazer com que a classe implemente a interface Comparable. Esta interface estabelece, através de seu contrato, que todas as classes que a implementarem devem fornecer uma implementação para o método compareTo(), tornando-as “comparáveis”. O Quadro 16 demonstra a implementação do método compareTo() na classe Pessoa. Perceba que na linha 3 declaramos a interface, o que obriga a implementação efetiva do método nas linhas 10 a 13. Como o critério de comparação escolhido foi o nome, basta delegar a comparação para o método compareTo da classe String. Pronto! A partir deste momento uma lista de pessoas pode ser ordenada alfabeticamente. O Quadro 17 mostra de que forma a ordenação é feita no código fonte.

QUADRO 72 – IMPLEMENTAÇÃO DA INTERFACE COMPARABLE

```
3 class Pessoa implements Comparable<Pessoa>{  
4  
5     private String cpf;  
6     private String nome;  
7  
8  
9  
10    @Override  
11    public int compareTo(Pessoa arg0) {  
12        return this.getNome().compareTo(arg0.getNome());  
13    }  
14}
```

FONTE: O autor

QUADRO 73 – ORDENAÇÃO DE UMA LISTA

```

10 class Testador {
11
12 public static void main(String[] args) {
13
14     Pessoa p1 = new Pessoa("8373635", "Juliana");
15     Pessoa p2 = new Pessoa("8223635", "Karol");
16     Pessoa p3 = new Pessoa("2345677", "Cecilia");
17     Pessoa p4 = new Pessoa("2349999", "Anne");
18
19     List<Pessoa> lista = new ArrayList<>();
20     lista.add(p1); lista.add(p2);
21     lista.add(p3); lista.add(p4);
22
23     Collections.sort(lista);
24     for (Pessoa pessoa : lista)
25         System.out.println(pessoa.getNome());
26 }
27 }
```

FONTE: O autor

O método estático `sort()` da classe `Collections` recebe uma lista de objetos do tipo `Comparable` e os ordena de acordo com o critério definido no método `compareTo()`. Mas de que forma essa ordenação ocorre? Perceba que o método retorna um inteiro. Caso os dois objetos sejam iguais, o retorno é igual a 0, caso o objeto que vem como parâmetro é maior, o retorno é um número negativo e, caso o objeto que vem como parâmetro seja menor, o retorno é um número positivo. Como a ordenação propriamente dita ocorre, não faz parte do escopo deste caderno. Se os objetos da lista não implementarem a interface `Comparable`, o método não resulta em alteração na ordem dos objetos. Perceba que a listagem dos nomes dos objetos agora ocorre em ordem alfabética.

FIGURA 56 – LISTAGEM DA COLEÇÃO ORDENADA

```
<terminated> Testador (11) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (06/11/2014 13:49:49)
Anne
Cecilia
Juliana
Karol
```

FONTE: O Autor

- 2) Nem sempre teremos acesso às classes que precisam ser ordenadas na coleção e é nestes casos que a segunda forma de fazer a ordenação entra em ação. Para exemplificarmos esta ordenação, adicionaremos um atributo chamado `altura`, do tipo `double` na classe `Pessoa`, forçando sua atribuição no construtor e fornecendo acesso através de Getters e Setters (Quadro 74). Para este segundo exemplo, queremos ordenar a coleção da pessoa mais alta para a mais baixa, ou seja, em ordem decrescente.

QUADRO 74 – ALTERAÇÃO NA CLASSE PESSOA

```

3 class Pessoa {
4
5     private String cpf;
6     private String nome;
7     private double altura;
8
9     public double getAltura() {
10         return altura;
11     }
12
13     public void setAltura(double altura) {
14         this.altura = altura;
15     }
16
17     public Pessoa(String cpf, String nome, double altura) {
18         this.cpf = cpf;
19         this.nome = nome;
20         this.altura = altura;
21     }

```

FONTE: O autor

Neste caso removemos a implementação da interface Comparable e isentamos a classe pessoa de qualquer responsabilidade na ordenação de uma coleção, simulando uma situação onde não é possível fazer alterações em uma classe. Para esta situação, o Java fornece a utilização de classes conhecidas como Providers. Os providers fornecem parâmetros de comparação entre dois objetos e são utilizados pelo método Sort da classe Collections para ordenar a coleção. No Quadro 75 ilustramos a criação de um provider que fornece ordenação pela altura, de forma decrescente.

QUADRO 75 – PROVIDER PARA ORDENAÇÃO

```

4
5 public class ProviderAltura implements Comparator<Pessoa>{
6
7     @Override
8     public int compare(Pessoa p1, Pessoa p2) {
9         return new Double(p2.getAltura()).compareTo(p1.getAltura());
10    }
11 }

```

FONTE: O autor

Perceba que, para a classe se tornar um provider, ela deve implementar a interface Comparator e, consequentemente, fornecer a implementação de um método compare. No caso do exemplo, observe que criamos um objeto Double com a altura de p2 e chamamos o método compareTo mandando a altura do primeiro objeto. Essa inversão da ordem dos parâmetros é que fará com que a ordenação ocorra de forma decrescente. No último exemplo, a ordem apresentada foi alfabética crescente, então compararamos os nomes dos objetos na ordem correta. O método compare retorna um int e para garantirmos um resultado correto para a comparação entre dois valores do tipo double, fazemos uso da técnica conhecida como *boxing* para chamar o método compareTo (linha 9). A partir deste momento, podemos instanciar o provider e passá-lo como parâmetro para o método Sort. No Quadro 76 mostramos um testador para o novo critério de ordenação.



A técnica de boxing e unboxing consiste em utilizar um objeto do tipo wrapper ao invés de um tipo primitivo. Em nosso exemplo, usamos Double ao invés de double.

QUADRO 76 – PROVIDER EXTERNO E ORDENAÇÃO

```

10 class Testador {
11
12     public static void main(String[] args) {
13
14         Pessoa p1 = new Pessoa("8373635", "Juliana", 1.67);
15         Pessoa p2 = new Pessoa("8223635", "Karol", 1.72);
16         Pessoa p3 = new Pessoa("2345677", "Cecilia", 1.78);
17         Pessoa p4 = new Pessoa("2349999", "Anne", 1.77);
18
19         List<Pessoa> lista = new ArrayList<>();
20         lista.add(p1); lista.add(p2);
21         lista.add(p3); lista.add(p4);
22
23         Collections.sort(lista, new ProviderAltura());
24         for (Pessoa pessoa : lista)
25             System.out.println(pessoa.getNome());
26     }
27 }
```

FONTE: O autor



A partir da versão 8, a linguagem Java já permite a ordenação de coleções através de Streams e Lambda expressions. Para maiores detalhes acesse: <<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>>.

A primeira parte exige que alteremos nossos objetos, de forma a conterem o valor da altura de cada pessoa (linhas 14 a 17). Uma vez instanciados os objetos, adicionamos cada um individualmente à lista (linhas 20 e 21). Na linha 23 chamamos o método sort da classe Collections, passando nossa lista e o provider como parâmetros. Finalmente, entre as linhas 24 e 25 fazemos a listagem das pessoas pela altura, de forma decrescente. O resultado pode ser visto na Figura 57.

FIGURA 57 – LISTAGEM DA ALTURA DAS PESSOAS

```
<terminated> Testador (11) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (10/11/2014 12:47:41)
Cecilia
Anne
Karol
Juliana
```

FONTE: O autor

RESUMO DO TÓPICO 1

- As coleções são recursos oferecidos pelas linguagens de programação orientadas a objetos para facilitar a vida dos programadores quando há a necessidade de utilização de tipos homogêneos de dados.
- As coleções visam substituir os vetores com diversas vantagens, sendo que as principais são o desempenho e a existência de métodos utilitários.
- As coleções podem conter QUALQUER tipo de dados, sendo tipadas através dos generics.
- As ArrayLists são estruturas semelhantes aos vetores, possuindo um número ilimitado de elementos e a capacidade de acesso aos objetos através de um índice.
- Os métodos add e remove são utilizados para adicionar e remover os objetos de uma lista.
- Os HashMaps são estruturas que utilizam de pares do tipo chave/valor para armazenar objetos.
- Tanto a chave quanto o valor podem ser de qualquer derivado de object, ou seja, qualquer tipo de dados.
- A ordem de armazenamento dos elementos não necessariamente é a mesma de inserção, ou seja, ao contrário das listas, a ordem não é importante para o HashMap.
- OHashMap permite duplicatas desde que sejam referenciadas por índices distintos.
- Em contrapartida, é perfeitamente permitido que um objeto seja referenciado por mais de uma chave.
- Podemos pegar objetos em um HashMap através de sua chave e verificar a existência de objetos através do valor.
- Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.
- Outra característica fundamental é que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.
- A vantagem de um set ao invés de uma lista é que a busca por um elemento em um set é mais performática.

- A sugestão é que, sempre que não haja a necessidade da ordem dos elementos, a coleção utilizada seja a set.
- O operador == compara a igualdade entre tipos primitivos e a localização da referência em memória, no caso de objetos.
- Caso haja a necessidade de comparar igualdade entre objetos, o java disponibiliza um mecanismo através de uma implementação polimórfica do método equals.
- O método equals permite a comparação de acordo com o critério que o implementador desejar, podendo utilizar um ou n atributos para tal.
- O hashCode é um atributo para auxiliar a distribuição dos objetos em collections que utilizam tabelas hash para o armazenamento.
- Dois objetos que retornem true para uma comparação com o método equals, devem necessariamente ter o MESMO valor para hashCode.
- Algumas coleções, como a ArrayList, permitem a ordenação dos objetos através de critérios definidos pelo implementador.
- A ordenação pode ser feita através da própria classe, que deve implementar a interface Comparable ou através de um provider, que é uma classe externa que implementa a interface Comparator.
- O método que efetivamente faz a ordenação é o método estático sort, da classe Collections.

AUTOATIVIDADE



1 Assinale a alternativa CORRETA:

- a) Um HashSet considera a ordem de inserção, quando se procura buscar determinado elemento.
 - b) Existem três maneiras de se ordenar coleções no Java:
 - Implementação da interface Comparator na classe que se deseja ordenar.
 - Implementação da interface Comparable em um provider.
 - Através de lambda expressions.
 - c) O método equals determina a igualdade entre dois objetos com base em seu endereço na memória.
 - d) O método hashCode é utilizado para auxiliar o espalhamento dos objetos nas tabelas hash.
- 2 Descreva linha por linha a funcionalidade do método equals mostrado no Quadro 69 deste tópico.
- 3 Explique detalhadamente as diferenças entre as três coleções a seguir:
- a) HashSet.
 - b) HashMap.
 - c) ArrayList.
- 4 Crie um exemplo de código fonte onde uma coleção de objetos do tipo automóvel é ordenada de forma crescente pela placa através de um provider. Os demais atributos são marca, quilometragem e modelo. Crie um testador e as demais classes de forma a demonstrar a ordenação e criação do critério para tal.



STATIC E TRATAMENTO DE EXCEÇÕES

1 INTRODUÇÃO

Você já deve ter percebido que em todos os nossos programas Testadores o método main é marcado com o modificador static. Outro exemplo que acabamos de ver é o método estático sort, da classe Collections. Mas o que significa utilizar esse modificador? Qual é a consequência de sua utilização em nossos programas? Veremos a resposta para todas estas perguntas nesta unidade.

O segundo conteúdo diz respeito a uma questão arquitetural da plataforma Java: o tratamento de exceções. Provavelmente, em seus exercícios e na cópia dos exemplos do caderno, você já deve ter se deparado com muitas delas, em especial a NullPointerException. Mas o que é uma exceção? O que causa uma exceção? De que forma devo proceder em minha aplicação? Entenderemos a resposta para estas perguntas ao abordarmos o mecanismo de tratamento de exceções da plataforma Java.

Prontos para me acompanharem em mais esse desafio de programação? Então vamos lá. Mão à obra!

2 STATIC

Um objeto é uma instância de uma classe, ou seja, uma ocorrência de uma classe em memória, com valores próprios para seus atributos e implementações específicas para seus métodos. Conforme veremos neste tópico, nem sempre é desejável que determinado objeto seja instanciado, pois a instanciação é sempre o momento mais custoso em termos de performance e uso de memória para sua aplicação. Vamos exemplificar a utilização do modificador static em um atributo de uma classe, conforme ilustrado no Quadro 77.

QUADRO 77 – MODIFICADOR STATIC EM UM ATRIBUTO

```
3 public class ContaCorrente {  
4     private double saldo;  
5     private int numero;  
6     private static double taxaDeJuros;  
7  
8     public double getTaxaDeJuros() {  
9         return taxaDeJuros;  
10    }  
11  
12    public void setTaxaDeJuros(double taxa) {  
13        taxaDeJuros = taxa;  
14    }  
15  
16  
17    public ContaCorrente(int nro){  
18        numero = nro;  
19    }
```

FONTE: O autor

No quadro, podemos perceber que o atributo taxaDeJuros foi colocado na classe, juntamente com o modificador static e acessores para o mesmo. Ainda aparece um construtor para a classe, que exige um número de conta. Os demais acessores e métodos foram ocultos por questão de espaço.

Para entendermos de que forma o modificador static funciona, vamos instanciar dois objetos do tipo ContaCorrente no Quadro 78 e verificar o que acontece. Nas linhas 10 e 13 atribuímos valores diferentes para as taxas de juros dos objetos c1 e c2 e nas linhas 15 e 16, fazemos a impressão de cada um deles, através do método `toString()` que trará o número da conta seguido da taxa de Juros.

QUADRO 78 – INSTANCIACÃO DOS OBJETOS

```

3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         ContaCorrente c1 = new ContaCorrente(1);
10        c1.setTaxaDeJuros(1.3);
11
12        ContaCorrente c2 = new ContaCorrente(2);
13        c2.setTaxaDeJuros(1.7);
14
15        System.out.println(c1);
16        System.out.println(c2);
17    }
18 }
```

FONTE: O autor

Ao executarmos o programa, obtemos o resultado mostrado na Figura 60. Perceba que as DUAS contas correntes estão com o mesmo valor para a taxa de juros. Mas por que isso acontece, visto que atribuímos 2 valores diferentes? Esse é o objetivo do modificador static, ou seja, não permitir a utilização de instâncias. Neste caso específico, TODOS os objetos da classe ContaCorrente compartilham o MESMO atributo. Por este motivo ao modificar o valor de uma instância modificamos automaticamente o valor de todas, conforme mostrado.

FIGURA 58 – EXECUÇÃO DO PROGRAMA

```

<terminated> Testador [12] [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (11/11/2014 12:36:14)
ContaCorrente [numero=1, taxaDeJuros=1.7]
ContaCorrente [numero=2, taxaDeJuros=1.7]
```

FONTE: O autor

Existem algumas restrições para a utilização do modificador static, bem como algumas recomendações. Por exemplo, no caso da classe ContaCorrente, os métodos acessores do atributo taxaDeJuros deveriam ser estáticos também, indicando ao implementador qual seu comportamento. Os quadros 79 e 80 ilustram respectivamente a alteração na classe ContaCorrente e no Testador de forma a utilizar o modificador static corretamente.

QUADRO 79 – MÉTODOS ACESSORES ESTÁTICOS

```

3 public class ContaCorrente {
4     private double saldo;
5     private int numero;
6     private static double taxaDeJuros;
7
8     public static double getTaxaDeJuros() {
9         return taxaDeJuros;
10    }
11
12    public static void setTaxaDeJuros(double taxa) {
13        taxaDeJuros = taxa;
14    }
15
16    public ContaCorrente(int nro){
17        numero = nro;
18    }

```

FONTE: O autor

Perceba que no exemplo do Quadro 80 passamos a utilizar o acessor da conta corrente através de seu método estático (linhas 10 e 13). O resultado prático desta alteração é exatamente o mesmo obtido na Figura 60, entretanto, agora sinalizamos ao implementador que o método se refere à classe e não às instâncias.

QUADRO 80 – CHAMADA AO MÉTODO ATRAVÉS DA CLASSE

```

3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         ContaCorrente c1 = new ContaCorrente(1);
10        ContaCorrente.setTaxaDeJuros(1.3);
11
12        ContaCorrente c2 = new ContaCorrente(2);
13        ContaCorrente.setTaxaDeJuros(1.3);
14
15        System.out.println(c1);
16        System.out.println(c2);
17    }
18 }

```

FONTE: O autor

A restrição ao modificador static refere-se ao acesso de membros não estáticos. Por exemplo, ao criarmos um método estático em uma classe, este método não pode acessar nenhum atributo não estático. Como assim? É simples entender o motivo, uma vez que se entende o significado de estático. Estático significa compartilhado entre TODAS as instâncias de uma classe, ou seja, todas as classes utilizam o mesmo atributo ou método. Caso eu queira fazer uso de determinado atributo específico de uma instância, como o método estático saberia de QUAL instância estou falando? Esta determinação não é possível, o que torna todos os atributos não estáticos inacessíveis dentro de um método estático. O contrário já não é verdade, visto que TODAS as instâncias saberiam tranquilamente como acessar um método ou atributo estático.

Para exemplificar esta situação faremos uma alteração no método estático setTaxaDeJuros da classe ContaCorrente (Quadro 81). Na linha 13 perguntamos se o atributo saldo é maior do que zero. Perceba que o eclipse nos informa exatamente o que acontece, ou seja, não é possível acessar um membro não estático em um membro estático. Como o método é compartilhado entre TODAS as instâncias, não é possível saber a qual das instâncias o atributo saldo está relacionado e seu valor respectivo.

Existem diversas situações onde a utilização do modificador static é recomendada. Em geral, métodos estáticos são úteis quando a classe não tem estado, ou seja, não possui valor para seus atributos e é responsável somente por alguma tarefa. O Collections.sort() é um exemplo perfeito desta utilização, pois simplesmente mandamos como parâmetro uma coleção e a recebemos como retorno, ordenada de acordo com os critérios que definimos. Não há necessidade de colocar valor em nenhum atributo, o que a deixa sem estado e, consequentemente, sem necessidade de instância.

QUADRO 81 – ACESSO A MEMBRO NÃO ESTÁTICO EM MÉTODO ESTÁTICO

```

3 public class ContaCorrente {
4     private double saldo;
5     private int numero;
6     private static double taxaDeJuros;
7
8     public static double getTaxaDeJuros() {
9         return taxaDeJuros;
10    }
11
12    public static void setTaxaDeJuros(double taxa) {
13        if (saldo > 0)
14            // Cannot make a static reference to the non-static field saldo
15    }
16
17    public C
18        numero = nro;
19    }

```

FONTE: O autor

No Quadro 82 mostramos outro exemplo, onde uma classe utilitária chamada de DataUtil possui um método estático para calcular a idade de uma pessoa com base em uma String no formato dd/MM/yyyy, que representa a data de nascimento. A chamada desse método seria feita através de DataUtil.CalcularIdade("20/03/1978"), retornando o valor da idade em anos. A classe DataUtil não precisa de nenhum atributo e simplesmente faz cálculos referentes a data, o que a torna uma candidata ideal à implementação de métodos estáticos.

QUADRO 82 – CLASSE ESTÁTICA DATAUTIL

```

22 // Calcula a Idade baseado em java.util.Date
23 public static int calcularIdade(String dataNasc) {
24     Calendar dateOfBirth = new GregorianCalendar();
25     dateOfBirth.setTime(stringToDate(dataNasc));
26     // Cria um objeto calendar com a data atual
27     Calendar today = Calendar.getInstance();
28     // Obtém a idade baseado no ano
29     int age = today.get(Calendar.YEAR) - dateOfBirth.get(Calendar.YEAR);
30     dateOfBirth.add(Calendar.YEAR, age);
31     // se a data de hoje é antes do Nascimento, então diminui 1
32     if (today.before(dateOfBirth)) {
33         age--;
34     }
35     return age;
36 }
```

FONTE: O autor

A plataforma Java faz uso de métodos estáticos em diversas APIs, visto que economizam o custo de utilização do **new** e criação de uma instância. Apesar de apresentar vantagens tanto conceituais quanto práticas, é importante avaliar com parcimônia essa prática, visto que a criação de classes e métodos estáticos em excesso aumenta o uso de memória pela máquina virtual Java.

2.1 SINGLETON

Em algumas situações, existe a necessidade de se evitar que mais de uma instância de uma classe seja criada. Digamos que exista uma conexão com o banco de dados e que exista uma preocupação quanto ao número máximo de conexões que este banco aceite. Neste caso, esta conexão deve ser compartilhada entre um número de usuários do sistema, repassando para o próprio banco a responsabilidade de gerenciá-lo. Mas de que forma eu consigo impedir que determinado objeto seja instanciado? Este tipo de situação não é tão incomum, o que acabou resultando em um padrão de projeto para resolver este problema.



Padrões de Projetos são soluções testadas para problemas recorrentes no desenvolvimento de software. Consistem simplesmente na utilização de práticas e técnicas da orientação a objetos. Para saber mais sobre padrões de projetos, sugerimos a leitura de Silveira, Silveira e Lopes (2011).

O padrão de projeto é denominado de Singleton e possui a seguinte definição na WikiPedia: “Em engenharia de *software*, o padrão de projeto singleton é um padrão de projeto que restringe a instanciação de uma classe para um objeto. Esta prática é útil quando exatamente um objeto é necessário para coordenar ações dentro do sistema”. Este é um tópico onde o modificador static é extremamente útil, e através dele, conseguimos garantir que todos compartilhem a mesma instância de uma classe.

Para entender esse padrão, analise o código do Quadro 83. Perceba que na linha 5 criamos um atributo privado static do tipo da própria classe, ou seja Singleton. Este atributo estático será retornado quando alguém solicitar uma instância. Mas como impediremos alguém de instanciar um objeto? Pense sobre a forma através da qual todos os objetos são instanciados. Através do new, você responde. Sua resposta está corretíssima, afinal todos os objetos são instanciados primariamente através deste comando. Mas o que ocorre quando você chama o new de uma classe? O construtor é invocado, o que cria a nova instância do objeto. Aí é que reside a chave do singleton. Colocamos um construtor privado, na linha 7, o que proíbe qualquer classe externa a chamá-lo e consequentemente, usar o new. O singleton define que as instâncias retornam através de um método estático, e que este método estático deve retornar sempre à mesma instância. Perceba que na linha 10 criamos uma condição perguntando se a variável é nula e, caso for, criamos a instância. Não há problema em usar o new, afinal estamos dentro da mesma classe. Caso a instância não seja nula, retornamos à mesma que já foi criada anteriormente, afinal ela é estática.

QUADRO 83 – SINGLETON

```

2
3 public class Singleton {
4
5     private static Singleton instancia;
6
7     private Singleton() {}
8
9     public static Singleton getInstance(){
10        if (instancia == null)
11            instancia = new Singleton();
12        return instancia;
13    }
14 }
```

FONTE: O autor

Para fazer a chamada deste objeto, observe as linhas 9 e 11 do Quadro 84. Criamos dois objetos da classe singleton através do método getInstance() e em seguida mandamos imprimir o objeto. O comportamento padrão da impressão de um objeto (sem reescrevermos o método `toString()`) é imprimir o nome completo da classe e o seu endereço na memória, separados por um símbolo @. A figura 59 traz o resultado da execução abaixo, provando que, apesar de criarmos dois objetos, os dois referenciam o mesmo endereço de memória, ou seja, a mesma instância.

QUADRO 84 – CRIAÇÃO DO OBJETO SINGLETON

```

3 public class Testador {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9         Singleton singleton1 = Singleton.getInstance();
10
11         Singleton singleton2 = Singleton.getInstance();
12
13         System.out.println(singleton1);
14         System.out.println(singleton2);
15     }
16 }
```

FONTE: O autor

FIGURA 59 –TESTE DO SINGLETON

```
<terminated> Testador [13] [Java Application] C:\Program Files\Java\jre1\bin\javaw.exe (13/11/2014 13:03:26)
singleton.Singleton@3be67280
singleton.Singleton@3be67280
```

FONTE: O autor

Atualmente, o Singleton é considerado um antipattern, e sua utilização deve ser avaliada com cuidado. Conforme Silveira (2006), estado mantido de forma estático em uma aplicação é tão venenoso quanto variáveis globais. A utilização de singletons é um indício de que você precisa pensar melhor sua arquitetura.

Não existe bala de prata em desenvolvimento de *software* e, apesar de ser considerado perigoso, o singleton e os métodos estáticos podem ser úteis em determinadas situações. Cabe a você, assim que obtiver experiência e discernimento suficiente em programação orientada a objetos, decidir sobre sua utilização em seu projeto. Como quase tudo em desenvolvimento de *software*, a escolha sobre métodos estáticos e singletons representa uma troca. Você abre mão de uma característica para dar destaque a outra.

2.2 TRATAMENTO DE EXCEÇÕES

Uma exceção é disparada em seu código fonte por um simples e único motivo: uma linha de código não conseguiu concluir sua execução. A plataforma Java possui uma arquitetura robusta para tratamento de exceções, dividindo-as em exceções verificadas e exceções não verificadas.

Para dar início ao estudo das exceções, observe o código a seguir. Perceba que o método main chama o metodo1 e esse, por sua vez, chama o metodo2, cada um com suas próprias variáveis locais. Para forçarmos um erro, criamos um vetor de 10 posições e logo em seguida tentamos iterar esse vetor até a posição 15. Como o vetor inicia no 0, quando tentarmos acessar a posição 10, essa linha não conseguirá ser executada e consequentemente disparará uma exceção, conforme mostrado na Figura 60.

QUADRO 85 – EXEMPLO DE EXCEÇÃO

```

3  public class TestadorErro {
4    public static void main(String[] args) {
5      System.out.println("inicio do main");
6      metodo1();
7      System.out.println("fim do main");
8    }
9
10   static void metodo1() {
11     System.out.println("inicio do metodo1");
12     metodo2();
13     System.out.println("fim do metodo1");
14   }
15
16   static void metodo2() {
17     System.out.println("inicio do metodo2");
18     int[] array = new int[10];
19     for (int i = 0; i <= 15; i++) {
20       array[i] = i;
21       System.out.println(i);
22     }
23     System.out.println("fim do metodo2");
24   }

```

FONTE: Adaptado de Caelum (2014)

Essa é o conhecido rastro da pilha (stacktrace). É uma saída importantíssima para o programador – tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace. Por que isso aconteceu? O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é lançada (throws), a JVM entra em estado de alerta e verifica se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como o metodo2 não toma nenhuma medida diferente do que vimos até agora, a JVM interrompe a execução dele anormalmente, sem esperar seu término e volta um stackframe para baixo, onde será feita nova verificação: o metodo1 está se precavendo de um problema chamado `ArrayIndexOutOfBoundsException`? Não... volta para o main, onde também não há proteção, então a JVM morre e o programa deixa de executar (CAELUM, 2014).

FIGURA 60 – DISPARO DE EXCEÇÃO

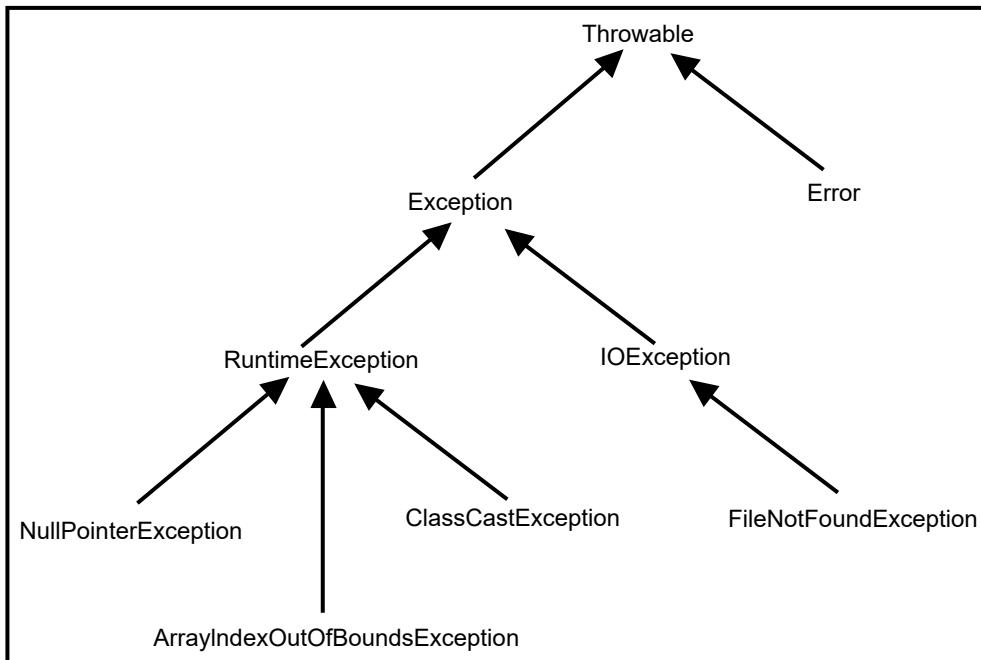
```
<terminated> TestadorErro [Java Application] C:\Program Files\Java\jre\bin\javaw.exe (14/11/2014 12:28:51)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at excecoes.TestadorErro.metodo2(TestadorErro.java:20)
at excecoes.TestadorErro.metodo1(TestadorErro.java:12)
at excecoes.TestadorErro.main(TestadorErro.java:6)
```

FONTE: O autor

Essa mensagem de erro, apesar de ser assustadora no início, é um dos recursos mais valiosos para o programador. Perceba que ela indica exatamente onde ocorreram os erros. O primeiro erro ocorreu na classe TestadorErro, no método 2 linha 20. Depois a stacktrace informa que o metodo2 foi chamado pelo metodo1, na linha 12. Finalmente, o stacktrace segue até o método que chama o metodo1, no nosso caso, o main na linha 6. O stacktrace volta a execução de traz para frente procurando por um tratamento da exceção disparada. No eclipse, basta clicar na linha destacada através do stacktrace para ser direcionado para o local exato no código-fonte. Caso nenhum tratamento seja encontrado até o main, a execução é interrompida.

Um detalhe importantíssimo é que as exceções são como qualquer outro tipo de classe do java, ou seja, possuem métodos e atributos, permitem herança (o que torna possível a você escrever suas próprias exceções) e polimorfismo. Neste caso existe uma exceção chamada `ArrayIndexOutOfBoundsException`, que indica já em seu nome que tentamos acessar um índice inexistente de um array. Existe toda uma hierarquia de exceções dentro da plataforma Java, categorizadas de acordo com o tipo de situação a ser tratada. A Figura 61 mostra uma parte dessa hierarquia.

FIGURA 61 – HIERARQUIA DE EXCEÇÕES



FONTE: Caelum (2014)

Existem maneiras de prevenir que uma exceção interrompa a execução do programa e “estoure” na mão do usuário, continuando a executar normalmente. Uma das maneiras é cercar o código perigoso com um comando try, conforme o Quadro 86.

QUADRO 86 – CAPTURA DA EXCEÇÃO

```

16  static void metodo2() {
17      System.out.println("inicio do metodo2");
18      int[] array = new int[10];
19      try {
20          for (int i = 0; i <= 15; i++) {
21              array[i] = i;
22              System.out.println(i);
23          }
24      } catch (ArrayIndexOutOfBoundsException ex) {
25          System.out.println("Capturei o erro e continuei a execução");
26      }
27      System.out.println("fim do metodo2");
28  }
  
```

FONTE: O autor

Perceba que na linha 19, o comando try foi colocado antes da execução do for. Em termos práticos, isso significa dizer para a JVM o seguinte: tente executar esse código, mas fique atento. Na linha 24 fazemos o tratamento da exceção do tipo destacado, onde basicamente dizemos o seguinte: se esse tipo de exceção disparar, faça o que está na linha 25. O resultado da execução pode ser visto na Figura 61.

FIGURA 61 – TRATAMENTO DA EXCEÇÃO



```

inicio do main
inicio do metodo1
inicio do metodo2
0
1
2]
3
4
5
6
7
8
9
Capturei o erro e continuei a execução
fim do metodo2
fim do metodo1
fim do main

```

FONTE: O autor

Com a alteração que realizamos, o programa ainda não consegue executar o comando que tenta acessar a posição inexistente no vetor, entretanto, continua sua execução até o final. Esse é o objetivo do tratamento de exceções: impedir que o programa pare de executar caso alguma exceção seja disparada.

Mas então não é mais fácil garantir e cobrir todo o código com comandos try/catch? Você pode até fazer isso, mas todo o código coberto com tratamento de exceção executa de forma muito menos performática do que o código sem tratamento, o que, na prática, tornaria isso inviável.



No exemplo colocamos o tratamento da exceção direto no método onde o código fonte ocasiona o erro, entretanto, poderíamos colocá-lo antes de chamar o metodo1 ou o metodo2 e obtermos praticamente o mesmo resultado.

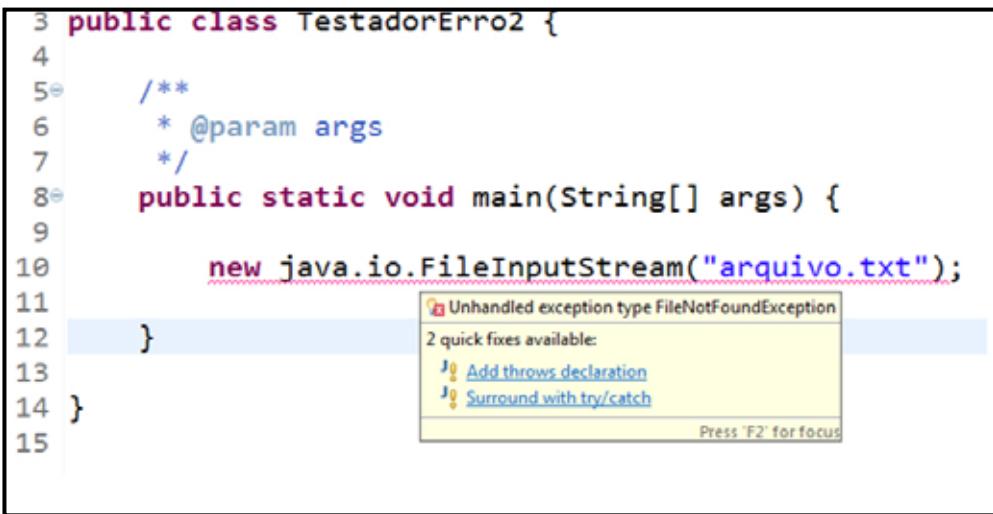
A maioria das exceções da plataforma Java é semelhante a que acabamos de demonstrar, enquadrando-se na categoria de exceções não verificadas. As exceções não verificadas (*unchecked*), como o próprio nome sugere, não exigem tratamento, cabendo ao implementador definir qual o melhor local para colocá-las. Em geral, estas exceções podem ser evitadas com verificações feitas através de validações no código.

Existe outro tipo de exceção da plataforma Java que é conhecida como verificada (*checked*), caracterizando-se por exigir um tratamento por parte do implementador. Esse tipo de exceção é levantado pela plataforma Java quando se tenta acessar um recurso que não diz respeito exclusivamente à JVM. Por exemplo, quando você tenta criar um arquivo em disco, a JVM vai solicitar ao Sistema Operacional que o faça, mas não tem controle sobre o recurso propriamente dito. O mesmo ocorre com acesso à rede, banco de dados etc. Por esse motivo, quando seu código foge à sandbox fornecida pela JVM, existe a obrigatoriedade de se tratar a exceção ou incorrer em um erro de compilação.

Esta situação é demonstrada no Quadro 87, que demonstra a tentativa de criação de um arquivo texto em disco. Perceba que a linha 10 está sublinhada pelo Eclipse, indicando um erro de compilação referente ao não tratamento de uma exceção do tipo `FileNotFoundException`. O próprio Eclipse sugere duas alternativas para o tratamento desta exceção:

- 1) Add throws declaration: neste caso, o método onde o código passível de erro está indica em sua assinatura que pode lançar determinado tipo de exceção. A responsabilidade do tratamento desta exceção fica a cargo de quem chamá-lo.
- 2) Surround with try catch: neste caso, o código passível de erro é cercado pelo `try/catch` e o tratamento da exceção ocorre *in loco*.

QUADRO 87 – EXCEÇÃO VERIFICADA



```

3 public class TestadorErro2 {
4
5     /**
6      * @param args
7      */
8     public static void main(String[] args) {
9
10        new java.io.FileInputStream("arquivo.txt");
11    }
12
13
14 }
15

```

A screenshot of the Eclipse IDE interface. On the left, there is a code editor window containing Java code. Line 10, which contains the statement `new java.io.FileInputStream("arquivo.txt");`, is underlined with a red squiggly line, indicating a syntax error. On the right, a tooltip box is displayed with the following content:

- Unhandled exception type `FileNotFoundException`
- 2 quick fixes available:
- [Add throws declaration](#)
- [Surround with try/catch](#)

Below the tooltip, there is a note: "Press F2 for focus".

FONTE: Adaptado de Caelum (2014)

Saber qual das duas estratégias para o tratamento de exceções utilizar é uma questão complicada. Caelum (2014) afirma que não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

A própria decisão de qual exceção tratar pode ser complicada. No caso do código que tenta criar um arquivo, poderíamos tratar a exceção específica indicada pelo Eclipse, ou ainda qualquer uma de suas superclasses. Por exemplo, se tratássemos IOException ou mesmo Exception, o compilador pararia de indicar o erro. Mas qual é a vantagem de tratar uma exceção de forma genérica? Inicialmente, isso significa que mais de um erro será capturado por aquele tratamento. Ao tratarmos IOException, estamos automaticamente capturando qualquer uma de suas subclasses, entre as quais está inclusa FileNotFoundException. Esse tipo de tratamento deixa o código mais robusto, pois mais de um tipo de erro será automaticamente tratado, entretanto, quanto menos especificada for a exceção disparada ou capturada, mais difícil é a descoberta do que o está causando.

Para exemplificar a diferença, consideremos uma situação onde uma FileNotFoundException é disparada. Ao avaliar o stacktrace, o implementador sabe EXATAMENTE o que está acontecendo. Ao encontrarmos uma Exception, muito provavelmente uma análise mais detalhada do código será necessária para se descobrir o que está acontecendo.

Para evitar problemas relativos a esta situação, o Java permite que se trate mais de uma exceção por instrução try, ou seja, o implementador pode dizer ao compilador para tentar executar determinado código e, caso determinados tipos de erros ocorram, procede-se com o tratamento. O Quadro 32 ilustra esta situação.

Note que o método criarArquivo lança dois tipos de exceção em sua assinatura, uma FileNotFoundException e uma SQLException. Isso delega a responsabilidade de tratamento destas exceções para quem o estiver chamando. No exemplo, o método main faz a chamada para criarArquivo e deve tratá-las ou passá-las para a frente através de outra cláusula throws. Como estamos no método main, não faz sentido passar a exceção para a frente, pois o main é o último método do stacktrace e caso uma exceção seja lançada, ela interromperá a execução de qualquer forma. O próprio Eclipse faz o tratamento, bastando escolher a opção surround with try/catch.



Essa possibilidade de se tratar a exceção in loco ou delegar o tratamento para quem estiver chamando o método é comumente conhecida como handle or declare.

QUADRO 88 – TRATAMENTO DE MÚLTIPLAS EXCEÇÕES

```

7 public class TestadorErro2 {
8
9  public static void main(String[] args) {
10
11     try {
12         criarArquivo("arquivo.txt");
13     } catch (IOException e) {
14
15         e.printStackTrace();
16     } catch (SQLException e) {
17
18         e.printStackTrace();
19     }
20
21 }
22
23 public static void criarArquivo(String nome) throws FileNotFoundException,
24             SQLException {
25     new java.io.FileInputStream(nome);
26 }
27 }
```

FONTE: O autor

2.3 CRIANDO SUAS PRÓPRIAS EXCEÇÕES

Já vimos que as exceções não passam de objetos dentro da linguagem de programação Java. Esta característica permite que qualquer implementador crie suas próprias exceções, adicionando comportamento conforme entender necessário. Por exemplo, é possível criar uma exceção do tipo CPFInvalidoException, informando o usuário ou outro implementador que o CPF digitado não foi validado. Demonstraremos essa situação no Quadro 33.

Perceba que além de estender a superclasse Exception, ainda criamos dois atributos. O primeiro atributo é do tipo enumeração e visa definir um grau de severidade para o erro encontrado, enquanto o segundo é simplesmente um código de erro, que pode auxiliar o suporte técnico. No construtor da classe, invocamos o construtor da superclasse enviando uma mensagem personalizada (linha 9), enquanto nas linhas 10 e 11 imputamos valores específicos para os atributos da classe.

QUADRO 89 – CRIAÇÃO DE UMA EXCEÇÃO PERSONALIZADA

```

3 public class CPFInvalidoException extends Exception{
4
5     private Severidade severidade;
6     private int codigoDeErro;
7
8     public CPFInvalidoException(){
9         super("O CPF digitado é inválido");
10        severidade = severidade.BAIXA;
11        codigoDeErro = 35;
12    }
13}

```

FONTE: O autor

Esse tipo de exceção não verificada serve em geral para informar que uma regra de negócio foi violada e, em geral, não é tratada onde ocorre. Mas por que criar uma exceção se não vamos tratá-la? Cada vez que usamos um bloco try/catch, isso implica perda de performance, então, em caso de regras de negócios, fazemos a validação com o IF e, caso a regra seja quebrada, daí sim lançamos a exceção com o tratamento específico (Quadro 90).

QUADRO 90 – VALIDAÇÃO DE CPF

```

9     public void setCPF(String cPF) throws CPFInvalidoException {
10        if (cPF.length() == 0)
11            throw new CPFInvalidoException();
12        else
13            CPF = cPF;
14    }

```

FONTE: O autor

Neste exemplo, fazemos uma validação fictícia de um CPF verificando simplesmente se o tamanho da string é igual a 0 (linha 10). Atente para o fato de que o atributo somente receberá o valor se a validação for bem sucedida, caso contrário, disparamos uma exceção (linha 11). Ao lançarmos essa exceção, estamos sujeitos à regra *handle or declare*, o que nos obriga a fazer esse tratamento em algum lugar de nosso código.

Existe muita controvérsia na prática de criar suas próprias exceções, afinal, a linguagem possui diversos tipos e é provável que um deles se encaixe em sua necessidade. Em nosso caso, por exemplo, poderíamos lançar uma `IllegalArgumentException`, pois efetivamente o que ocorreu foi que o parâmetro do método (`argument`) continha um valor inválido. Alguns autores, ao contrário, defendem que quanto mais específica for uma exceção, mais específica é a informação sobre o erro, o que facilita sua correção posterior. Mais uma vez, não existe bala de prata, o que obrigará você a decidir qual rumo tomar em seus próprios projetos.



O termo “bala de prata” é comumente utilizado na área de desenvolvimento de software ao se referir a uma solução única para todos os problemas. O termo surgiu inicialmente em Brook (1986), onde o autor defendia que não se pode generalizar uma solução.

RESUMO DO TÓPICO 2

- O modificador static indica que determinado atributo ou método pertence a uma classe e não a uma instância.
- Ao marcarmos um atributo ou método com static, todos os objetos compartilham um valor ou uma implementação específica.
- Ao alterarmos o valor de um atributo do tipo static, alteramos o valor de todas as instâncias daquele objeto; por isso, podemos acessar o atributo ou método através do nome da classe. Por exemplo, Integer.parseInt() é um método da classe Integer que faz a conversão de uma String para um inteiro.
- Em geral, os métodos estáticos são utilizados em classes utilitárias responsáveis por realizarem operações. Isso ocorre porque não há benefício em manter um estado (valor de atributos) para instâncias específicas.
- Um atributo do tipo static não pode acessar um atributo ou utilizar um método que não seja deste tipo.
- Essa restrição ocorre porque o static é compartilhado por todas as classes, enquanto o não static possui valor ou implementação específica. Por exemplo, um atributo número possui valor específico para cada objeto ContaCorrente, enquanto o atributo taxaDeJuros pode possuir o mesmo valor para todos os objetos. Ao tentarmos acessar a taxaDeJuros de um método não estático, não há problema, pois o valor é compartilhado. O contrário já não seria possível, pois se tentássemos acessar o atributo nome de um método estático, não saberíamos a qual nome de qual objeto estamos nos referindo.
- Um singleton é um padrão de projeto para evitar que mais de uma instância de um objeto seja criada.
- Para impedir a criação de mais de um objeto, o singleton utiliza um construtor privado e métodos estáticos.
- Uma exceção consiste basicamente de um objeto lançado pela JVM para indicar que algo não esperado aconteceu.
- Esse algo não esperado é, em geral, uma linha de código fonte que não consegue ser executada.
- Existem diversos tipos de exceção, e como a exceção é uma classe, podemos criar nossas próprias exceções.

- Uma exceção obrigatoriamente deve ser tratada, seja no local que ocorre, seja no local onde o método que lançou a exceção é chamado.
- Essa regra de obrigatoriedade no tratamento de exceções é comumente conhecida como *handle or declare*.
- A linguagem de programação Java possui dois tipos de exceção as checked e as unchecked.
- As checked exceptions são aquelas que a plataforma Java obriga o tratamento, pois acessam recursos externos à JVM, como, por exemplo: rede, disco etc.
- As unchecked exceptions são aquelas em que não existe a obrigatoriedade de tratamento, por exemplo, ao tentarmos acessar um objeto não instanciado (`NullPointerException`) ou ao tentarmos acessar um índice de um vetor fora de seus limites (`ArrayIndexOutOfBoundsException`).

AUTOATIVIDADE



- 1 Um dos problemas do padrão de projetos singleton ocorre porque, em caso de sistemas multithread, se duas threads acessarem o método getInstance simultaneamente, pode ser que mais de uma instância seja criada. Pesquise sobre o tema, implemente um singleton que seja seguro em relação a threads simultâneas no Eclipse e justifique sua solução nas linhas a seguir.

- 2 No exemplo de exceção personalizada, utilizamos uma característica da linguagem de programação Java que ainda não havia sido mostrada: as enumerações. Pesquise sobre o tema e implemente no Eclipse uma classe que faça uso de uma enumeração chamada de Sexo, que contenha os valores MASCULINO E FEMININO.

- 3 Com relação ao conteúdo do Tópico 2, assinale V para as alternativas VERDADEIRAS e F para as FALSAS:
 - () O modificador static pode ser utilizado quando precisamos de uma variável global.
 - () O modificador static pode ser aplicado em métodos e atributos.
 - () Métodos estáticos são normalmente característicos de classes utilitárias que não precisam de estado.
 - () Uma exceção somente é disparada quando ocorre um erro no código fonte.
 - () Uma ArithmeticException é um tipo de exceção verificada.

- 4 Descreva detalhadamente por que a utilização de métodos e atributos do tipo static pode, em algumas situações, ser nociva para seu código-fonte.

- 5 O Singleton é um padrão de projeto desenvolvido para uma finalidade específica. Pesquise sobre o tema Padrões de Projeto e implemente um deles no Eclipse, justificando a sua aplicabilidade.



JDBC E DAO

1 INTRODUÇÃO

A maioria das aplicações desenvolvidas no mercado tem a necessidade de persistir suas informações em alguma espécie de repositório que permita a recuperação das mesmas em um momento posterior. Por exemplo, ao realizar uma venda para um cliente, é interessante que os dados desta e das próximas vendas sejam armazenados e posteriormente consultados, no sentido de determinar o perfil deste cliente.

Atualmente, o tipo de repositório mais utilizado para persistir essas informações é o banco de dados relacional. Provavelmente, você já ouviu falar em Oracle, MySQL, SQLServer, DB2, PostgreSQL, entre outros, pois são os bancos de dados mais comumente utilizados na atividade de desenvolvimento de *software*. O grande volume de utilização dos bancos de dados relacionais ocorre basicamente devido a duas razões básicas:

- 1) Os bancos de dados relacionais já estão no mercado por um longo período de tempo, portanto, possuem um grau de confiabilidade alto.
- 2) Os paradigmas que vieram antes do paradigma de programação orientada a objetos eram completamente compatíveis com os bancos de dados relacionais.

A linguagem de programação Java possui uma API projetada unicamente para se trabalhar com bancos de dados relacionais, conhecida como JDBC (Java Database Connectivity) e que dá suporte a um grande número de produtos existentes no mercado. O detalhe aqui é o seguinte: existe um grande problema ao utilizar a linguagem de programação Java (e qualquer outra linguagem de programação orientada a objetos) em combinação com os bancos de dados relacionais. Antes de demonstrarmos a utilização do JDBC, é importante ilustrar esse problema e fazer uma contextualização no conceito conhecido como mapeamento objeto relacional.

2 MAPEAMENTO OBJETO RELACIONAL

Toda a teoria da programação orientada a objetos é baseada na teoria dos grafos. Essencialmente, quando temos uma cadeia de objetos inter relacionados, podemos vê-los na memória como um grafo direcionado e navegável.



Grafos são estruturas normalmente utilizadas para representar dados relacionados de forma não convencional. Um exemplo clássico desse tipo de representação são os mapas utilizados em serviços de GPS. Maiores detalhes sobre os grafos podem ser obtidos em: <<http://www.ime.usp.br/~pf/teoriadosgrafos/>>.

Por outro lado, os bancos de dados relacionais são baseados na teoria dos conjuntos, o que pode ser facilmente comprovado quando se faz uma pesquisa buscando dados de mais de uma tabela através dos conectivos E e OU.

A verdade é que precisamos instanciar objetos em nossas aplicações e armazenar seus estados em algum mecanismo de persistência eficiente. Analisando friamente, os bancos de dados relacionais não são a resposta mais coerente para esse problema, entretanto, como dissemos antes, um grande número de aplicações legadas utiliza os bancos de dados relacionais há anos (se não décadas), o que os consolidou no mercado. A combinação de um modelo com forte base científica e a vasta disponibilidade de literatura e ferramentas fazem com que as bases relacionais sejam uma escolha sólida para a maioria das aplicações em desenvolvimento (CARDIM, 2010).

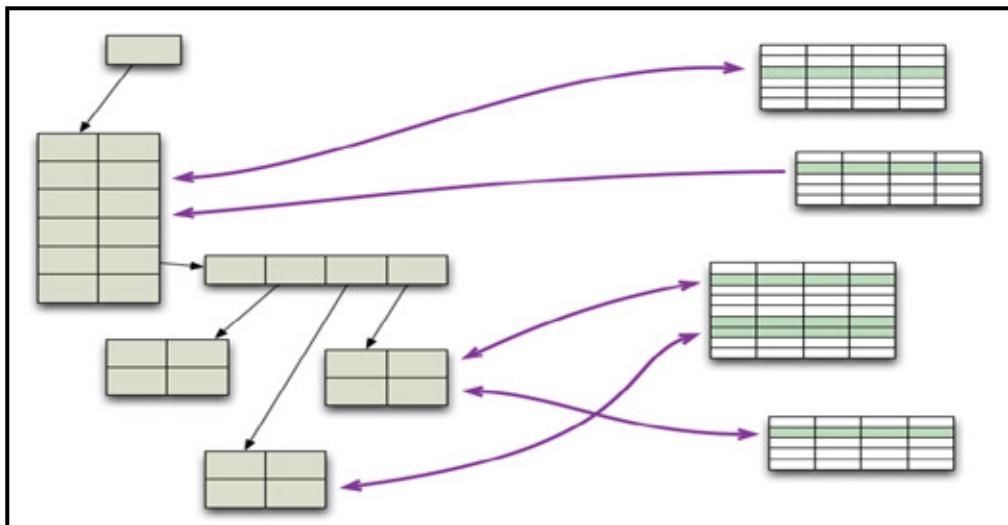
Existem alguns bancos de dados não relacionais e até mesmo orientados a objetos, mas a fatia de mercado ocupada por este tipo de mecanismo de persistência é, até o momento, praticamente insignificante. Faremos uma breve menção a estes bancos de dados ainda neste tópico.

Cardim (2010) afirma que objetos, relações e registros são conceitos de natureza fundamentalmente diferente, apesar de apresentarem algumas similaridades deceptivas. A navegação entre objetos ocorre através de ponteiros e referências, enquanto as relações são unidas através de produtos cartesianos e chaves estrangeiras, o que acaba levando a soluções de otimização diferentes para ambos os casos, além das próprias operações de acesso.

Ainda sobre o tema, Fowler (2012) coloca que o problema de mapeamento objeto relacional é um problema difícil de ser resolvido, pois o que se faz é sincronizar dois tipos completamente diferentes de representação de dados, um contido em uma base de dados relacional e outro na memória.

A Figura 62 (Martin Fowler) exemplifica uma tentativa de sincronização. Perceba que os objetos relacionados entre si, representados do lado esquerdo da figura, são mapeados para as tabelas do lado esquerdo. Enquanto os objetos possuem como única limitação, a memória do computador, as tabelas precisam representar cada objeto em forma de um registro único, o que torna a tarefa complicada.

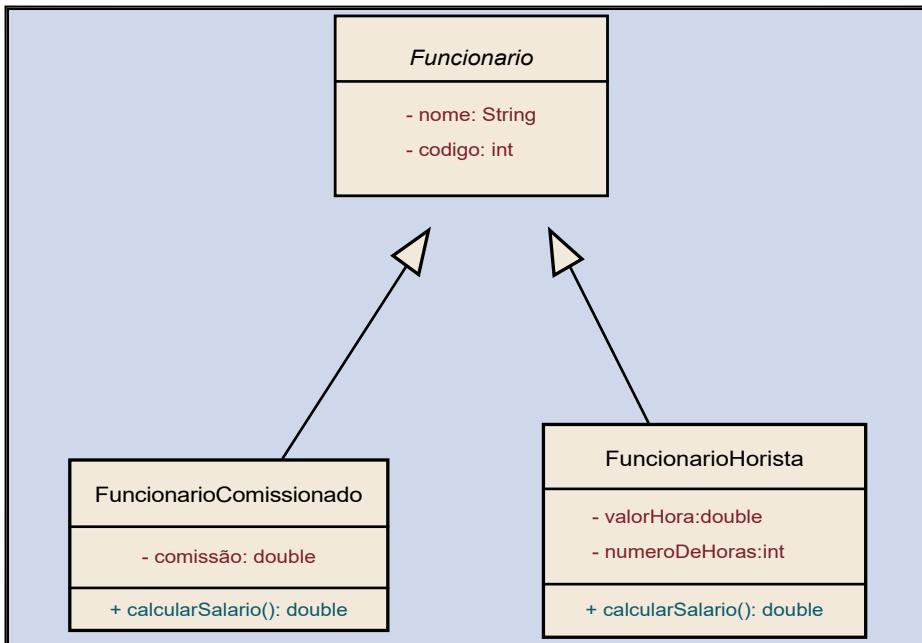
FIGURA 62 – MAPEAMENTO OBJETO RELACIONAL



FONTE: Adaptado de Fowler (2012)

Este problema é comumente conhecido como diferença de impedância e resolvido através de um conjunto de técnicas e ferramentas conhecido como mapeamento objeto relacional. Para entender a complexidade de tal problema, imagine uma estrutura de classes que represente parte do domínio de um aplicação, conforme ilustrado na Figura 63.

FIGURA 63 – ESTRUTURA DE HERANÇA



FONTE: O autor

Aqui temos um relacionamento de herança envolvendo três classes. Lembre-se de que tanto *FuncionarioComissionado* quanto *FuncionarioHorista* compartilham os atributos de *Funcionario* (que é abstrato) e ainda possuem seus próprios atributos. Como faremos para armazenar objetos deste tipo em um banco de dados relacionais?

Existem diversas técnicas documentadas e exemplificadas na literatura e internet para o mapeamento do relacionamento de herança entre objetos para tabelas relacionais, entretanto, visando cumprir com os objetivos de aprendizado deste caderno, nos ateremos a mapeamentos menos complexos e de representação mais fácil.

3 ARQUITETURA DO DATA ACCESS OBJECT

Uma das soluções mais aplicadas para o problema do mapeamento objeto relacional é a utilização do padrão de projeto conhecido como DAO (data access object). Esse padrão consiste em implementar uma camada para fazer o mapeamento entre os objetos e as tabelas do banco de dados. Essa camada pode conter diversas classes e ser construída de forma manual ou até mesmo através da utilização de frameworks para automatizar o processo.



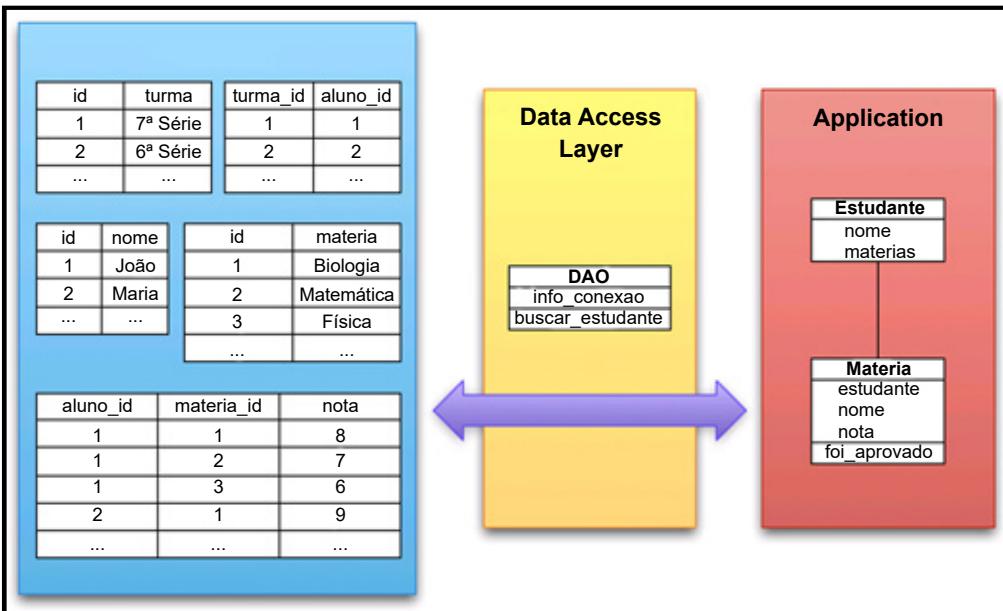
Um framework provê uma solução para uma família de problemas semelhantes, usando um conjunto de classes e interfaces que mostra como decompor a família de problemas e como objetos dessas classes colaboram para cumprir suas responsabilidades. O conjunto de classes deve ser flexível e extensível para permitir a construção de várias aplicações com pouco esforço, especificando apenas as particularidades de cada aplicação. Fonte: Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>>.

Essencialmente, o que o DAO propõe é a criação de um conjunto de classes onde cada classe representa um conceito de objeto mapeado para uma ou mais tabelas do banco de dados. Em geral, cada DAO possui operações que permitem a criação, leitura, atualização e busca de registros no banco de dados. A função primordial do DAO é fazer a tradução de objetos em registros e de registros em objetos. Quando a implementação é bem feita, algumas vantagens adicionais podem ser obtidas, tais como:

- Abstração completa do mecanismo de persistência: a aplicação em si não precisa conhecer absolutamente nada sobre o mecanismo de persistência, bastando conhecer somente a interface externa do DAO.
- Troca de mecanismo de persistência: caso o mecanismo de persistência deva ser trocado, basta trocar o DAO e todo o resto poderia continuar funcionando perfeitamente.
- Coesão: uma camada DAO bem implementada concentra TODAS as funções de persistência e configuração de banco, facilitando alterações futuras.

A Figura 17 ilustra a arquitetura de uma camada de mapeamento objeto relacional. Perceba que temos essencialmente três partes essenciais. O **storage** representa o banco de dados relacional, com suas tabelas e registros individuais. A **data access layer** é a camada de mapeamento entre os dois mundos. Atente para o fato de que existe uma classe DAO que é responsável pelas informações de conexão com o banco de dados e pela operação de buscar um estudante. Finalmente, a **application** é responsável por conter as classes que representam o domínio da área de negócios da aplicação. Através das setas podemos inferir que a camada de mapeamento atua nos dois sentidos das operações, convertendo objetos para registros e vice-versa.

FIGURA 64 – ARQUITETURA DE MAPEAMENTO OBJETO RELACIONAL DO TIPO DAO



FONTE: Cardim (2010)

4 JDBC

Conforme JavaFree (2014), a Java Database Connectivity ou JDBC é um conjunto de classes e interfaces escritas em Java que faz o envio de instruções SQL para qualquer banco de dados relacional. Esta api possibilita o uso de banco de dados já instalados. Para cada banco de dados há um driver JDBC que pode cair em uma de quatro categorias:

1) Ponte JDBC-ODBC: É o tipo mais simples, mas restrito à plataforma Windows. Utiliza ODBC para conectar-se com o banco de dados, convertendo métodos JDBC em chamadas às funções do ODBC. Esta ponte é normalmente usada quando não há um driver puro-Java (tipo 4) para determinado banco de dados, pois seu uso é desencorajado devido à dependência de plataforma;

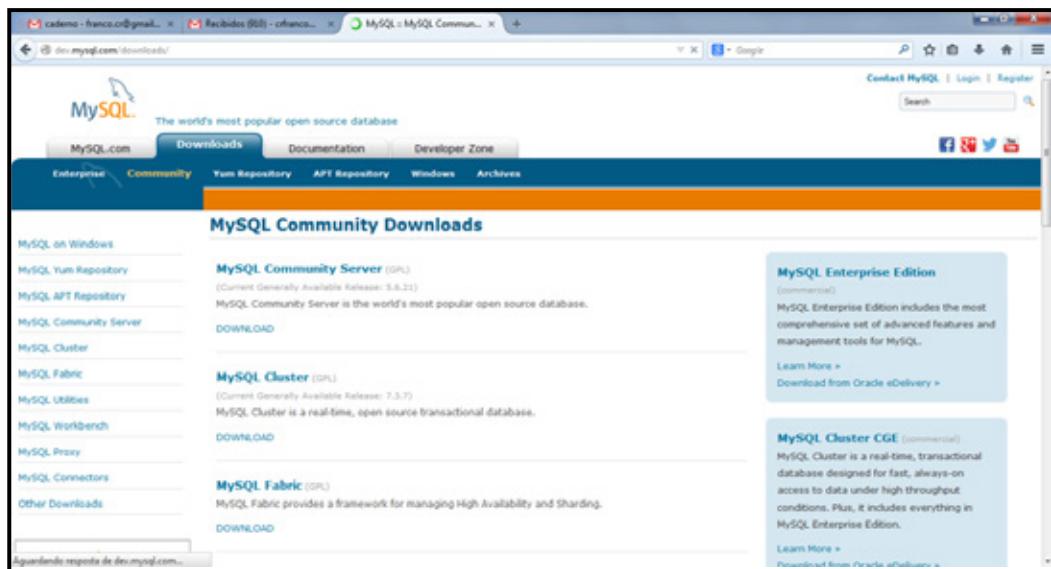
Driver API-Nativo: O driver API-Nativo traduz as chamadas JDBC para as chamadas da API cliente do banco de dados usado. Como a Ponte JDBC-ODBC, pode precisar de *software* extra instalado na máquina cliente.

Driver de Protocolo de Rede: Traduz a chamada JDBC para um protocolo de rede independente do banco de dados utilizado, que é traduzido para o protocolo do banco de dados por um servidor. Por utilizar um protocolo independente, pode conectar as aplicações clientes Java a vários bancos de dados diferentes. É o modelo mais flexível.

Driver nativo: Converte as chamadas JDBC diretamente no protocolo do banco de dados. Implementado em Java, normalmente é independente de plataforma e escrito pelos próprios desenvolvedores. É o tipo mais recomendado para ser usado.

Para nossas implementações práticas utilizaremos o banco de dados MySQL, que pode ser obtido gratuitamente em <<http://www.mysql.com>> (Figura 65). A conexão da linguagem de programação Java com o banco de dados será feita através de um driver JDBC do tipo 4, que pode ser obtido também de forma gratuita em <dev.mysql.com/downloads/connector/j/> (Figura 66).

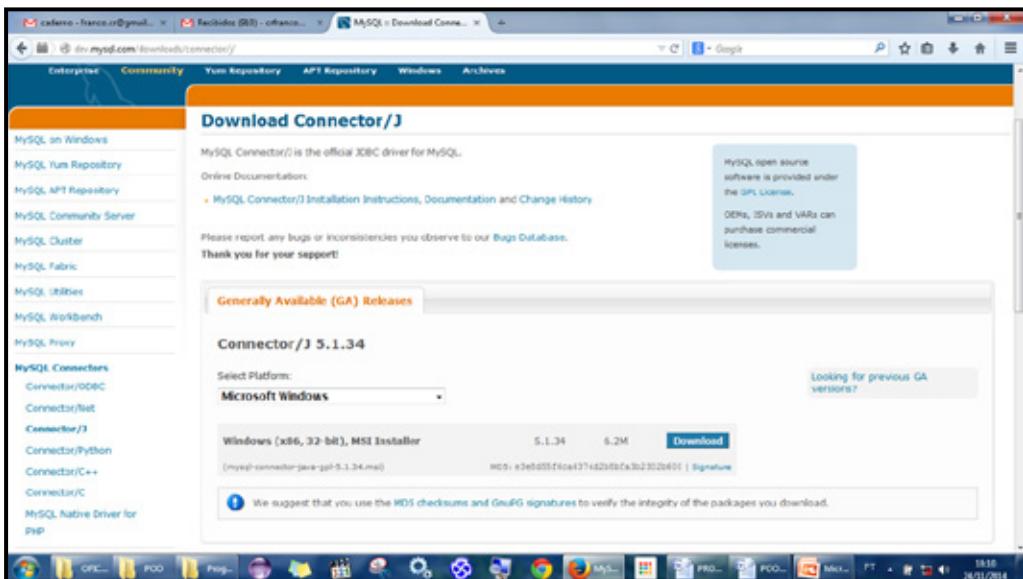
FIGURA 65 – SITE DO MYSQL



FONTE: O autor

Caso você se sinta mais à vontade utilizando outro banco de dados, vá em frente. A única restrição existente é que o banco de dados deve possuir um driver JDBC do tipo 4. Como a maioria das empresas que desenvolvem bancos de dados quer garantir a compatibilidade de seus produtos com a linguagem de programação Java, é bem provável que você consiga encontrar um driver compatível.

FIGURA 66 – DRIVER JDBC PARA MYSQL



FONTE: O autor



Os procedimentos de instalação e configuração do MYSQL não fazem parte do escopo deste caderno, portanto não serão demonstrados. Em caso de dúvidas sugerimos consultar a documentação no próprio site do produto em <<http://www.mysql.com>>.

5 IMPLEMENTAÇÃO DO DATA ACCESS OBJECT (DAO)

No sentido de organizar os projetos e códigos no Eclipse, criaremos um projeto chamado PERSISTENCIA. Dentro deste projeto haverá dois pacotes, conforme Figura 67:

- domain: conterá as classes que representam o domínio da aplicação;
- dataservices: conterá pacotes e classes que farão a conexão com o banco de dados e também as implementações dos DAOs propriamente ditos.

FIGURA 67 – PACOTES DO PROJETO



FONTE: O autor

A classe candidata para o mapeamento chama-se Livro, sendo que seus atributos e construtores estão listados no Quadro 93. A criação de um construtor vazio e de getters e setters para todos os atributos (omitidos por questão de espaço) auxilia na implementação do DAO, conforme veremos posteriormente. Esta classe será colocada no package domain.

QUADRO 91 – CLASSE LIVRO

```

9 public class Livro {
10     private long id;
11     private String titulo;
12     private String autor;
13     private int paginas;
14     private String editora;
15     private String isbn;
16     private int avaliacao;
17
18     public Livro() {}
19
20     public Livro(long id, String titulo, String autor, int paginas,
21                 String editora, String isbn, int avaliacao) {
22         this.id = id;
23         this.titulo = titulo;
24         this.autor = autor;
25         this.paginas = paginas;
26         this.editora = editora;
27         this.isbn = isbn;
28         this.avaliacao = avaliacao;
29     }

```

FONTE: O autor

Antes de procedermos com a implementação da classe DAO, é necessário criarmos uma classe que estabeleça a conexão com o banco de dados propriamente dito. Nesta classe ficam os parâmetros de configuração do banco, o usuário e senha que acessará o banco, a base de dados onde faremos as operações etc. Nossa classe será implementada de forma a permitir sua reutilização por todos os DAOs da aplicação que fizerem acesso àquela base de dados específica. O código fonte da classe Conexao é mostrado no Quadro 94.

QUADRO 92 – CLASSE CONEXAO

```

5 public class Conexao {
6     private Connection conexao;
7     private final String URL = "jdbc:mysql://localhost:3306/java";
8     private final String USER = "root"; private final String PASSWORD = "root";
9     private final String TPCONEXA0 = "com.mysql.jdbc.Driver";
10
11    public Connection abrirConexao() {
12        try {
13            Class.forName(TPCONEXA0);
14            conexao = DriverManager.getConnection(URL, USER, PASSWORD);
15        } catch (ClassNotFoundException | SQLException ex) {
16            ex.printStackTrace();
17        }
18        return conexao;
19    }
20
21    public void fecharConexao() {
22        if (conexao != null) {
23            try {
24                conexao.close();
25            } catch (SQLException e) {
26                e.printStackTrace();
27            }
28        }
29    }

```

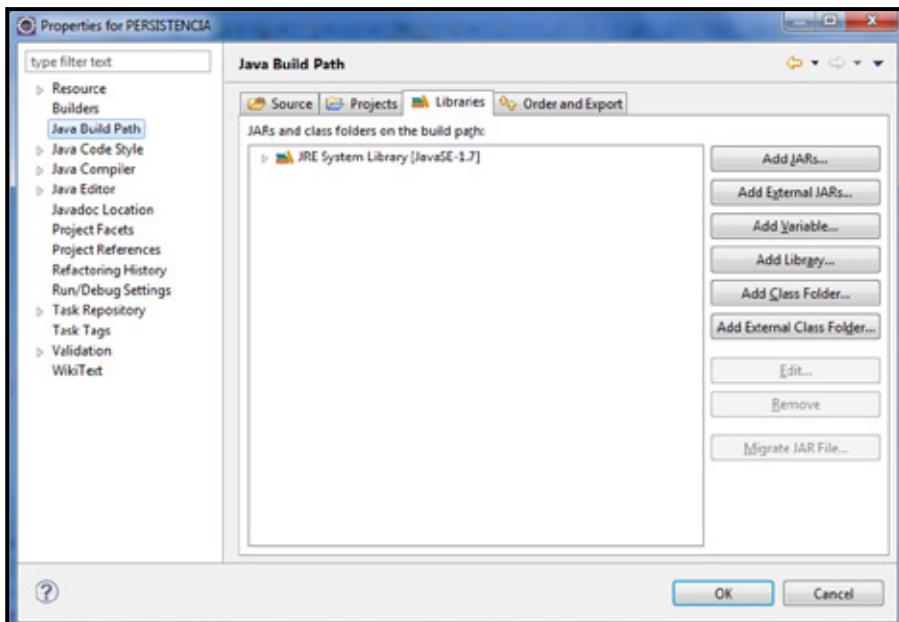
FONTE: O autor

Algumas particularidades da implementação merecem destaque. O atributo conexão armazena o objeto de conexão com o banco de dados. Poderíamos tê-lo utilizado diretamente, entretanto, a criação de uma classe contendo este objeto como atributo nos fornece mais flexibilidade. O atributo URL diz respeito às informações de localização do banco de dados. Neste caso podemos inferir que a conexão será feita através de jdbc, com um banco MySql que executa na máquina local na porta 3306. Caso seu banco de dados esteja em outro servidor, basta colocar o endereço IP do mesmo. A palavra java na String diz respeito à base de dados (Schema) onde nossas tabelas estarão. A linha 8 traz as informações de login e senha e a linha 9 referencia o driver que baixamos do site do mysql.

O restante da classe refere-se a métodos de abertura e fechamento da conexão com o banco de dados. O fechamento da conexão é um aspecto extremamente importante do acesso a qualquer banco de dados, pois caso não a fechemos continuaremos a consumir recursos do sistema operacional e do servidor. Esta classe será colocada no pacote dataservices.conexao.

Um detalhe importante sobre nosso projeto refere-se à utilização do driver JDBC. Fizemos o *download* do mesmo, mas perceba que ainda não o referenciamos. Para referenciá-lo, devemos clicar com o botão direito no projeto PERSISTENCIA e selecionar a opção Properties. Selecionando esta opção, a tela mostrada na Figura 68 aparecerá. Nesta tela, você deverá clicar na opção destacada em azul na figura (Java Build Path) e clicar no botão Add External JARs. Em seguida basta selecionar o arquivo “mysql-connector-java-x.zip”, lembrando que o x é referente ao número da versão e pode variar de acordo com o arquivo que foi obtido anteriormente. Para encerrar, basta clicar no botão OK até que a janela de Properties feche.

FIGURA 68 – REFERÊNCIA AO DRIVER JDBC

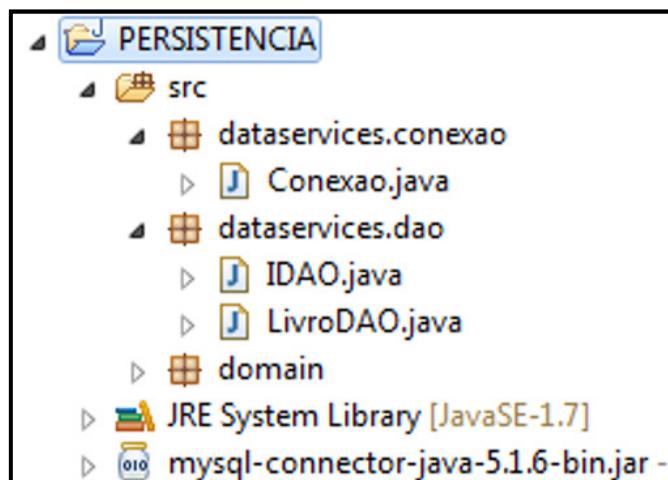


FONTE: O autor

A partir de agora você pode utilizar livremente o driver JDBC do MySQL no projeto e é possível visualizá-lo (Figura 22).

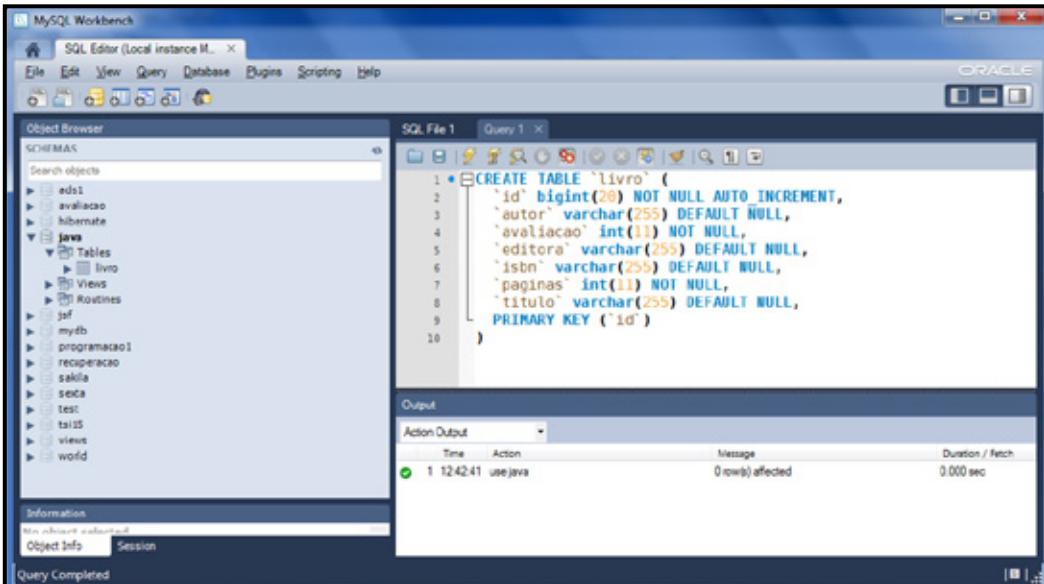
Precisamos ainda criar a tabela que receberá as informações da classe Livro no banco de dados. Neste caso específico, as estruturas da tabela e da classe são praticamente idênticas, embora isso não seja mandatório em todos os casos. A Figura 70 mostra a interface do editor do MySQL e o código para criação da tabela.

FIGURA 69 – DRIVER JDBC ADICIONADO AO PROJETO



FONTE: O autor

FIGURA 70 – CRIAÇÃO DA TABELA



FONTE: O autor



Em nosso exemplo, utilizamos a ferramenta MySQL Workbench para a criação da tabela. Caso você prefira pode utilizar qualquer ferramenta, inclusive o console do prompt de comando, visto que isso não interferirá na criação da tabela e posterior utilização pela aplicação Java.

Uma vez criada a infraestrutura da aplicação e a tabela propriamente dita, podemos implementar a classe que fará o mapeamento propriamente dito, chamada de LivroDAO. Existem estratégias para evitar que cada entidade mapeada tenha um DAO específico, através de generics e reflection, mas como nosso intuito é simplesmente introduzir você ao conceito, mapearemos uma classe para uma tabela, praticamente sem criar estruturas genéricas.

Como primeiro passo criaremos uma interface chamada de IDAO, que será implementada por todas as entidades que desejarem ser mapeadas para o banco. O objetivo desta interface é padronizar as operações que cada entidade realizará. O código fonte está listado no quadro 37. Note que fazemos uso do polimorfismo paramétrico, pois a classe recebe um dado do tipo T em sua construção. A mesma entidade T será utilizada em todos os métodos da interface. As operações básicas de qualquer classe que implementar essa interface deverão incluir:

- Salvar um objeto do tipo T.
- Obter um objeto do tipo T por seu id.
- Obter uma lista contendo todos os objetos T.
- Remover um objeto do tipo T do banco de dados.
- Atualizar o estado de um objeto T no banco de dados.

QUADRO 93 – INTERFACE IDAO

```

13 public interface IDAO<T> {
14
15     public void save(T entity);
16
17     public T getLivro(long id);
18
19     public List<T> list();
20
21     public void remove(T entity);
22
23     public void update(T entity);
24 }
```

FONTE: O autor

Na nossa classe LivroDAO, o T será substituído pela classe Livro e as operações encapsularão chamadas ao JDBC com instruções enviadas ao banco de dados MYSQL. Esta classe isola a complexidade de acesso ao banco, fornecendo simplesmente os 5 métodos como interface externa para o resto da aplicação. Caso se resolva trocar o mecanismo de persistência para outro banco relacional ou mesmo para outro tipo de banco de dados, podemos trocar a implementação de LivroDAO sem afetar o resto.

Nossa classe é relativamente extensa, então faremos a listagem da mesma em vários quadros, cada um para um método distinto. No quadro 94 mostramos a assinatura da classe e o método construtor. Na assinatura trazemos a implementação da interface IDAO parametrizada pelo Livro. Os dois atributos referem-se aos objetos de conexão com o banco de dados. A criação de uma classe responsável unicamente por lidar com a conexão e até mesmo a abertura da conexão automaticamente no construtor da LivroDAO é opcional.



Atualmente a recomendação para persistência de dados na Linguagem de Programação Java é conhecida como Java Persistence API (JPA). Atualmente em sua versão 2.1, a JPA possui diversas implementações no mercado, sendo que a mais utilizada chama-se Hibernate. Maiores detalhes sobre a JPA 2.1 podem ser encontrados na Leitura Complementar desta unidade.

QUADRO 94 – CLASSE LIVRODAO

```

24 public class LivroDAO implements IDAO<Livro> {
25
26     private Connection connection;
27     private Conexao conexao;
28
29     public LivroDAO() {
30         conexao = new Conexao();
31         connection = conexao.abrirConexao();
32     }
33

```

FONTE: O autor

Nossa classe é relativamente extensa, então faremos a listagem em vários quadros, cada um para um método distinto. No Quadro 38 apontamos a assinatura da classe e o método construtor. Na assinatura trazemos a implementação da interface IDAO parametrizada pelo Livro. Os dois atributos referem-se aos objetos de conexão com o banco de dados. A criação de uma classe responsável unicamente por lidar com a conexão e até mesmo a abertura da conexão automaticamente no construtor da LivroDAO é opcional.

No Quadro 39 é mostrada a implementação do método save, responsável pela inserção dos objetos no banco de dados. A instrução para inserção dos registros no banco de dados está na linha 37 e é armazenada em uma variável chamada SQL. Note que o método recebe um objeto livro e desmonta este objeto, inserindo os valores dos atributos como parâmetros da instrução. Na linha 50 é capturada uma SQLException, o que sempre ocorre quando fazemos uso do JDBC.

QUADRO 95 – MÉTODO SAVE

```

34@ Override
35 public void save(Livro livro) {
36     // instrucao sql
37     String sql = "Insert into Livro (autor, avaliacao, editora, isbn, "
38         + "paginas, titulo) values(?, ?, ?, ?, ?, ?)";
39     PreparedStatement pstmt;
40     try {
41         pstmt = connection.prepareStatement(sql);
42         // setar os parametros
43         pstmt.setString(1, livro.getAutor());
44         pstmt.setInt(2, livro.getAvaliacao());
45         pstmt.setString(3, livro.getEditora());
46         pstmt.setString(4, livro.getIsbn());
47         pstmt.setInt(5, livro.getPaginas());
48         pstmt.setString(6, livro.getTitulo());
49         pstmt.execute();
50     } catch (SQLException e) {
51         e.printStackTrace();
52     }
53 }

```

FONTE: O autor

No Quadro 40 trazemos a implementação do método update, que funciona de maneira bastante semelhante ao save.

QUADRO 96 – MÉTODO UPDATE

```

114@ Override
115 public void update(Livro livro) {
116     try {
117         String sql = "update Livro "
118             + "set autor = ?, avaliacao = ?," 
119             + " editora = ?, isbn = ?, paginas=? , titulo=? where id = ?";
120         PreparedStatement pstmt;
121         pstmt = connection.prepareStatement(sql);
122         // setar os parametros
123         pstmt.setString(1, livro.getAutor());
124         pstmt.setInt(2, livro.getAvaliacao());
125         pstmt.setString(3, livro.getEditora());
126         pstmt.setString(4, livro.getIsbn());
127         pstmt.setInt(5, livro.getPaginas());
128         pstmt.setString(6, livro.getTitulo());
129         pstmt.setLong(7, livro.getId());
130         pstmt.execute();
131     } catch (SQLException ex) {
132         ex.printStackTrace();
133     }
134 }

```

FONTE: O autor

No Quadro 41 é listado o código fonte do método getLivro, responsável por retornar um objeto do tipo Livro para a aplicação após buscá-lo no banco de dados através de seu ID. Lembre-se de que cada Livro possui um Id e que este Id é único, o que garante que 0 ou 1 objetos serão retornados pelo método. O laço criado no comando while (linha 64) varrerá os registros encontrados pela instrução e criará um objeto com os valores dos campos, atribuindo-os ao objeto. Caso não existam registros, o objeto p será retornado com o valor null.

QUADRO 97 – MÉTODO GETLIVRO

```

55@Override
56 public Livro getLivro(long id) {
57     Livro p = null;
58     try {
59         Statement sta = connection.createStatement();
60         ResultSet elements = sta
61             .executeQuery("select * from livro where id = " + id);
62         while (elements.next()) {
63             p = new Livro(elements.getLong("id"),
64                           elements.getString("titulo"),
65                           elements.getString("autor"),
66                           elements.getInt("paginas"),
67                           elements.getString("editora"),
68                           elements.getString("isbn"),
69                           elements.getInt("avaliacao"));
70         }
71     } catch (SQLException ex) {
72         ex.printStackTrace();
73     }
74     return p;
75 }
76 }
```

FONTE: O autor

O Quadro 42 traz o método list, responsável por buscar todos os livros existentes na tabela do banco de dados. Sua implementação é bastante semelhante à do método getLivro, excetuando-se a criação e população da lista (linhas 80 e 92, respectivamente). Neste método, caso nenhum registro seja encontrado, a lista é retornada sem nenhum objeto dentro.

QUADRO 98 – MÉTODO LIST

```

78@Override
79 public List<Livro> list() {
80     List<Livro> result = new ArrayList<Livro>();
81     try {
82         Statement sta = connection.createStatement();
83         ResultSet elements = sta.executeQuery("SELECT * from Livro");
84         while (elements.next()) {
85             Livro l = new Livro();
86             l.setId(elements.getInt("id")); l.setAutor(elements.getString("autor"));
87             l.setAvaliacao(elements.getInt("avaliacao"));
88             l.setEditora(elements.getString("editora"));
89             l.setPaginas(elements.getInt("paginas"));
90             l.setIsbn(elements.getString("ISBN"));
91             l.setTitulo(elements.getString("titulo"));
92             result.add(l);
93         }
94     } catch (SQLException ex) {
95         ex.printStackTrace();
96     }
97     return result;
98 }
```

FONTE: O autor

Finalmente, o Quadro 43 mostra o método remove, responsável por excluir um livro do banco de dados através de seu id. Note que apesar de enviarmos um objeto livro inteiro para o método, extraímos o id deste objeto e o atribuímos como parâmetro da instrução SQL.

QUADRO 99 – MÉTODO REMOVE

```

100+     @Override
101      public void remove(Livro livro) {
102          String sql = "delete from livro where id = ?";
103          PreparedStatement pstmt;
104          try {
105              pstmt = connection.prepareStatement(sql);
106              pstmt.setLong(1, livro.getId());
107              pstmt.execute();
108          } catch (SQLException e) {
109              e.printStackTrace();
110          }
111      }
112  }
```

FONTE: O autor

O mais importante destes exemplos é perceber que criamos uma camada onde encapsulamos toda a complexidade do acesso aos dados via SQL, isolando-a do resto da aplicação. Essa arquitetura auxilia a reforçar o conceito de coesão, pois tratamos o mecanismo de persistência exatamente como ele deve ser tratado: um repositório para armazenamento e recuperação de dados. Com isso, podemos nos concentrar no mais importante da aplicação, o domínio de negócio para o qual o sistema será desenvolvido.

Nossa aplicação está pronta para o teste. Inicialmente criaremos dois objetos do tipo Livro e faremos uso do método save para os inserirmos no banco de dados (linhas 16 e 22 do Quadro 44). Para comprovar que os objetos foram inseridos, fazemos uma busca no banco de dados onde podemos perceber os dois registros (Figura 24).

QUADRO 100 – INSERÇÃO DOS REGISTROS

```

5 public class Testador {
6
7 	/*
8   * @param args
9  */
10 public static void main(String[] args) {
11     Livro l = new Livro();
12     l.setAutor("Joshua Bloch"); l.setTitulo("Java Efetivo 2.0");
13     l.setEditora("Alta books"); l.setPaginas(300);
14     l.setIsbn("8373635353"); l.setAvaliacao(5);
15     LivroDAO dao = new LivroDAO();
16     dao.save(l);
17
18     Livro l2 = new Livro();
19     l2.setAutor("Kathy Sierra"); l2.setTitulo("Use a cabeça: Java");
20     l2.setEditora("Alta books"); l2.setPaginas(500);
21     l2.setIsbn("8322635353"); l2.setAvaliacao(5);
22     dao.save(l2);
23
24 }
25 }
```

FONTE: O autor

FIGURA 71 – REGISTROS INSERIDOS NO BANCO DE DADOS

	id	autor	avaliacao	editora	isbn	paginas	titulo
▶	25	Joshua Bloch	5	Alta books	8373635353	300	Java Efetivo 2.0
*	26	Kathy Sierra	5	Alta books	8322635353	500	Use a cabeça: Java
*		NULL	NULL	NULL	NULL	NULL	NULL

FONTE: O autor

No Quadro 101 testaremos as demais operações do DAO e na Figura 72 mostraremos a respectiva saída no console. Na linha 10 buscamos todos os objetos existentes na tabela Livro, imprimindo-os na saída padrão. Entre as linhas 12 a 14, buscamos um objeto por seu Id, alteramos seu número de páginas e o atualizamos no banco de dados. Na linha 18 fazemos a remoção deste mesmo objeto e para provar que o mesmo não existe mais, imprimimos novamente o resultado de um método list na linha 20. Finalmente, como encerramos as operações com o banco de dados, na linha 22 fazemos o fechamento da conexão.

QUADRO 101 – TESTE DAS DEMAIS FUNÇÕES DO DAO

```

5 public class Testador {
6
7     public static void main(String[] args) {
8         LivroDAO dao = new LivroDAO();
9
10        System.out.println(dao.list());
11
12        Livro l = dao.getLivro(25);
13        l.setPaginas(456);
14        dao.update(l);
15
16        System.out.println(dao.list());
17
18        dao.remove(l);
19
20        System.out.println(dao.list());
21
22        dao.getConexao().fecharConexao();
23    }
24 }
```

FONTE: O autor

FIGURA 72 – SAÍDA DO CONSOLE

The screenshot shows a Java IDE's console window. At the top, there are several tabs: Markers, Properties, Servers, Data Source Expl..., Snippets, and Console. Below the tabs, the console area displays three lines of text, each representing a list of two book objects. The first line shows 'Java Efetivo 2.0' and 'Use a cabeça: Java'. The second line shows the same two titles again. The third line shows only 'Use a cabeça: Java'. This indicates that the code has successfully updated the database and removed one of the books.

FONTE: O autor

Atualmente existe uma nova especificação dentro da plataforma Java para persistência de objetos e mapeamento objeto relacional, conhecida como Java Persistence API (JPA). A JPA é a recomendação da Oracle para as operações de persistência em bancos de dados relacionais através do Java, pois automatiza muito do que tivemos que fazer manualmente em nossos exemplos deste tópico. A leitura complementar desta unidade fala exclusivamente sobre JPA e uma de suas implementações mais usadas no mercado, conhecida como Hibernate.

LEITURA COMPLEMENTAR

Uma introdução prática ao JPA com Hibernate. Disponível originalmente em: <<http://www.caelum.com.br/apostila-java-web/uma-introducao-pratica-ao-jpa-com-hibernate/>>. Adaptado livremente pelo autor.

Mapeamento Objeto Relacional

Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor era gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas.

Além de um problema de produtividade, algumas outras preocupações aparecem: SQL que, apesar de ter um padrão ANSI, apresenta diferenças significativas dependendo do fabricante. Não é simples trocar um banco de dados pelo outro.

Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e precisamos pensar das duas maneiras para fazer um único sistema. Para representarmos as informações no banco, utilizamos tabelas e colunas. As tabelas geralmente possuem chave primária (PK) e podem ser relacionadas por meio da criação de chaves estrangeiras (FK) em outras tabelas.

Quando trabalhamos com uma aplicação Java, seguimos o paradigma orientado a objetos, onde representamos nossas informações por meio de classes e atributos. Além disso, podemos utilizar também herança, composição para relacionar atributos, polimorfismo, enumerações, entre outros. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos "transformar" objetos em registros e registros em objetos.

Java Persistence API e Frameworks ORM

Ferramentas para auxiliar nesta tarefa tornaram-se popular entre os desenvolvedores Java e são conhecidas como ferramentas de mapeamento objeto-relacional (ORM). O Hibernate é uma ferramenta ORM open source e é a líder de mercado, sendo a inspiração para a especificação Java Persistence API(JPA). O Hibernate nasceu sem JPA mas hoje em dia é comum acessar o Hibernate pela especificação JPA. Como toda especificação, ela deve possuir implementações. Entre as implementações mais comuns, podemos citar: Hibernate da JBoss, EclipseLink da Eclipse Foundation e o OpenJPA da Apache. Apesar do Hibernate ter originado a JPA, o EclipseLink é a implementação referencial.

O Hibernate abstrai o seu código SQL, toda a camada JDBC e o SQL será gerado em tempo de execução. Mais que isso, ele vai gerar o SQL que serve para um determinado banco de dados, já que cada banco fala um "dialeto" diferente dessa linguagem. Assim há também a possibilidade de trocar de banco de dados sem ter de alterar código Java, já que isso fica de responsabilidade da ferramenta.

Como usaremos JPA abstraímos mais ainda, podemos desenvolver sem conhecer detalhes sobre o Hibernate e até trocar o Hibernate com uma outra implementação como OpenJPA.

Bibliotecas do Hibernate e JPA

Vamos usar o JPA com Hibernate, ou seja, precisamos baixar os JARs no site do Hibernate. O site oficial do Hibernate é o www.hibernate.org, onde você baixa a última versão na seção ORM e Download.

Com o ZIP baixado em mãos, vamos descompactar o arquivo. Dessa pasta vamos usar todos os JARs obrigatórios (required). Não podemos esquecer o JAR da especificação JPA que se encontra na pasta jpa.

Para usar o Hibernate e JPA no seu projeto é necessário colocar todos esses JARs no classpath.

O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos o arquivo .jar correspondente ao driver JDBC.

Para aprender a configurar estas bibliotecas de forma correta, sugerimos a leitura do material disponível em: <<http://blog.caelum.com.br/as-dependencias-do-hibernate-3-5/>>.

Mapeando uma classe Tarefa para nosso Banco de Dados

Para este capítulo, utilizaremos a classe que representa uma tarefa:

```
package br.com.caelum.tarefas.modelo;

public class Tarefa{
    private Long id;
    private String descricao;
    private boolean finalizado;
    private Calendar dataFinalizacao;
}
```

Criamos os getters e setters para manipular o objeto, mas fique atento que só devemos usar esses métodos se realmente houver necessidade.

Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Precisamos configurar o Hibernate para que ele saiba da existência dessa classe e, desta forma, saiba que deve inserir uma linha na tabela Tarefa toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em **mapear** uma classe a tabela.

Para mapear a classe Tarefa, basta adicionar algumas poucas **anotações** em nosso código. Anotação é um recurso do Java que permite inserir **metadados** em relação a nossa classe, atributos e métodos. Essas anotações depois poderão ser lidas por frameworks e bibliotecas, para que eles tomem decisões baseadas nessas pequenas configurações.

Para essa nossa classe em particular, precisamos de apenas quatro anotações:

```
@Entity
public class Tarefa{

    @Id
    @GeneratedValue
    private Long id;

    private String descricao;
    private boolean finalizado;

    @Temporal(TemporalType.DATE)
    private Calendar dataFinalizacao;

    // métodos...
}
```

`@Entity` indica que objetos dessa classe se tornem "persistível" no banco de dados. `@Id` indica que o atributo `id` é nossa chave primária (você precisa ter uma chave primária em toda entidade) e `@GeneratedValue` diz que queremos que esta chave seja populada pelo banco (isto é, que seja usado um auto increment ou sequence, dependendo do banco de dados). Com `@Temporal` configuramos como mapear um `Calendar` para o banco, aqui usamos apenas a data (sem hora), mas poderíamos ter usado apenas a hora (`TemporalType.TIME`) ou timestamp (`TemporalType.TIMESTAMP`). Essas anotações precisam dos devidos imports, e pertencem ao pacote `javax.persistence`.

Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe Tarefa será gravada na tabela de nome também Tarefa, e o atributo `descricao` em uma coluna de nome `descricao` também!

Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo dataFinalizacao numa coluna chamada data_finalizado faríamos:

```
@Column(name ="data_finalizado", nullable =true)
private Calendar dataFinalizacao;
Para usar uma tabela com o nome tarefas:
@Entity
@Table(name="tarefas")
public class Tarefa{
```

Repare que nas entidades há todas as informações sobre as tabelas. Baseado nelas podemos até pedir gerar as tabelas no banco, mas para isso é preciso configurar o JPA.

CONFIGURANDO o JPA COM AS PROPRIEDADES DO BANCO

Em qual banco de dados vamos gravar nossas Tarefas? Qual é o login? Qual é a senha? O JPA necessita dessas configurações, e para isso criaremos o arquivo persistence.xml.

Alguns dados que vão nesse arquivo são específicos do Hibernate e podem ser bem avançados, sobre controle de cache, transações, connection pool etc.

Para nosso sistema, precisamos de quatro linhas com configurações que já conhecemos do JDBC: string de conexão com o banco, o driver, o usuário e senha. Além dessas quatro configurações, precisamos dizer qual dialeto de SQL deverá ser usado no momento que as queries são geradas; no nosso caso, MySQL. Vamos também mostrar o SQL no console para acompanhar o trabalho do Hibernate.

Vamos também configurar a criação das tabelas baseado nas entidades.

Segue uma configuração completa que define uma unidade de persistência (*persistence-unit*) com o nome *tarefas*, seguidos pela definição do provedor, entidades e properties:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

<persistence-unit name="tarefas">
```

```

<!-- provedor/implementacao do JPA -->
<provider>org.hibernate.ejb.HibernatePersistence</provider>

<!-- entidade mapeada -->
<class>br.com.caelum.tarefas.modelo.Tarefa</class>

<properties>
<!-- dados da conexao -->
<propertyname="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
<propertyname="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost/fj21"/>
<propertyname="javax.persistence.jdbc.user"
value="root"/>
<propertyname="javax.persistence.jdbc.password"
value="" />

<!-- propriedades do hibernate -->
<propertyname="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
<propertyname="hibernate.show_sql" value="true"/>
<propertyname="hibernate.format_sql" value="true"/>

<!-- atualiza o banco, gera as tabelas se for preciso -->
<propertyname="hibernate.hbm2ddl.auto" value="update"/>

</properties>
</persistence-unit>
</persistence>

```

É importante saber que o arquivo persistence.xml deve ficar na pasta **META-INF** do seu *classpath*.

USANDO O JPA

Para usar o JPA no nosso código Java, precisamos utilizar sua API. Ela é bastante simples e direta e rapidamente você estará habituado com suas principais classes.

Nosso primeiro passo é fazer com que o JPA leia a nossa configuração: tanto o nosso arquivo persistence.xml quanto as anotações que colocamos na nossa entidade Tarefa. Para tal, usaremos a classe com o mesmo nome do arquivo XML:Persistence. Ela é responsável de carregar o XML e inicializar as configurações. Resultado dessa configuração é uma EntityManagerFactory:

```
EntityManagerFactoryfactory = Persistence.  
createEntityManagerFactory("tarefas");
```

Estamos prontos para usar o JPA. Antes de gravar uma Tarefa, precisamos que exista a tabela correspondente no nosso banco de dados. Em vez de criarmos o script que define o *schema* (ou DDL de um banco, *data definition language*) do nosso banco (os famosos CREATE TABLE) podemos deixar isto a cargo do próprio Hibernate. Ao inicializar a EntityManagerFactory também já será gerada uma tabela Tarefas pois configuramos que o banco deve ser atualizada pela propriedade do Hibernate: hbm2ddl.auto.

TRABALHANDO COM OS OBJETOS: O ENTITYMANAGER

Para se comunicar com o JPA, precisamos de uma instância de um objeto do tipo EntityManager. Adquirimos uma EntityManager através da fábrica já conhecida: EntityManagerFactory.

```
EntityManagerFactory factory = Persistence.  
createEntityManagerFactory("tarefas");  
EntityManager manager = factory.createEntityManager();  
manager.close();  
factory.close();
```

Persistindo novos objetos

Através de um objeto do tipo EntityManager, é possível gravar novos objetos no banco. Para isto, basta utilizar o método persist dentro de uma transação:

```
Tarefa tarefa =new Tarefa();
tarefa.setDescricao("Estudar JPA");
tarefa.setFinalizado(true);
tarefa.setDataFinalizacao(Calendar.getInstance());

EntityManagerFactory factory = Persistence.
createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();
manager.persist(tarefa);
manager.getTransaction().commit();

System.out.println("ID da tarefa: "+ tarefa.getId());
manager.close();
```

Este é apenas o começo. O JPA com Hibernate é muito customizável: podemos configurá-lo para que gere as queries de acordo com dicas nossas, dessa forma otimizando casos particulares em que as queries que ele gera por padrão não são desejáveis.

Uma confusão que pode ser feita à primeira vista é pensar que o JPA com Hibernate é lento, pois, ele precisa gerar as nossas queries, ler objetos e suas anotações e assim por diante. Na verdade, o Hibernate faz uma série de otimizações internamente que fazem com que o impacto dessas tarefas seja próximo a nada. Portanto, o Hibernate é, sim, performático, e hoje em dia pode ser utilizado em qualquer projeto que se trabalha com banco de dados.

RESUMO DO TÓPICO 3

- A api dentro da linguagem de programação Java para se trabalhar com bancos de dados relacionais chama-se Java Database Connectivity ou JDBC.
- Existe um problema quando utilizamos bancos de dados relacionais como repositórios de dados para sistemas orientados a objetos, conhecidos como diferença de impedância.
- Este problema ocorre porque os bancos de dados relacionais e os programas orientados a objetos representam a informação de forma diferente. As soluções para resolver este problema trabalham com o conceito de mapeamento objeto relacional, onde objetos são mapeados para registros de tabelas relacionais e vice versa.
- O DAO (data access object) é um padrão de projeto utilizado para resolver o problema de mapeamento objeto relacional. O conceito do DAO é simples e se baseia na implementação de uma camada de isolamento entre a aplicação e o banco de dados relacional.
- Além do DAO, os padrões de projeto Active Record e Repository também atuam no domínio de mapeamento objeto relacional.
- Atualmente, a linguagem de programação Java recomenda a especificação JPA (java persistence api) para atacar o problema de mapeamento objeto relacional.

AUTOATIVIDADE



- 1 Em nosso exemplo de inserção de objetos no banco de dados utilizamos o construtor vazio e setters para atribuir valores para o livro. Por que não utilizamos o construtor completo do objeto?
- 2 Sempre que fazemos uso de um recurso do sistema operacional através de uma conexão, devemos fechar esta conexão assim que terminarmos nossas tarefas. No Quadro 101 fazemos esse fechamento através de um método específico, que inclusive não está implementado na classe LivroDAO. Implemente esse método e altere o exemplo do Quadro 92 para que façamos o fechamento da conexão após inserir os dois livros no banco de dados.
- 3 Na Figura 72 mostramos a impressão dos objetos que retornaram com o método list() da classe LivroDAO. Se você fizer os mesmos testes, perceberá que seu resultado no console é ligeiramente diferente. Descubra o motivo e faça a modificação necessária para que seu exemplo funcione exatamente ao do caderno.

REFERÊNCIAS

BARTH, Carlos. **Trabalhando com interfaces**. 2003. Disponível em: <<http://www.guj.com.br/articles/123>>. Acesso em: 24 ago. 2014.

BATES, Bert; SIERRA, Kathy. **Use a cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2010.

BLACK, Andrew. *Object-oriented programming: some history, and challenges for the next fifty years*, 2012. Disponível em: <<http://web.cecs.pdx.edu/~black/publications/O-JDahl.pdf>>. Acesso em: 18 ago. 2014.

BLOCH, Joshua. **Effective Java**. 2. ed. Addison- Wesley, 2008.

BROOK, Frederick. **No Silver Bullet**: Essence and Accident in Software Engineering. 1986. Disponível em: <<http://worrydream.com/refs/Brooks-NoSilverBullet.pdf>>. Acesso em: 2 maio 2014.

CAELUM. **Java e orientação a objetos**, 2014. Disponível em: <<http://www.caelum.com.br/apostila-java-orientacao-objetos/>>. Acesso em: 18 ago. 2014.

CARDIM, Eder. **Bancos de dados relacionais, orientação a objetos e DBIX**: Class. 2010. Disponível em: <<http://sao-paulo.pm.org/artigo/2010/DBIC>>. Acesso em: 2 maio 2014.

CPAI – Centro de Pesquisa em Arquitetura da Informação. **OpenJDK**, 2011. Disponível em: <<http://www.cpai.unb.br/en/web/cpai/openjdk>>. Acesso em 2 jul. 2014.

CUNHA, Douglas. 2009. **Padrões para atribuição de responsabilidades**. Disponível em: <<http://www.olharcritico.com/engenhariadesoftware/2009/09/grasp-como-atribuir-responsabilidades-com-eficiencia-introducao-padroes-para-atribuicao-de-responsabilidade/>>. Acesso em: 30 jun. 2014.

EVANS, Eric. **Domain-Driven Design**: tackling complexity in the heart of software. Prentice Hall. 2003.

FOWLER, Martin. **ORM Hate**. 2012. Disponível em: <<http://martinfowler.com/bliki/OrmHate.html>>. Acesso em: 8 maio 2014.

HOMEM aranha. Direção: Sam Raimi. Columbia Pictures, 2002. 1 DVD (136 min.)

HUNT, Andrew; THOMAS, David. *The pragmatic programmer: from journeyman to master*. Boston: Addison-Wesley, 1999.

JAVAFREE. JDBC. Disponível em: <<http://javafree.uol.com.br/wiki/jdbc>>. Acesso em: 3 abr. 2014.

MILLER, Jeremy. **Coesão e Acoplamento**. 2008. Disponível em: <<http://msdn.microsoft.com/pt-br/magazine/cc947917.aspx>>. Acesso em: 2 ago. 2014.

O PROGRAMA EM C. **O programa em C**. Disponível em: <http://www.urisan.tche.br/~janob/Cap_1.html>. Acesso em: 8 jun. 2014.

ORACLE, *Your first cup*, 2012. Disponível em: <<http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html>>. Acesso em: 1 maio 2014.

ORACLE, **Your first cup**, 2012. Disponível em: <<http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html>>. Acesso em: 1 maio 2014.

ORACLE. **Java Timeline**, 2011. Disponível em: <<http://oracle.com.edgesuite.net/timeline/java/>>. Acesso em: 3 jun. 2014.

SILVEIRA, Paulo. **Singletons e Static**: Perigo à vista. 2006. Disponível em: <<http://blog.caelum.com.br/singletons-e-static-perigo-a-vista/>>. Acesso em: 21 mar. 2014.

SILVEIRA, Paulo; SILVEIRA, Guilherme; LOPES, Sergio; et. al. **Introdução à arquitetura e design de software**: uma visão sobre a plataforma Java. Rio de Janeiro: Campus, 2011.

SINTES, Tony. **Aprenda programação orientada a objetos em 21 dias**. São Paulo: Pearson Education do Brasil, 2002.

TECH INSIDER. **The Java FAQ**, 1997. Disponível em: <<http://tech-insider.org/java/research/1998/05.html>>. Acesso em: 2 jul. 2014.

TECHOPEDIA. **Platform**, 2010. Disponível em: <<http://www.techopedia.com/definition/3411/platform>>. Acesso em: 14 jun. 2014.

TREKCORE. **Captain Kirk**. Disponível em: <<http://trekcore.com/specials/rarephotos.html>>. Acesso em: 24 ago. 2014.

WIKIPEDIA. **Object-oriented programming**, 2012. Disponível em: <http://en.wikipedia.org/wiki/Object-oriented_programming#History>. Acesso em: 15 ago. 2014.

WIKIPEDIA. **Object-oriented programming**, 2012. Disponível em: <http://en.wikipedia.org/wiki/Object-oriented_programming#History>. Acesso em: 15 ago. 2014.

ZANONI, Dias. **MC336 – Paradigmas de programação**. Disponível em: <<http://www.ic.unicamp.br/~zanoni/mc336/>>. Acesso em: 1 set. 2014.

ANOTAÇÕES