easy java Magazine
Edição 50

Orientação a Objetos: Primeiros passos
Domine os conceitos e melhore o reuso e a manutenção de seu código

Java Collections Framework
Aprenda a utilizar as principais coleções

DEVMEDIA

POO + UML

Interprete e codifique o diagrama de classes

ISSN 2179625-4

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

CERTIFICAÇÕES:

- Oracle Certified Professional, Java SE 6 Programmer
- Oracle Certified Professional, Java EE 5 Web Component Developer

infnet TRAINING

Central de Matrículas: 21 2122-8800 | www.infnet.edu.br

Sumário

Conteúdo sobre Boas Práticas

06 - Orientação a Objetos: Primeiros passos
[*Luís Fernando Orleans*]

Artigo no estilo Curso

16 - Como interpretar Diagramas de Classes da UML – Parte 1
[*Everson Mauda*]

Conteúdo sobre Boas Práticas

24 - Programando com o Java Collections Framework
[*John Soldera*]

Edição 50 • 2015 • ISSN 2173625-4

EXPEDIENTE

Editor

Eduardo Spínola (*eduspinola@gmail.com*)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse **www.devmedia.com.br/central**, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:

EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

summa
Technology + Business
Premium Partner

CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de codigo (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

toolscloud@toolscloud.com

twitter.com/toolscloud

ToolsCloud

toolscloud.com

Orientação a Objetos: Primeiros passosProjeto Orientado a Objetos

Saiba nesse artigo como o entendimento de alguns conceitos básicos pode aumentar o reuso e facilitar a manutenção do código

Quando ouvimos falar em Java como linguagem de programação, a expressão "Orientação a Objetos" logo vem à cabeça. Apesar de ser um dos primeiros temas a estudar quando aprende-se a programar em Java, muitas pessoas o negligenciam achando, por exemplo, que Encapsulamento se reduz a utilizar métodos do tipo **getXXX()** e **setXXX()** em suas classes quando, na verdade, estes tipos de métodos devem ser evitados a qualquer custo justamente por *ferir* este princípio fundamental. Outros exibem orgulhosos a "árvore genealógica" das classes que compõem o seu sistema, sem saber que o esforço para estender e manter o código funcionando corretamente aumenta proporcionalmente com o tamanho da "árvore".

Diante deste cenário, este artigo tem como objetivo principal revisitar os conceitos fundamentais da Orientação a Objetos: Encapsulamento, Herança e Polimorfismo. Ao mesmo tempo, alguns padrões GRASP (*General Responsibility Assignment Software Patterns*) são introduzidos. Estes padrões, ao contrário dos Padrões de Projeto escritos pela "Gangue dos Quatro", possuem o objetivo de facilitar o entendimento sobre quais são as responsabilidades de cada classe em um sistema. Apesar de muitos desenvolvedores considerarem estes padrões mais básicos, sua importância é enorme e não deve ser menosprezada pois, nem mesmo uma abordagem ágil com todas as suas técnicas que preveem suporte a alterações frequentes no código, é capaz de diminuir os impactos negativos de um código mal escrito.

Como exemplo, considere as **Listagens 1**, **2** e **3**, onde são exibidas as classes de um sistema bancário simples que permite a criação de agências. O código foi propositalmente escrito com a classe que representa o domínio (**AgenciaBancaria**) contendo somente métodos **getXXX()** e **setXXX()**, funcionando como um recipiente de dados. O padrão arquitetural Modelo-Visão-Controle (outro conceito mal compreendido por muitos desenvolvedores) foi utilizado como forma de separar as responsabilidades envolvidas em cada classe. A **Figura 1** mostra o diagrama de classes deste sistema.

Fique por dentro

Neste artigo serão apresentados alguns dos erros mais comuns realizados por desenvolvedores Java, principalmente por não compreenderem perfeitamente os fundamentos da própria Orientação a Objetos. Exemplos mostram como identificar trechos de código com alto acoplamento e como isso prejudica a manutenção e a extensão de um sistema OO. Além disso, os propósitos de conceitos como Encapsulamento, Herança e Polimorfismo, bem como o padrão Modelo-Visão-Controle, são revisitados, mostrando como utilizá-los corretamente para deixar o projeto mais rico e facilitar o reuso de classes.

Nota

Durante o desenvolvimento de softwares orientados a objetos, existem alguns problemas que são recorrentes, ou seja, aparecem várias vezes e em diferentes situações, às vezes até mascarados como problemas diferentes. Desta forma, há algum tempo, quatro dos grandes nomes da comunidade de orientação a objetos (Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm) perceberam esse padrão e começaram a catalogar as soluções utilizadas para resolver estes problemas, dando nomes a elas e descrevendo como e quando utilizá-las. Posteriormente, estas soluções foram compartilhadas através do livro Padrões de Projeto: soluções reutilizáveis de softwares orientados a objetos e seus autores passaram a ser conhecidos como a Gangue dos Quatro . O livro é uma referência até hoje para aqueles que querem se aprofundar no estudo da orientação a objetos.

O sistema funciona da seguinte maneira: a classe **VistaAgencia** é instanciada e seu método **exibir()** é chamado. Ele desenha um menu na tela (**Listagem 3**, linhas 8 a 11), onde o usuário digita 1 para cadastrar uma nova agência, 2 para exibir os dados de uma agência específica, ou 3 para sair. Caso opte por cadastrar uma nova agência, um código único é gerado automaticamente e atribuído a ela e sua lista de contas bancárias é criada, estando inicialmente vazia (linhas 14 a 19). Caso opte por exibir os dados de uma agência, o usuário informa em seguida o número da agência e o sistema realiza uma busca. Caso seja encontrada, os dados da agência (código e número de contas bancárias vinculadas) são exibidos na tela (linhas 25 e 26).

Apesar de o código anterior funcionar a contento, ele possui diversos problemas, com apenas alguns deles sendo descritos a seguir:

1. A responsabilidade de gerar o número de uma agência pertence à classe **VistaAgencia**;
2. A classe **VistaAgencia** é responsável, também, por exibir os dados de uma agência;
3. A classe **ControladorAgencia** funciona somente como uma "ponte" entre a visão e o modelo;
4. A classe **VistaAgencia** possui lógica de negócio;
5. A classe **AgenciaBancaria** é utilizada somente como um recipiente de dados.

Como mencionado, estes são apenas alguns dos problemas que podem ser encontrados neste pequeno sistema. Apesar de parecerem muitos, todos eles têm a mesma base: a falta de entendimento correto de conceitos fundamentais da Orientação a Objetos, bem como do padrão Modelo-Visão-Controle (MVC), tornando o código confuso e extremamente difícil de manter.

Para ilustrar essa afirmação, considere que seja adicionada uma nova restrição de negócio, onde uma agência bancária pode estar ativa ou inativa, que pode ser materializada com a criação de um campo *booleano* de nome **ativo** na classe correspondente. Assim, caso o *flag* esteja marcado como falso, a referida agência não deve possuir nenhuma conta, ou seja, sua lista de contas bancárias deve ser vazia.

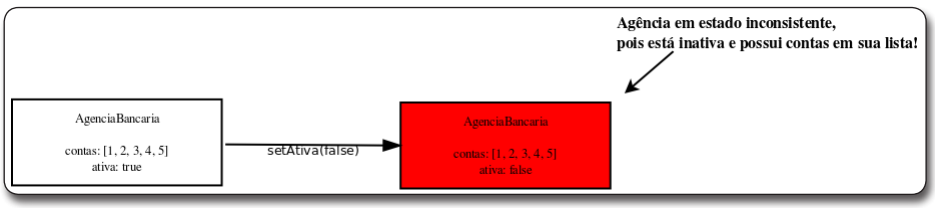

Figura 1. Método setAtiva() pode levar o objeto a um estado inconsistente

Listagem 1. Classe que representa o domínio do problema no sistema bancário.

```
01 public class AgenciaBancaria {
02   private String codigo;
03   private List<ContaBancaria> contas = new LinkedList<>();
04   public List<ContaBancaria> getContas() {
05     return contas;
06   }
07   public void setContas(List<ContaBancaria> contas) {
08     this.contas = contas;
09   }
10   public String getCodigo() {
11     return codigo;
12   }
13   public void setCodigo(String codigo) {
14     this.codigo = codigo;
15   }
16 }
```

Listagem 2. Classe que representa a lógica de negócio no sistema bancário.

```
01 public class ControladorAgencia {
02   private List<AgenciaBancaria> agencias = new LinkedList<>();
03   public void adicionaAgencia(AgenciaBancaria agencia) {
04     agencias.add(agencia);
05   }
06   public AgenciaBancaria buscarAgencia(String codigo) {
07       for (AgenciaBancaria agenciaBancaria : agencias) {
08       if(agenciaBancaria.getCodigo().equals(codigo)){
09         return agenciaBancaria;
10       }
11     }
12     return null;
13   }
14 }
```

Listagem 3. Classe que representa a interface com o usuário do sistema bancário.

```
01 public class VistaAgencia {
02   private int codigoAgencia = 1;
03   private ControladorAgencia controlador = new ControladorAgencia();
04   public void exibir(){
05     Scanner scan = new Scanner (System.in);
06     String opcao = "";
07     while(opcao != null){
08       System.out.println ("Digite :");
09       System.out.println("[1] Para criar nova agencia");
10       System.out.println("[2] Para buscar uma agencia");
11       System.out.println("[3] Para sair.");
12       opcao = scan.nextLine();
13       if(opcao.equals("1")){
14         String codigo = String.valueOf(codigoAgencia);
15         codigoAgencia = codigoAgencia + 1;
16         AgenciaBancaria agencia = new AgenciaBancaria();
17         agencia.setCodigo(codigo);
18         controlador.adicionaAgencia(agencia);
19         System.out.println("Agencia criada com sucesso. Código: "+codigo");
20       }
21       if(opcao.equals("2")){
22         System.out.println("Digite o codigo da agencia:");
23         String codigo = scan.nextLine();
24         AgenciaBancaria agencia = controlador.buscarAgencia(codigo);
25         System.out.println("Codigo da agencia: "+agencia.getCodigo());
26         System.out.println("Numero de contas: "+agencia.getContas().size());
27       }
28       if(opcao.equals("3")){
29         opcao = null;
30       }
31     }
32     scan.close();
33   }
34 }
```

O que chama atenção neste código da **Listagem 1** é a complexidade para incluir esta nova demanda, pois o código-fonte do sistema deveria ser percorrido por inteiro, procurando por todas e quaisquer chamadas ao método **AgenciaBancaria.setContas()** e confirmando, manualmente, se o flag é verificado antes de sua invocação. Da mesma forma, a marcação de uma agência como inativa através da chamada do método **AgenciaBancaria .setAtiva(false)** deve ser precedida por um trecho de código onde a lista contendo as contas daquela agência seria limpa.

Nos próximos tópicos deste artigo será explicado como melhorar esse código somente através da utilização correta dos conceitos fundamentais da Orientação a Objetos em Java: Encapsulamento, Herança e Polimorfismo e do poder destes para reduzir o acoplamento entre os objetos e aumentar a coesão.

Acoplamento e Coesão

Em um projeto orientado a objetos, sempre busca-se diminuir o acoplamento entre as classes e, como consequência, aumentar a coesão.

Acoplamento

Segundo Craig Larman em seu livro "Utilizando UML e Padrões", "acoplamento é a medida de quanto um elemento tem conhecimento, está conectado ou depende de outros elementos. Caso o acoplamento seja *baixo*, a dependência em relação a outros elementos é baixa".

É importante salientar que é impossível remover todo o acoplamento, pois relacionamentos entre classes representam que há um nível, ainda que baixo, de acoplamento entre elas. O que deve ser evitado é a forte dependência entre objetos – tão forte que mesmo uma alteração considerada simples em uma das classes force alterações em cascata em *outras* classes.

Por ser crucial para a reutilização e manutenção de código, o baixo acoplamento é considerado um dos padrões de projeto GRASP (*General Responsibility Assignment Software Patterns*), que sinaliza que deve-se estudar criteriosamente como acontece o relacionamento entre as classes e como é possível reduzir ao máximo possível suas iterações.

Coesão

Ainda no mesmo livro, Craig Larman define o conceito de coesão como "a medida de quão fortemente as responsabilidades de um elemento estão relacionadas e isoladas". De um modo geral, pode-se enxergar a coesão como o *oposto* do acoplamento: ao aumentar uma medida, diminui-se a outra. A alta coesão também é considerada um padrão GRASP.

Resumindo: acoplamento é o grau de dependência entre objetos, enquanto coesão é o grau de independência de uma classe em relação às outras. A partir disso, a primeira questão que surge é: como identificar a dependência entre classes? Analisando o código das **Listagens 1**, **2** e **3**, pode-se identificar um acoplamento alto entre as classes **VistaAgencia** e **AgenciaBancaria**, pois a primeira é a responsável por definir os dados contidos na última. Colocando de outra forma, um objeto de **AgenciaBancaria** *depende* de um objeto **VistaAgencia**. Agora, a segunda e talvez a mais importante questão é: por qual razão isso é considerado uma má prática? A resposta é simples: uma vez que existe uma dependência forte entre os objetos, é muito provável que, ao realizar uma alteração em uma classe, seja para corrigir um defeito ou para incluir um novo requisito, outras funcionalidades do sistema deixarão de funcionar corretamente, causando um efeito dominó que pode até mesmo fazer um sistema deixar de funcionar totalmente em casos extremos.

Para evitar efeitos indesejáveis nas alterações de código, um enorme esforço vem sendo feito por desenvolvedores que compraram a ideia de que testes automatizados resolvem todos os problemas. Contudo, um projeto OO pobre, onde o acoplamento é alto, apenas aumenta o número de testes a serem realizados – alguns autores renomados chegam inclusive a utilizar o termo "explosão de testes". Ou seja, por causa de um código mal escrito, desenvolvedores passam mais tempo criando testes. Contudo, a presença de um número maior de testes não diminui a dificuldade de realizar mudanças em códigos altamente acoplados: apenas vai fazer com que mais testes passem a falhar, alguns até difíceis de antever – um efeito causado pela imprevisibilidade imposta pela baixa coesão.

Por outro lado, um projeto OO bem feito diminui o número de testes necessários, atenuando drasticamente essa imprevisibilidade. Um código com baixo acoplamento é fácil de manter, corrigir e estender, pois as modificações ocorrem somente na classe que possui a responsabilidade de executar uma determinada ação. E esta previsibilidade aumenta a produtividade de um desenvolvedor. Estes são argumentos fortes para que desenvolvedores abandonem a prática de "codificar primeiro, projetar depois ou nunca" para "projetar primeiro, codificar certo depois". Ou seja, realizar o projeto de software não significa perder tempo, mas *ganhar* tempo.

Algumas pessoas tentam vender a ideia de que realizar projeto de software vai contra as metodologias ágeis, como Programação Extrema (XP) e Scrum. Na verdade, Kent Beck, ao assumir a coordenação do projeto que o levou a definir as práticas da XP, optou por ter em sua equipe somente os desenvolvedores sêniores, ou seja, aqueles que possuíam os conceitos fundamentais da OO mais consolidados. Desta forma, somente os desenvolvedores que já **sabiam** como realizar o projeto foram mantidos na equipe, tornando o conhecimento profundo sobre conceitos fundamentais um pré-requisito implícito para a adoção da metodologia. Para aqueles que ainda não são exatamente "fluentes" nestes conceitos, ainda é possível utilizar os conceitos ágeis – contudo, deve-se reservar um tempo para a realização do projeto e este deve ser revisto por alguém com mais experiência. Isto deve ser feito até que os conceitos fiquem "no sangue" do desenvolvedor, que começará a realizar os projetos de forma correta automaticamente. Uma excelente prática para agilizar o aumento da expertise de desenvolvedores mais novos é a programação em par com um desenvolvedor sênior.

Encapsulamento

Para o perfeito entendimento deste conceito, é necessário entender (ou talvez estender) o propósito da orientação a objetos. Um objeto do domínio *não* é, definitivamente, um armazenador de dados. Muito pelo contrário: é um objeto rico, que possui tanto um *estado* como um *comportamento*. O estado é representado pelos valores contidos em seus atributos, enquanto o comportamento são as possíveis maneiras de alterar o seu estado. O Encapsulamento é o princípio que garante que os métodos levarão o objeto de um estado consistente a outro estado igualmente consistente.

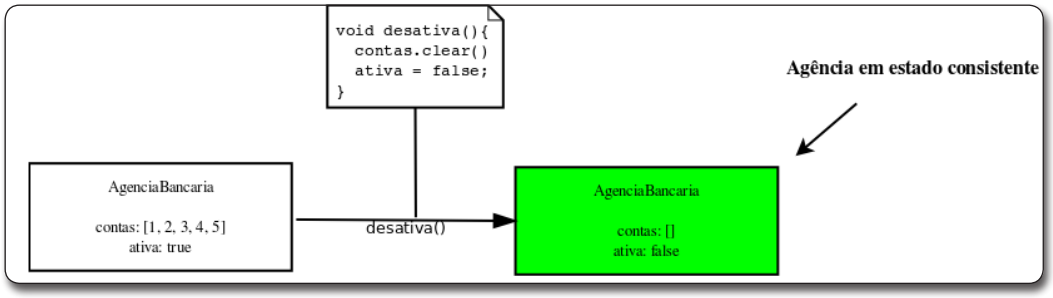

Figura 2. Ação foi encapsulada, mantendo a consistência do objeto

Voltando à regra de não permitir que existam contas vinculadas a agências desativadas, a existência dos métodos **AgenciaBancaria.setContas()** e **AgenciaBancaria.setAtiva()** permite que se vincule contas independentemente de a agência estar inativa, deixando um objeto em um estado *inconsistente*, violando o princípio do Encapsulamento. Mesmo a inclusão de uma verificação no método **setContas()** não resolveria totalmente o problema, uma vez que seria possível desativar uma agência que tivesse contas vinculadas a ela.

Reduzindo o acoplamento (e aumentando a coesão)

O que deve ser feito, então? As seguintes alterações são necessárias para tornar o código mais coeso:

1. O método **AgenciaBancaria.setContas()**, que recebe como parâmetro uma lista de contas, deve ser trocado para **AgenciaBancaria.adicionaConta(ContaBancaria)**, onde uma conta bancária é recebida como parâmetro e adicionada internamente à lista de contas da agência. A conta somente será adicionada caso a agência esteja ativa;
2. Dividir o método **AgenciaBancaria.setAtiva(boolean)** em dois: **AgenciaBancaria.ativa()** e **AgenciaBancaria.desativa()**. Desta forma, o método **AgenciaBancaria.desativa()**, ao ser chamado, limpará a lista de contas daquele objeto e alterará o flag para falso. De forma similar, o método **AgenciaBancaria.adicionaConta(ContaBancaria)** verifica, antes de adicionar qualquer conta, se a agência está ativa e pode abrir novas contas.

O efeito imediato desta pequena alteração é que agora é possível garantir que um objeto do tipo **AgenciaBancaria** estará com seu estado *sempre consistente*, pois as regras de negócio estão encapsuladas em seus comportamentos (vide **Figura 2**).

Outra razão para não utilizar métodos com nomes getXXX() e setXXX()

O livro "Object Thinking", um dos melhores que existem para abrir a cabeça e os olhos dos desenvolvedores de linguagens orientadas a objetos, recomenda que, mesmo nos raros casos em que seja necessário acessar os valores de algum atributo diretamente, isso não deve ser feito através de métodos com nomes **getXXX()** e **setXXX()**. A principal explicação para isso está no fato de que estes nomes de métodos raramente representam de fato os comportamentos de um objeto, dificultando seu entendimento por outros desenvolvedores.

Analisando novamente o exemplo anterior, o método **AgenciaBancaria.getAtiva()** não possui um nome realmente explicativo sobre o que ele faz. Segundo o livro, este método deveria ser renomeado para explicitar, em seu nome, o que ele faz. Por exemplo, ao alterar seu nome para **AgenciaBancaria.estaAtiva()**, melhoramos significativamente o entendimento do que ele faz.

Nota

Martin Fowler, um dos nomes mais importantes na comunidade de desenvolvimento de sistemas na atualidade, considera a modelagem de classes de domínio com métodos getXXX() e setXXX() como anêmica e deve ser considerada um anti-padrão, ou seja, uma má-prática. Segundo ele, o horror fundamental deste anti-padrão está no fato de ele ser completamente contrário às ideias mais básicas de um projeto orientado a objeto, que consiste em combinar dados e processamentos no mesmo lugar. Este é, de fato, somente um estilo de projeto procedural ( ). O pior é que muitas pessoas acreditam que objetos anêmicos são objetos reais e esquecem completamente o propósito da orientação a objetos.

Como dar estado inicial aos objetos

Uma pergunta é muito comum entre os desenvolvedores que estão começando a programar de forma correta e a fugir dos métodos **getXXX()** e **setXXX()**: como dar o estado inicial a um objeto? A resposta é muito simples: utilizando aquilo que foi projetado para esse propósito ou, em outras palavras, os construtores! Contudo, caso a classe possua muitos atributos (mais que 10, por exemplo), pode ser preferível utilizar outros métodos para criação de objetos.

Uma classe com muitos atributos pode ser considerada de alta complexidade. Assim, a utilização dos padrões de projeto Builder, Factory e Abstract Factory, descritos no livro da Gangue dos Quatro, será de muita ajuda.

Quando é permitido utilizar métodos getXXX e setXXX?

Apesar de tudo o que foi descrito até aqui, existem alguns casos, bem específicos, onde os métodos **getXXX()** e **setXXX()** podem e devem ser utilizados.

O principal cenário de uso é quando projeta-se componentes que serão utilizados por outros sistemas, como o caso de componentes de interfaces estilo Delphi. Este tipo de objeto praticamente não possui estados inconsistentes, pois representam elementos gráficos e os valores de seus atributos são totalmente independentes uns dos outros.

Infelizmente, existem alguns *frameworks* onde a utilização destes métodos é obrigatória. Um destes vilões, senão o principal deles, é a Java Persistence API, ou JPA, uma ferramenta que realiza o mapeamento entre as classes de um projeto orientado a objetos e suas correspondentes tabelas em um banco de dados relacional (tarefa conhecida como Mapeamento Objeto-Relacional). A JPA utiliza estes métodos para acessar o estado dos objetos na hora de sincronizar os dados entre sistema e o SGBD, fazendo uso da API de Reflexão do Java. Contudo, algumas destas ferramentas permitem que estes métodos sejam criados com visibilidade *privada*, impedindo que estes métodos sejam chamados por objetos de outras classes. Apesar de ainda não ser ideal, é uma alternativa bem melhor do que permitir que objetos inconsistentes sejam salvos no banco de dados.

Revisitando o Padrão Modelo-Visão-Controle (MVC)

Martin Fowler diz, em seu livro "Padrões de Arquitetura de Aplicações Corporativas" que o padrão de projeto Modelo-Visão-Controle ou, como é mais conhecido, MVC, é "provavelmente o padrão mais citado no desenvolvimento de interfaces e, provavelmente, o que é mais citado de forma equivocada.". Originalmente projetado para ser um padrão para **gerenciamento de interfaces gráficas com o usuário** (GUI), o MVC, por alguma razão, começou a ser utilizado como padrão para arquiteturas de sistemas em geral, especialmente sistemas Web.

O principal problema nesta abordagem está nos entendimentos errôneos que passaram a acompanhar o padrão. A seguir alguns destes enganos são listados:

- As classes de modelo servem simplesmente para representar, dentro do programa, as tabelas do banco de dados, sendo simples "armazenadores de dados";
- Outras abordagens afirmam que a camada de modelo deve ser a responsável pela comunicação com o banco de dados;
- Dentro das classes controladoras reside toda a lógica de negócio;
- A classes que compõem a visão não devem acessar diretamente as classes de modelo;
- Entre outros.

Estes enganos são alguns dos maiores vilões na elaboração de um bom projeto orientado a objetos, pois suas afirmações aumentam consideravelmente o acoplamento entre as camadas (e, como consequência, entre as classes), dificultando tanto a compreensão do código como sua extensibilidade e sua manutenibilidade.

O padrão MVC, em sua forma original, possui o diagrama conforme a **Figura 3**.

É importante reparar que, por esse diagrama, as classes de vista têm acesso às classes de modelo, enquanto o controlador pode ser entendido somente como um coordenador de interfaces, analisando de qual tela o usuário veio e, dependendo do resultado do método executado pela classe do modelo, para qual tela o usuário deve ser direcionado. Em sua forma original, as regras de negócio não ficam no controlador, mas nos métodos das classes que compõem a camada de modelo. Além disso, as classes que representam a interface podem acessar e trabalhar com as classes da camada de modelo, pois precisam dos seus dados para exibir as informações na tela.

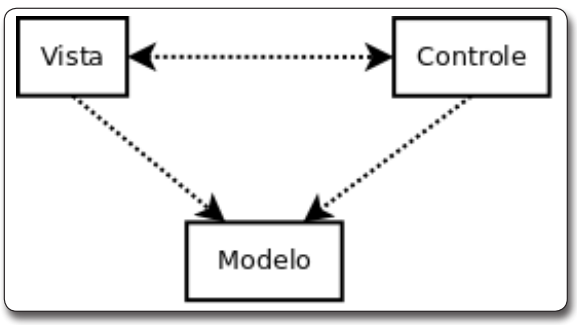

Figura 3. Diagrama do padrão MVC

Herança

A seguinte história foi contada por Allen Holub em seu famoso artigo "*Why extends is evil*" (veja a seção **Links**): em um evento, James Gosling, o criador do Java, foi perguntado sobre o que deixaria de fora caso tivesse que refazer a linguagem. A sua resposta foi simples e direta: "Deixaria de fora as classes". Após as risadas cessarem, ele explicou que o problema principal não eram as classes, mas a Herança.

Apesar de ser um conceito extremamente poderoso, capaz de enriquecer um projeto e deixá-lo mais compreensível, a Herança é utilizada na maioria das vezes somente como uma maneira de eliminar duplicação de código, sem que os desenvolvedores prestem atenção em seus efeitos colaterais no projeto. Por exemplo, ao fazer com que uma classe estenda as funcionalidades de outra, cria-se um altíssimo acoplamento – talvez o maior que exista – e, como foi visto anteriormente, isso deve ser sempre evitado.

Para ilustrar, considere a seguinte hierarquia de classes contida na **Figura 4**. A **Listagem 4** exibe o código das classes.

Como podemos verificar, o código é bem simples: uma pessoa tem nome e sobrenome, e seu método **exibir()** imprime na tela seu nome completo. Já um contribuinte, que é uma pessoa, estende as funcionalidades da classe, adicionando o atributo CPF. Por fim, a classe **Imigrante** adiciona a informação sobre o passaporte e sobre o país de origem. É nesse momento que os problemas começam a aparecer. Considere o código a seguir:

```
Imigrante imigrante = new Imigrante("Reinaldo", "Silva", "01010101234", "CCB1234",
"Portuguesa");
empregadoEstrangeiro.exibir();
```

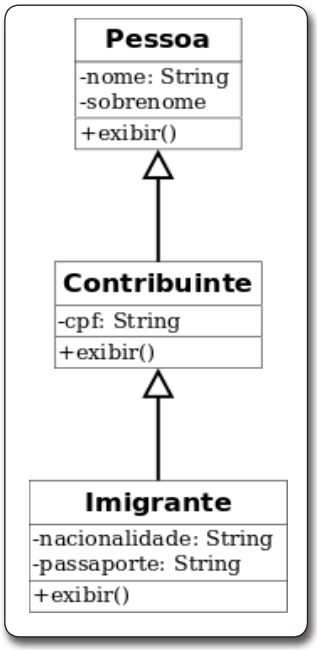

Figura 4. Hierarquia de classes

Listagem 4. Exemplo de código com Herança.

```
public class Pessoa {
   private String nome;
   private String sobrenome;
   public Pessoa(String nome, String sobrenome){
      this.nome = nome;
      this.sobrenome = sobrenome;
   }
   public void exibir(){
      System.out.println("Nome completo: "+ nome + " "+sobrenome);
   }
}
public class Contribuinte extends Pessoa {
   private String cpf;
   public Contribuinte(String nome, String sobrenome, String cpf){
         super(nome, sobrenome);
      this.cpf = cpf;
   }
   public void exibir(){
      super.exibir();
      System.out.println("CPF: "+cpf);
   }
}
public class Imigrante extends Contribuinte {
   private String nacionalidade;
   private String passaporte;
   public Imigrante (String nome, String sobrenome,
            String cpf, String passaporte, String nacionalidade) {
      super(nome, sobrenome, cpf);
      this.passaporte = passaporte;
      this.nacionalidade = nacionalidade;
   }
   @Override
   public void exibir() {
      super.exibir();
      System.out.println("Passaporte: "+passaporte);
      System.out.println("Nacionalidade: "+nacionalidade);
   }
}
```

A saída da execução dessa linha de código seria a seguinte:

```
Nome completo: Reinaldo Silva
CPF: 01010101234
Passaporte: CCB1234
Nacionalidade: Portuguesa
```

O programa funciona a contento, até que alguém perceba que deve-se alterar o método **exibir()** no caso de a nacionalidade de um empregado for chinesa, pois é costume apresentar o sobrenome *antes* do nome, ou seja, a ordem de exibição é invertida. Como resolver este impasse? Para que as informações fossem impressas na tela de forma correta, o jeito natural seria reescrever completamente o método **exibir()** na classe **Imigrante**, o que é impossível de fazer uma vez que os atributos nome e sobrenome são privados na classe **Pessoa**, ou seja, inacessíveis.

A saída que muitos adotam é recorrer aos métodos **getXXX()** e **setXXX()**, retornando ao problema anterior relacionado ao Encapsulamento. Obviamente, esta não é a melhor solução, pois empobrece o projeto.

Outra solução adotada é "subir" na hierarquia de classes os atributos das subclasses, tornando possível, por exemplo, que o método **exibir()**, na classe **Pessoa**, possa realizar a checagem da nacionalidade. Esta abordagem também não é interessante, principalmente por agregar em uma classe atributos que não são de sua responsabilidade.

Por fim, a saída que resta é a classe **Imigrante** deixar de estender **Empregado**. Desta forma, todos os atributos ficariam contidos na própria classe, permitindo que o método **exibir()** fosse implementado de forma correta em cada uma das classes. Esta abordagem é satisfatória e elimina todo e qualquer acoplamento entre as classes **Pessoa**, **Contribuinte** e **Imigrante**. A **Listagem 5** mostra como ficaria a implementação da classe **Imigrante**.

Listagem 5. Classe Imigrante após a inclusão da checagem de nacionalidade.

```
public class Imigrante {
   private String nacionalidade;
   private String passaporte;
   public Imigrante(String nome, String sobrenome,
            String cpf, String passaporte, String nacionalidade) {
      this.nome = nome;
      this.sobrenome = sobrenome;
      this.cpf = cpf;
      this.passaporte = passaporte;
      this.nacionalidade = nacionalidade;
   }
   public void exibir() {
      if(nacionalidade.equals("chinesa"){
         System.out.println("Nome completo: "+ sobrenome + " " + nome);
      }else{
         System.out.println("Nome completo: "+ nome + " " + sobrenome);
      }
      System.out.println("CPF: " + cpf);
      System.out.println("Passaporte: "+passaporte);
      System.out.println("Nacionalidade: "+nacionalidade);
   }
}
```

Polimorfismo

No caso de existir a necessidade de trabalhar com objetos do tipo **Pessoa**, **Contribuinte** e **Imigrante** de forma indistinta, faz-se uso do terceiro conceito fundamental da Orientação a Objetos: o Polimorfismo que, por sua vez, também é um padrão GRASP e utilizado de forma abundante no livro da Gangue dos Quatro (**BOX 1**).

BOX 1. Polimorfismo e Padrões de Projeto da Gangue dos Quatro

O famoso livro sobre Padrões de Projeto da Gangue dos Quatro descreve diversas soluções para problemas recorrentes no desenvolvimento de softwares orientados a objetos. Tais problemas são agrupados em três tipos de categorias: para criar objetos, para representar sua estrutura e para realização de operações (padrões de criação, estruturais e comportamentais). Todos os exemplos no livro são apresentados em C++, uma linguagem que não possui o conceito de implementação de interface, apenas Herança. Contudo, ao analisar cuidadosamente cada exemplo, um programador Java conseguirá reparar que na grande maioria das vezes a Herança pode ser substituída pela implementação de interfaces com delegação, o que reforça a afirmação de que esta prática reduz o acoplamento

Em sua forma mais simples, o Polimorfismo significa poder referenciar objetos através de suas superclasses. Assim, a construção a seguir é perfeitamente aceitável:

```
Object pessoa = new Pessoa("Joao","Silva");
```

Contudo, apenas os métodos definidos na classe **Object** (**toString()**, **hashCode()**, **equals(...)**, etc.) poderão ser acessados na variável **pessoa**. Em Java, o conceito de Polimorfismo está ligado não somente à Herança, como também – e principalmente – à implementação de interfaces.

Em princípio, pode parecer muito mais trabalhoso implementar interfaces do que estender classes – afinal, *todos* os métodos da interface serão reimplementados nas subclasses, enquanto na Herança é possível reaproveitar código, bastando movê-lo para a superclasse. Contudo, existe o problema do aumento do acoplamento descrito no tópico anterior. Na verdade, existe um truque (na realidade, é mais um conceito do que um truque), que consiste na conversão de uma Herança para uma *Delegação*, através da criação de um campo naquela que seria a subclasse da classe que seria sua classe pai. A **Figura 5** ilustra como acontece a Delegação.

A **Listagem 6** mostra o código correspondente.

Listagem 6. Polimorfismo com implementação de interfaces e Delegação.

```
public interface IPessoa {
   void exibir();
}

public class Pessoa implements IPessoa {
   private String nome;
   private String sobrenome;
   public Pessoa(String nome, String sobrenome) {
      this.nome = nome;
      this.sobrenome = sobrenome;
   }
   @Override
   public void exibir() {
      System.out.println("Nome completo: " + nome + " " + sobrenome);
   }
}

public class Contribuinte implements IPessoa {
   private IPessoa pessoa;
   private String cpf;
   public Contribuinte(String nome, String sobrenome, String cpf) {
      pessoa = new Pessoa(nome, sobrenome);
      this.cpf = cpf;
   }
   @Override
   public void exibir() {
      pessoa.exibir();
      System.out.println("CPF: "+cpf);
   }
}

public class Imigrante implements IPessoa {
   private Contribuinte contribuinte;
   private String passaporte;
   private String nacionalidade;
   public Imigrante(String nome, String sobrenome, String cpf, String passaporte,
   String nacionalidade) {
      contribuinte = new Contribuinte(nome, sobrenome, cpf);
      this.passaporte = passaporte;
      this.nacionalidade = nacionalidade;
   }
   @Override
   public void exibir() {
      contribuinte.exibir();
      System.out.println("Passaporte: "+passaporte);
      System.out.println("Nacionalidade: "+nacionalidade);
   }
}
```

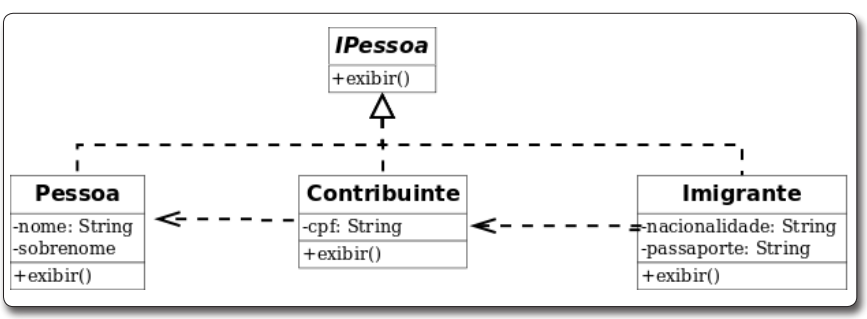

Figura 5. Substituindo herança por delegação

Repare que todas as classes implementam a mesma interface **IPessoa** e, como consequência, possuem o método **exibir()**. Assim, em vez de a classe **Contribuinte** herdar da classe **Pessoa**, ela possui um atributo desse tipo. De forma semelhante, a classe **Imigrante** possui um atributo do tipo **Contribuinte**. Em cada implementação do método **exibir()**, antes de mais nada, há a *delegação* da chamada para o atributo que representaria a superclasse (na prática, ocorre uma troca de **super.exibir()** para **atributo.exibir()**).

A principal diferença neste caso em comparação ao anterior, quando utilizou-se Herança, está na facilidade de resolver o problema de imprimir o nome invertido caso a nacionalidade do imigrante seja chinesa: basta acertar *somente* a classe **Imigrante**. As outras implementações nas classes **Pessoa** e **Contribuinte** continuam inalteradas, mostrando a principal vantagem de se possuir um projeto com baixo acoplamento e alta coesão. A **Listagem 7** exibe como ficaria o código para atender ao requisito.

Listagem 7. Classe Imigrante após a alteração para exibição do nome completo de acordo com a nacionalidade.

```
public class Imigrante implements IPessoa {
   private String nome;
   private String sobrenome;
   private String cpf;
   private String passaporte;
   private String nacionalidade;
   public Imigrante(String nome, String sobrenome, String cpf, String passaporte,
   String nacionalidade) {
      this.nome = nome;
      this.sobrenome = sobrenome;
      this.cpf = cpf;
      this.passaporte = passaporte;
      this.nacionalidade = nacionalidade;
   }
   @Override
   public void exibir() {
      if(nacionalidade.equals("Chinesa")){
         System.out.println("Nome completo: " + sobrenome + " " + nome);
      }else{
         System.out.println("Nome completo: " + nome + " " + sobrenome);
      }
      System.out.println("CPF: "+cpf);
      System.out.println("Passaporte: "+passaporte);
      System.out.println("Nacionalidade: "+nacionalidade);
   }
}
```

Para atingir esse objetivo, bastou que a classe **Imigrante** passasse a abrigar todos os atributos necessários dentro dela. Por fim, note que todas as classes continuam sendo do mesmo tipo, pois implementam a mesma interface **IPessoa**, e, portanto, o conceito de Polimorfismo não fica prejudicado.

Suporte da IDE Eclipse

A IDE Eclipse já vem com suporte a algumas destas boas práticas, facilitando e agilizando a criação de diversos métodos. Por exemplo, o código na **Listagem 8** mostra uma classe especial de lista, na qual é possível adicionar ou ler elementos somente em seu início ou ao seu final.

Ou seja, possui apenas um atributo do tipo **LinkedList**. Ao clicar com o botão direito do mouse em cima desse atributo, abre-se um menu que, entre várias opções, possui uma chamada *Source*. Ao posicionar o mouse em cima desta opção, um submenu aparece e, nele, há a opção para a geração de métodos de delegação. Ao selecionar esta opção, veremos uma tela semelhante à apresentada na **Figura 6**.

Se marcarmos os métodos **addFirst(E)**, **addLast(E)**, **clear()**, **getFirst()** e **getLast()** e apertarmos *Ok*, nossa classe ficará conforme a **Listagem 9**.

Listagem 8. Classe contendo somente o atributo.

```
import java.util.LinkedList;
public class MinhaColecao<E> {
   private LinkedList<E> lista;
}
```

Listagem 9. Código após a geração dos métodos de delegação pelo Eclipse.

```
import java.util.LinkedList;
public class MinhaColecao<E> {
   public void addFirst(E e) {
      lista.addFirst(e);
   }
   public void addLast(E e) {
      lista.addLast(e);
   }
   public void clear() {
      lista.clear();
   }
   public E getFirst() {
      return lista.getFirst();
   }
   public E getLast() {
      return lista.getLast();
   }
}
```

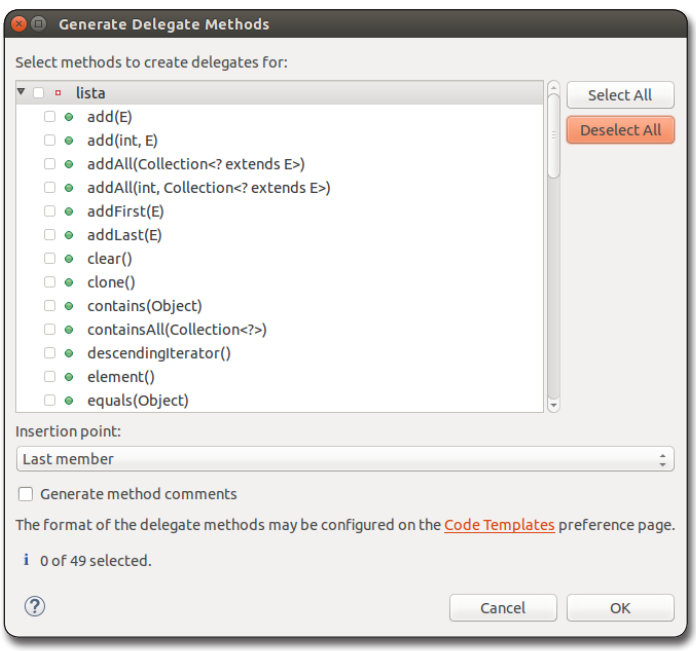

Figura 6. Geração de métodos de delegação com o Eclipse

Outro recurso muito útil do Eclipse é a criação automática de interfaces a partir de classes, permitindo a troca substancial da utilização de Herança pela implementação de interfaces. Para tanto, com a classe **MinhaColecao** aberta, clique no menu *Refactor* e depois em *Extract Interface*. A tela na **Figura 7** será exibida em seguida. Então, basta definir o nome da nova interface e os métodos que farão parte dela. Ao confirmar a operação, a classe ficará conforme exibida na **Listagem 10**.

Estas são apenas duas pequenas amostras do que é possível automatizar com o Eclipse. Sendo assim, vale muito a pena dedicar algum tempo para aprender o que cada ação nestes menus faz, pois o aumento na produtividade pode ser imenso.

Muitos desenvolvedores possuem entendimentos incorretos sobre os conceitos fundamentais de Orientação a Objetos. Estes enganos frequentemente levam a códigos altamente acoplados, criando um efeito de imprevisibilidade para toda e qualquer alteração a ser realizada no programa. Para atenuar este problema, desenvolvedores passam horas criando testes automatizados que somente ajudam a detectar *onde* o programa quebrou e não a *causa* da instabilidade.

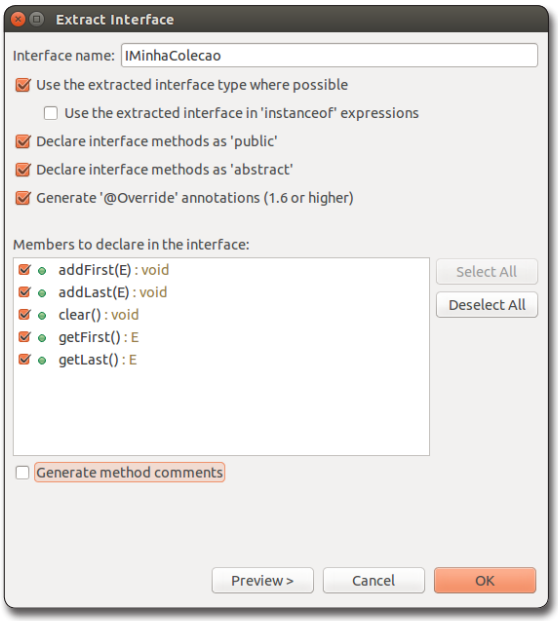

Figura 7. Extraindo uma interface a partir de uma classe no Eclipse

Listagem 10. Código após a extração da interface pelo Eclipse.

```
public interface IMinhaColecao<E> {
   public abstract void addFirst(E e);
   public abstract void addLast(E e);
   public abstract void clear();
   public abstract E getFirst();
   public abstract E getLast();
}

public class MinhaColecao<E> implements IMinhaColecao<E> {
   private LinkedList<E> lista;
   @Override
   public void addFirst(E e) {
      lista.addFirst(e);
   }
   @Override
   public void addLast(E e) {
      lista.addLast(e);
   }
   @Override
   public void clear() {
      lista.clear();
   }
   @Override
   public E getFirst() {
      return lista.getFirst();
   }
   @Override
   public E getLast() {
      return lista.getLast();
   }
}
```

Um bom projeto orientado a objetos diminui o acoplamento e aumenta a coesão, tornando as mudanças inevitáveis simples de serem realizadas. Possui ainda, como efeito colateral de sua previsibilidade, a redução do número de testes automatizados – o que permite aos desenvolvedores focarem na implementação de novas funcionalidades em vez de correções de bugs.

Para os leitores que desejarem aprofundar o estudo sobre os conceitos apresentados neste artigo, a leitura do livro "Utilizando UML e Padrões", de Craig Larman, é o ponto de partida. O livro "Object Thinking" realiza uma transição perfeita e suave entre os pensamentos procedural e orientado a objetos, sendo assim de muita relevância para o tema, enquanto o livro da Gangue dos Quatro constitui uma leitura obrigatória para todos aqueles que desejarem se tornar desenvolvedores eficientes.

Ainda dentro deste tema, o livro "Refatoração: Aperfeiçoando o Projeto de Código Existente", escrito por Martin Fowler, traz um catálogo de possíveis melhorias no código – muitas delas foram incorporadas por IDEs, como o Eclipse. A leitura deste livro é recomendada para aqueles que estão trabalhando em sistemas que já estão em produção, cujos projetos podem ser melhorados.

Autor

Luís Fernando Orleans
lforleans@{ieee.org, ufrrj.br} - http://lforleans.blogspot.com.br/
É Professor Adjunto do curso de Ciências da Computação da Universidade Federal Rural do Rio de Janeiro (UFRRJ) e obteve o grau de Doutor em Engenharia de Sistemas e Computação em 2013. Desenvolve e coordena diversos projetos em Java desde 2001, tendo acompanhado de perto todas as novidades da linguagem desde então. Apaixonado pela família, por tecnologia, café, vinho, Flamengo e bons papos.

Links:

Modelagem Anêmica.
http://www.martinfowler.com/bliki/AnemicDomainModel.html

Object Thinking, Livro. David West, Microsoft Press, 2004.

Refatoração: Aperfeiçoando o Projeto de Código Existente. Martin Fowler, Bookman, 2004.

Utilizando UML e Padrões. Craig Larman, Artmed, 2007.

Why Extends is Evil.
http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html

Why getter and setter methods are evil.
http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos. Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm, Bookman, 2004

Como interpretar Diagramas de Classes da UML – Parte 1

Pondo em prática os conceitos de orientação a objetos a partir da interpretação de um diagrama de classes

Este artigo faz parte de um curso

A linguagem Java segue, em grande parte de sua concepção, o paradigma da Orientação a Objetos. Dessa forma, quanto melhor for o seu conhecimento sobre esse paradigma, melhor será o reuso do seu código e a resolução de alguns problemas triviais a partir de conceitos como herança, polimorfismo e encapsulamento. Outro aspecto, muito relacionado à orientação a objetos, está na produção e interpretação de diagramas UML (*Unified Modeling Language*), sendo um dos diagramas mais básicos o Diagrama de Classes, o qual às vezes não é estudado profundamente pelos desenvolvedores de uma equipe e, devido a isso, muitas regras implícitas acabam passando despercebidas.

A partir do diagrama de classes é possível estabelecer algumas regras de negócio nas associações entre as classes e essas regras podem vir a alterar a criação de métodos básicos como construtores e métodos setters de atributos, dependendo da obrigatoriedade da associação.

Dito isso e com o objetivo de explorar esse tipo de diagrama, neste artigo vamos apresentar e analisar detalhadamente um exemplo de diagrama de classes projetado para um sistema que realizará o cadastro e a venda de bilhetes aéreos. Para isso, retrataremos apenas a parte mais básica de um software, a camada model, que contém todas as abstrações do mundo real essenciais para o estabelecimento das regras de negócio.

Fique por dentro

Independentemente da área de atuação no desenvolvimento de software, o diagrama de classes possui muitas informações que podem estar ocultas sob seus elementos para pessoas que não estão treinadas em observá-las, sejam elas desenvolvedores ou analistas de negócio, estes que muitas vezes são responsáveis pela diagramação mesmo sem conhecimentos aprofundados de como fazer isso. Vale ressaltar que é muito importante a correta interpretação deste para gerar classes e atributos corretos e, principalmente, para a utilização dos construtores das classes na construção das regras de negócio de um software a partir das obrigatoriedades existentes nas associações entre as classes. Com base nisso, este artigo visa explorar a apresentação desses aspectos para solidificar o conhecimento na interpretação e produção de diagramas de classes da UML.

Requisitos do sistema de bilhetes aéreos

O sistema de bilhetes aéreos possui como premissa básica a reserva, compra e check-in de bilhetes aéreos de diversas companhias por um passageiro. Para cumprir essa premissa, a primeira funcionalidade do sistema deverá ser o cadastro de aeroportos por um administrador. Este cadastro deve ser composto pelas seguintes informações: nome, código e endereço completo (Logradouro, CEP, Cidade, Estado, País).

Para continuar o cumprimento da premissa básica, o sistema deve permitir o cadastro de companhias aéreas (nome e código) e, a partir da conclusão deste, possibilitar o cadastro de informações relacionadas a aviões, rotas de voos, funcionários e cargos que pertencem à companhia.

Para cadastrar um avião, deve ser informado a qual companhia aérea ele pertence, o código do avião, a carga máxima suportada e a quantidade de assentos para as três modalidades (classes) de bilhetes existentes: primeira, executiva e econômica. Além disso, vale destacar que ao cadastrar um avião este deverá pertencer somente a uma companhia aérea, mas esta poderá ter vários aviões.

O sistema deve permitir também o cadastro de cargos de trabalho que uma companhia aérea possui. Um cargo dentro de uma companhia aérea representa um oficio de trabalho; por exemplo, um piloto. Para realizar esse cadastro, deve ser mencionada a qual companhia aérea este pertence, além do nome do cargo e sua descrição.

Outro cadastro necessário e relacionado a uma companhia é o de funcionários. No sistema, todo funcionário deve possuir somente um cargo. Por outro lado, um cargo pode ser associado a diversos funcionários. Além do cargo, o cadastro de um funcionário exige as informações de nome, e-mail, telefone, data de nascimento, login e senha no sistema, código de matrícula dentro da companhia, uma conta corrente para o depósito do salário e o endereço completo.

Por fim, o sistema deve permitir que uma companhia aérea cadastre rotas de voo. Uma rota de voo é composta por dois aeroportos, sendo um de origem e um de destino. Além disso, a rota obriga o cadastro de um nome e descrição.

Após a criação das rotas de voo, a companhia aérea deve disponibilizar os horários em que atua para cada uma das rotas existentes. Esses horários são compostos basicamente pela data, o horário de partida e uma estimativa da data e horário de chegada. Ademais, cada horário deve ter um código, uma quantidade de bilhetes disponíveis para cada classe (primeira, executiva e econômica) e a reserva de um avião para a sua realização. Por fim, um último aspecto relacionado ao horário é que ao criar um, este deverá gerar os bilhetes correspondentes para cada classe, a partir da quantidade informada no momento do cadastro do bilhete.

Neste momento é importante ressaltar que cada bilhete possui um estado relacionado à sua comercialização, que pode ser: Disponível, Reservado ou Comprado. O estado Disponível representa que o bilhete ainda não foi reservado ou adquirido por algum passageiro. O estado Reservado representa que o bilhete foi reservado por um passageiro para sua futura aquisição. E o estado Comprado, representa que o bilhete já foi adquirido por um passageiro.

Conforme o comportamento esperado, um passageiro pode comprar qualquer bilhete, independente da classe deste, e deve possuir as seguintes informações de cadastro: nome, e-mail, telefone, data de nascimento, login e senha no sistema, número de cartão de crédito e o documento, o qual pode ser o CPF, RG ou outra informação de identificação.

Por fim, ao adquirir um bilhete é permitido ao passageiro realizar o despacho das bagagens. Neste momento, cada uma das classes de bilhete especifica um número máximo de bagagens (primeira, 3 bagagens, executiva, 2 e econômica, 1 bagagem) e para cada bagagem deve ser informado o seu peso na hora do check-in.

Conceitos de Orientação a Objetos

O conceito base da Orientação a Objetos são as classes. Uma classe é a representação do mundo real de acordo com o escopo de um software. Exemplificando: Se está sendo desenvolvido um software bancário, é necessário possuir um atributo em pessoa que represente o tipo sanguíneo? E se fosse um software médico, seria necessário?

A resposta para a primeira pergunta provavelmente será "Não!". Já para a segunda pergunta, "Sim!". Contudo, isso pode variar. Por exemplo: se for um software de Seguro de Vida, será que não começaria a ser interessante na classe **Pessoa** haver um atributo relacionado ao tipo sanguíneo? Deste modo, constatamos que o escopo do software determina quais informações podem estar em uma classe.

Como sabemos, não devemos implementar toda uma aplicação em apenas uma classe. Algo que solidifica esse conhecimento é o princípio da responsabilidade única (vide **BOX 1**). E a partir da implementação de diferentes classes, é de se esperar que existam relacionamentos entre elas para que com a soma disso tudo, alcancemos uma boa aplicação. A esse relacionamento é dado o nome de Associação.

BOX 1. Princípio da Responsabilidade Única

O princípio da responsabilidade única foi introduzido por Tom DeMarco em 1979, no seu livro Structured Analysis and Systems Specification. Esse princípio determina que: "Deve existir um e somente um motivo para que uma classe mude". Ou seja, uma classe deve ser implementada com apenas uma única responsabilidade, um único objetivo. Ao possuir mais de uma responsabilidade, provavelmente está fazendo mais coisas do que deveria. Por exemplo: uma classe Conta que também possui as informações de uma pessoa (nome, cpf, endereco). Em vez disso, o correto seria criar a classe Conta e também a classe Pessoa.

Associações entre classes

Uma associação entre duas classes é a forma de representar o relacionamento entre elas. Na notação de um diagrama de classes UML, uma associação é basicamente composta por uma linha entre as classes, como mostra a **Figura 1**.

A fim de facilitar o entendimento das próximas explicações sobre associações, vamos definir dois conceitos importantes, a saber: 1) a classe que inicia uma associação é chamada de *Classe Origem*, e no caso especificamente da **Figura 1**, é a classe Aeroporto; 2) a classe que finaliza uma associação é chamada de *Classe Destino*, e neste mesmo exemplo, é a classe **Endereco**.

Basicamente, uma associação faz com que a classe destino esteja presente na classe origem através de um atributo, isto é, conforme mostra a **Listagem 1**, na linha 02 existe um atributo **Endereco** na classe **Aeroporto**. Já no caso da classe **Endereco**, essa não possui nenhum atributo relacionado à classe **Aeroporto**, devido ao tipo da associação apresentada na **Figura 1**.

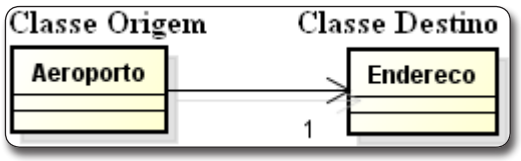

Figura 1. Associação entre Aeroporto e Endereco

Listagem 1. Exemplo de código da classe Aeroporto a partir da associação da **Figura 1**.

```
01    public class Aeroporto {
02      private Endereco endereco;
03
04      //Outros Atributos declarados
05
06      public Endereco getEndereco(){
07        return endereco;
08      }
09
10      //Outros Métodos declarados
11
12    }
```

Mas como foi possível definir que dentro da classe **Aeroporto** deveria ter um atributo da classe **Endereco**? Poderia ser uma lista de Enderecos? E por que não há um atributo da classe **Aeroporto** na classe **Endereco**? As respostas a essas perguntas são obtidas por quatro características básicas existentes dentro de uma associação UML:

- Navegabilidade;
- Multiplicidade;
- Conectividade;
- Obrigatoriedade.

Cada uma dessas características será explicada mais profundamente nos subtópicos a seguir.

Associações entre classes – Navegabilidade

A característica de navegabilidade está relacionada ao aspecto de visibilidade da classe origem sobre a classe destino, ou seja, se a classe origem "enxerga" a classe destino. Essa visibilidade significa, na prática, a existência de atributos internos na classe origem que representem a classe destino.

Para representar o conceito de navegabilidade em um diagrama de classes UML temos algumas opções. Uma delas está disposta na **Figura 1**, a qual é caracterizada por uma seta (→). Essa seta possui o significado de direção, fazendo com que a associação possua um sentido da classe origem para a classe destino. Tal recurso visual caracteriza uma associação **Unidirecional**, e neste caso ela pode ser interpretada como: a classe **Aeroporto** consegue visualizar a classe **Endereco**. No código, isso é definido adicionando um atributo do tipo **Endereco** na classe **Aeroporto**.

Agora, observe a **Figura 2**. Diferentemente da associação da **Figura 1**, não existe nenhuma seta, o que caracteriza uma associação **Bidirecional**. Neste caso, cada uma das classes consegue visualizar a outra através de um atributo que a representa. A leitura desta associação a partir do diagrama de classes deve ser feita da seguinte forma: *A Classe Origem possui N instâncias da Classe Destino*.

O valor N deve ser obtido através da leitura do cardinal (número) mais próximo da classe destino. Sendo assim, considerando a classe **Horario** como Origem e a classe **Rota** como Destino, a leitura da **Figura 2** deve ser realizada da seguinte forma: *Um Horario possui Uma Rota*.

O sentido contrário dessa leitura, que tem a classe **Rota** como Origem e **Horario** como Destino, deve ser: *Uma Rota possui Zero ou mais Horarios*. Interpretando esse texto para código Java, devemos codificar um atributo **Rota** dentro da classe **Horario** e um atributo que representa uma coleção de **Horarios** dentro da classe **Rota**. O código destas classes pode ser verificado nas **Listagens 2** e **3**, respectivamente.

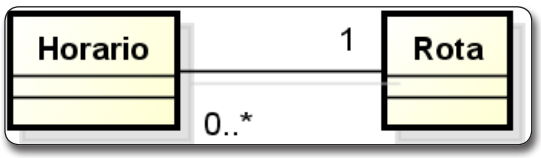

Figura 2. Representação da Associação entre Horario e Rota

Listagem 2. Exemplo de código da classe Horario a partir da associação da **Figura 2**.

```
01    public class Horario {
02      private Rota rota;
03
04      //Outros atributos declarados
05
06      public Rota getRota(){
07        return rota;
08      }
09
10      //Outros métodos declarados
11
12    }
```

Listagem 3. Exemplo de código da classe Rota a partir da associação da **Figura 2**.

```
01    public class Rota {
02      private java.util.List<Horario> horarios;
03
04      //Outros atributos declarados
05
06      public Horario getHorarios(){
07        return horarios;
08      }
09
10      //Outros métodos declarados
11
12    }
```

Para fixar melhor esse assunto, vamos reescrever esta associação utilizando apenas associações Unidirecionais, conforme o desmembramento apresentado na **Figura 3**. Repare que se a leitura de cada associação for realizada, serão obtidas as mesmas frases do parágrafo anterior (*Um horario possui Uma Rota; Uma Rota possui Zero ou mais Horarios*), mas assim fica mais fácil de entender o

conceito de associação bidirecional. Esse desmembramento possui o mesmo significado, logo os atributos continuam sendo criados conforme as **Listagens 2** e **3**.

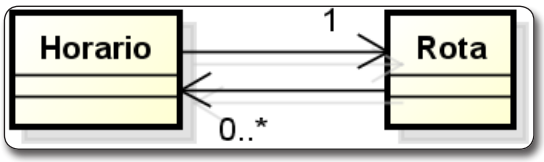

Figura 3. Reescrevendo a associação entre Horario e Rota com associações unidirecionais

Associações entre classes – Multiplicidade

A próxima característica de uma associação a ser aprofundada neste artigo é a Multiplicidade. Essa característica já foi apresentada nos parágrafos anteriores, mas sem fazer sua apresentação formal. A importância dessa característica está no fato de indicar quantos objetos da classe destino uma classe origem "suporta". Para tanto, esse número possui dois limites, um **limite inferior**, o qual indica a quantidade mínima de objetos que a classe deve suportar e também a possível obrigatoriedade da associação, e um **limite superior**, o qual indica a quantidade máxima de objetos que a classe deve suportar. Isso pode ser representado de várias formas, como pode ser visto na **Tabela 1**.

Multiplicidade	Simbologia em um Diagrama de Classes
Zero ou Um Objeto	0..1
Zero ou Muitos Objetos	0..* ou *
Apenas Um Objeto	1..1 ou 1
Um ou Muitos Objetos	1..*
Intervalo específico de Objetos	Limite Inferior..Limite Superior (Exemplo: 2..5)

Tabela 1. Exemplos de Multiplicidades presentes em um Diagrama de Classes

Resumindo, ao observar a **Listagem 2** é possível perceber que para atributos com limite superior igual a '1' (um), cria-se apenas um atributo na classe origem. Ao observar a **Listagem 3** é possível perceber que para atributos com limite superior '*' (Muitos), cria-se uma lista do tipo da classe destino na classe origem.

Agora, observe a **Figura 4**, que representa uma associação unidirecional entre as classes **Rota** e **Aeroporto**. O que deverá ser criado na classe **Rota** para representar essa associação? Repare que a associação possui um limite inferior igual a dois e um limite superior igual a dois. Essa associação está correta?

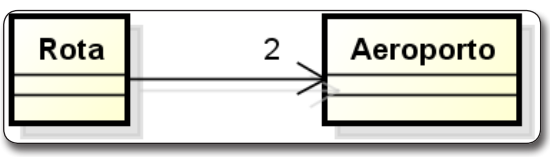

Figura 4. Associação entre Rota e Aeroporto

Sim, essa associação está correta! Para ajudar a entender o que deve ser feito nesse caso, pense no escopo do projeto de Bilhetes Aéreos. Em uma rota de voo, por exemplo, Rio de Janeiro – São Paulo, você tem dois aeroportos, um de origem e um de destino, isso é fato. Agora, vamos pensar na codificação do código Java: o que seria mais claro, uma Lista de Aeroportos ou dois atributos **Aeroporto**?

Pensando em uma lista, o limite máximo deve ser controlado, para que sempre permaneça em duas instâncias de aeroporto, no máximo, respeitando a regra de negócio implícita na associação. Além disso, devemos ter um "acordo" entre todos os desenvolvedores do projeto, especificando que o **get(0)** dessa lista representa o aeroporto de origem e o **get(1)**, representa o aeroporto de destino. Será que isso é claro para todos os envolvidos no desenvolvimento do software?

Se a opção for criar dois atributos do tipo **Aeroporto**, estes podem ter nomes autoexplicativos, como **origem** e **destino**, ajudando a criação de métodos de acesso, getters, que sejam intuitivos para os desenvolvedores, como **getOrigem()**. Além disso, ao optar por essa solução, será mais fácil de manter o limite máximo, pois o objeto conterá no máximo duas instâncias de aeroporto.

Com esses argumentos é nítido perceber que é mais limpa a criação de dois atributos do que a criação de uma lista, conforme pode ser visualizado na **Listagem 4**. Vale ressaltar que isso não é uma regra, mas existe um entendimento que quando o limite superior for no máximo três instâncias, é válido pensar em estabelecer essa regra de atributos individuais do que uma lista em seu sistema.

Listagem 4. Exemplo de código da classe Rota a partir da associação da **Figura 4**.

```
01    public class Rota {
02       private Aeroporto origem;
03       private Aeroporto destino;
04
05       //Outros atributos declarados
06
07       public Aeroporto getOrigem(){
08          return origem;
09       }
10
11       public Aeroporto getDestino(){
12          return destino;
13       }
14
15       //Outros métodos declarados
16    }
```

Associações entre classes – Conectividade

As características apresentadas anteriormente (navegabilidade e multiplicidade) até podem ser analisadas de forma independente em cada uma das direções da associação. A Conectividade, por sua vez, para ser avaliada corretamente, deve possuir uma análise dupla, observando os dois sentidos, mesmo para associações unidirecionais, sendo então necessário estabelecer a multiplicidade do sentido não exibido. E como existem duas opções de multiplicidade, um ou muitos, podemos, a partir disso, estabelecer quatro formas de conectividade, a saber:

- OneToOne – Um para um;
- OneToMany – Um para muitos;

• ManyToOne – Muitos para um;
• ManyToMany – Muitos para Muitos.

A leitura dessa característica deve ser realizada em linha, ou seja, deve ser interpretada a partir do seguinte padrão: *N instâncias da Classe Origem* ***To*** *M instâncias da Classe Destino*, onde o valor N representa o cardinal mais próximo da classe origem e o valor M representa o cardinal mais próximo da classe destino. Se a multiplicidade do cardinal for um (1), o valor a ser atribuído à frase será *one* (um). Se a multiplicidade do cardinal for asterisco (*), o valor a ser atribuído à frase será *many* (muitos).

Essa leitura é realizada de forma mais fácil em uma associação bidirecional, visto que todos os elementos necessários para a leitura, Classes e Cardinais, estão dispostos na associação. Por exemplo: vamos observar a associação entre as classes **Horario** e **Rota**, representada na **Figura 2**. Pensando que a classe Origem seja **Horario** e a classe Destino seja **Rota**, consideramos que a conectividade entre essas classes é do tipo *ManyToOne*. Assim, *existem muitos horários que dependem de uma rota*. Caso fosse o contrário, isto é, a classe Origem sendo **Rota** e a classe Destino sendo **Horario**, consideramos que esta conectividade é do tipo *OneToMany*. Assim, uma *rota pode gerar muitos horários*.

A leitura da conectividade se torna mais difícil de ser interpretada quando é necessário realizá-la sobre uma associação unidirecional. Observe como exemplo a **Figura 1**. É muito fácil interpretar o lado da classe Destino, pois existe o cardinal um ao lado da classe **Endereco**, sendo assim identificado como *One*, formando, deste modo, a metade final da cardinalidade (*?ToOne*). Mas e a metade inicial, o que a interpretação nos indica? One ou Many?

Uma das formas de facilitar a interpretação de uma associação unidirecional é observando o diagrama DER do banco de dados do projeto. Vamos supor, para facilitar a explicação desse ponto, que a classe Origem possui como referência no banco de dados uma tabela Origem. O mesmo para a classe **Destino**. Dito isso, se houver uma tabela associativa, uma tabela que possui colunas que representam as chaves primárias da tabela Origem e da tabela Destino, significa que a parte inicial da conectividade é do tipo *Many*. Caso a tabela Origem possua colunas que representam as chaves primárias da tabela Destino, ou o contrário, a tabela Destino possua colunas que representam as chaves primárias da tabela Origem, significa que a parte inicial da conectividade é do tipo *One*.

Ao analisar as tabelas do banco de dados que armazenam as informações de associação entre **Aeroporto** e **Endereco**, representada pela **Figura 1**, é provável que identifiquemos uma coluna na tabela *Aeroporto* que aponte para a chave primária da tabela *Endereco*, caracterizando assim uma conectividade do tipo *OneToOne*.

Antes de iniciarmos a explicação sobre a última característica de uma associação entre classes, a obrigatoriedade, é necessário descrever alguns conceitos sobre construtores de classes do paradigma da programação orientada a objetos.

Construtores de classes

O construtor de uma classe no paradigma orientado a objetos possui algumas características próprias, a saber:
• É um método que possui o mesmo nome da classe, respeitando o padrão CamelCase;
• Não possui um valor de retorno, nem mesmo **void**;
• Pode possuir parâmetros (argumentos);
• Toda classe deve possuir ao menos um construtor;
• Se não for escrito um construtor em uma classe, a JVM irá prover um construtor sem parâmetros, chamado de construtor default.

A fim de compreender melhor essas características, vamos verificar uma das hierarquias de classes do sistema de Bilhetes Aéreos, a hierarquia da classe **Pessoa**, conforme mostra a **Figura 5** (caso não seja do seu conhecimento, a associação com o triângulo branco, entre **Funcionario**, **Passageiro** e **Pessoa** representa uma Herança).

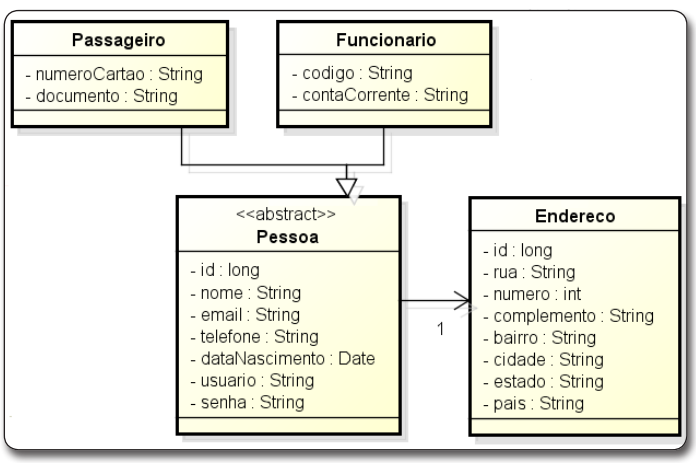

Figura 5. Hierarquia entre classes Pessoa e Endereco

Assim, é possível afirmar que a classe **Passageiro** é filha da classe **Pessoa**, e como não existe indicação na **Figura 5** que a classe **Pessoa** é filha de alguém, sabemos, pela definição da linguagem Java, que ela é filha de **Object**, pois todas as classes são filhas da classe **java.lang.Object**, direta ou indiretamente. Agora observe a seguinte linha de código:

```
Passageiro passageiro = new Passageiro();
```

Como já conhecido por todos, esse código realiza a invocação do construtor da classe **Passageiro**, e como esperado, essa invocação consiste na execução do método construtor sem parâmetros desta classe. Dito isso, vamos pensar um pouco sobre o código do método construtor... Na sua opinião, quais são as linhas de código que estarão presentes (lembre-se que a classe **Passageiro** está inserida dentro de uma hierarquia)?
1. Apenas as linhas de código do método construtor da classe **Passageiro**?

2. As linhas de código do método construtor da classe **Passageiro** e as linhas de código do método construtor da classe **Pessoa**, o qual será invocado pelo construtor da classe **Passageiro**?
3. As linhas de código do método construtor da classe **Passageiro**, as linhas de código do método construtor da classe **Pessoa**, o qual será invocado pelo construtor da classe **Passageiro**, e as linhas de código do método construtor da classe **Object**, o qual será invocado pelo construtor da classe **Pessoa**?

A opção correta é a de número três. O diagrama de sequência da **Figura 6** explicita essa iteração entre os construtores. Conforme destacado, note que o código analisado chama o construtor da classe **Passageiro**, que chama o construtor da classe **Pessoa**, que por fim chama o construtor da classe **Object**. Estas chamadas entre os construtores se dão através do método **super()**. Caso este não seja inserido na primeira linha do método construtor, será inserido automaticamente pelo Java. E se for colocado após a primeira linha do método construtor, será gerado um erro de compilação, conforme a linha 8 da **Listagem 5**.

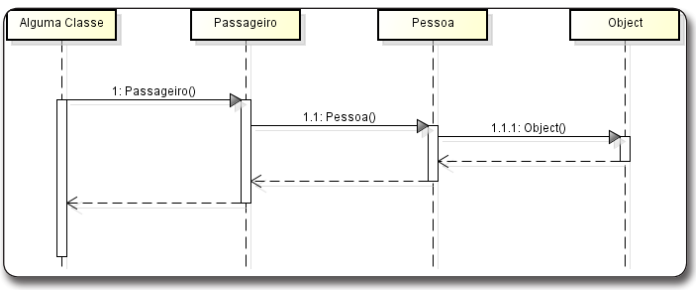

Figura 6. Diagrama de sequência da chamada ao construtor de Passageiro

Listagem 5. Exemplo de código com erro de compilação pela chamada errada ao método super().

```
01    public class Passageiro extends Pessoa {
02
03      private String documento;
04      private String numeroCartao;
05
06      public Passageiro(){
07        documento = "abc";
08        super(); //ERRO DE COMPILAÇÃO
09      }
10
11      //Outros métodos
12    }
```

Associações entre classes – Obrigatoriedade

Por fim, analisaremos a característica de Obrigatoriedade, que representa a quantidade mínima que uma classe origem deve conter de objetos da classe destino. Para essa característica, a observação principal será sobre o limite inferior da associação, pois ela informa o valor da quantidade mínima. Observando a **Figura 4**, qual seria a quantidade mínima de instâncias da classe **Aeroporto** que uma instância da classe **Rota** deveria conter? A quantidade mínima, e nesse caso a quantidade máxima também, é de dois Aeroportos.

Para que essa característica seja implementada em uma classe, devemos atacar um ponto em comum que TODAS as classes, não importa qual, passem ao menos uma vez durante o seu ciclo de vida. E esse ponto são os Construtores. Exemplificando, para criar uma nova instância da classe **Rota**, esta deve obrigatoriamente conter duas instâncias da classe **Aeroporto** válidas. Diante disso, um bom lugar para obrigar o desenvolvedor a informar estas instâncias de Aeroportos é em parâmetros de um construtor de **Rota**.

A partir disso, um erro muito comum que ocorre ao começar a trabalhar com construtores é criar um construtor que possui parâmetros para todos os atributos da classe. Não é uma boa prática fazer isso. Como exemplo, pense na construção, literalmente, de uma casa no mundo real. Quais são os "objetos" imprescindíveis para realizar essa construção? Você pode estar pensando em água encanada, energia elétrica, materiais de construção, etc., mas será que ao iniciar a construção de uma casa, esses itens realmente são imprescindíveis?

O objeto realmente necessário é o terreno. Se você não tiver um terreno, não adianta ter água, energia ou materiais, pois não sairá uma construção. Esse é um objeto básico, conforme demonstra a associação representada na **Figura 7**. Dessa forma, ao passar um objeto **Terreno** como parâmetro para o construtor da classe **Casa**, é atendida essa obrigatoriedade básica da associação, conforme demonstrado nas linhas 4 a 6 da **Listagem 6**.

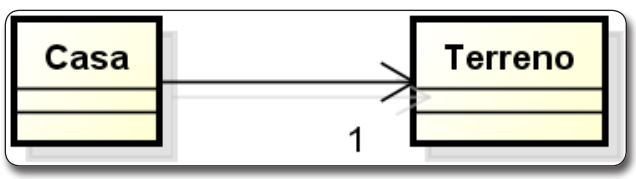

Figura 7. Representação da associação entre as classes Casa e Terreno

Listagem 6. Código da classe Casa, com o construtor recebendo uma instância de Terreno.

```
01    public class Casa {
02      private Terreno terreno;
03
04      public Casa(Terreno terreno){
05        this.terreno = terreno;
06      }
07
08      public Terreno getTerreno(){
09        return terreno;
10      }
11
12      //Outros métodos...
13
14    }
```

Uma última observação é necessária a respeito de associações obrigatórias bidirecionais. Conforme já mencionado, uma associação bidirecional permite a ambas as classes se "conhecerem", e para que seja possível esse conhecimento, ambas devem possuir atributos internos que possuam instâncias da outra classe.

Portanto, se há uma associação bidirecional de 1 para 1 entre **A** e **B**, significa que **A** possui uma instância de **B** e **B** possui uma instância de **A**.

Porém, apenas possuir os atributos não significa que uma classe esteja estabelecendo, em tempo de execução, uma associação bidirecional com outra. Para que isso ocorra é necessário que a instância da classe **B** seja passada para a classe **A**. Caso essa associação seja obrigatória, lembre-se que a instância deve ser passada via parâmetro de um construtor, como no código:

```
public A(B b){
  this.b = b;
  b.setA(this);
}
```

Em seguida, no código desse construtor, devemos desenvolver linhas que estabeleçam as ligações da associação. O primeiro passo é realizar a atribuição do parâmetro do construtor (**b**) ao atributo interno que representa a classe Destino (**this.b**). A linha **this.b = b;** cria o sentido da associação da classe **A** para a classe **B**.

O sentido contrário da associação, isto é, da classe **B** para a classe **A**, também deve ser implementado nesse construtor. Isto pode ser feito através do método setter da classe **B**, que recebe uma instância da classe **A**. Deste modo, a linha **b.setA(this);** informa para a instância da classe **B**, através do método setter, a nova instância de **A** que acaba de ser criada. Esse método setter muito provavelmente irá atribuir essa nova instância ao atributo interno da classe **B** que representa a classe **A**.

Um exemplo mais concreto de uma associação bidirecional obrigatória pode ser criado se mudarmos a associação representada pela **Figura 7** para uma associação bidirecional, em que, tanto a instância de **Casa** quanto a instância de **Terreno** devem se conhecer. Diante disso, para que ambos os lados da associação possam estar contemplados em tempo de execução, é necessário, dentro do construtor da classe **Casa**, adicionar a seguinte linha:

```
this.terreno.setCasa(this);
```

Esse código deve ser inserido entre as linhas 5 e 6 da **Listagem 6** e tem como objetivo acessar a instância de **Terreno** e invocar o método **setCasa()**, passando a nova instância de **Casa**, ou seja **this**, para a instância de **Terreno**.

Conceitos de camadas de software, o padrão MVC

A partir do Princípio da Responsabilidade Única é possível constatar que um software deve ser composto por várias partes. Você acha interessante que uma classe contenha todos os itens de um software? Ou seja, Iteração com o usuário, regras de negócio, interface para o banco de dados e muitas outras tarefas?

Não é desejável que isso aconteça, pois a manutenção desse código certamente será muito mais custosa. Uma forma de melhorar a manutenção de um software, além de projetá-lo com base em diferentes classes, é a sua divisão em camadas, o que foi definido por Dr. Trygve Reenskaug, na década de 1970. Essas camadas tentam imitar os anéis de uma cebola. Ou seja, ao cortar a cebola, observamos que o seu interior é particionado em anéis (ou camadas), e quanto mais interna a camada, mais anéis tiveram de ser removidos.

O conceito de camadas adotado no desenvolvimento de software se assemelha a isso e é bastante aceito entre os desenvolvedores, sendo o modelo mais comum o chamado de 3-Tier (três camadas). Este modelo foi planejado pensando no conceito das camadas mais básicas existentes para a criação de um software, que são as camadas Model, View e Controller, popularmente conhecidas como MVC (vide **Figura 8**).

Dentre as três camadas, a mais externa é a View. Esta é a camada que o usuário terá acesso, seja na forma de interfaces gráficas para desktop, web ou telas sensíveis ao toque. Além disso, ela pode ter regras de negócio próprias como, por exemplo, ao selecionar um valor em um campo, este abrirá novos campos a serem preenchidos pelo usuário.

A camada Model representa a abstração do mundo real em um software, ou seja, são as classes relacionadas ao escopo do projeto. Na fase de análise de um software, comumente por questões de custo, é realizada apenas a representação da camada Model em um diagrama de classes UML, não sendo criados diagramas de classes completos para as demais camadas, com exceção para trechos críticos destas como, por exemplo, a criação de uma factory para uma hierarquia de classes ou a representação de classes componentes para uma camada View.

Atualmente, em grande parte dos frameworks disponíveis, as instâncias das classes da camada Model possuem uma característica singular, pois "transitam" por outras camadas com informações per-

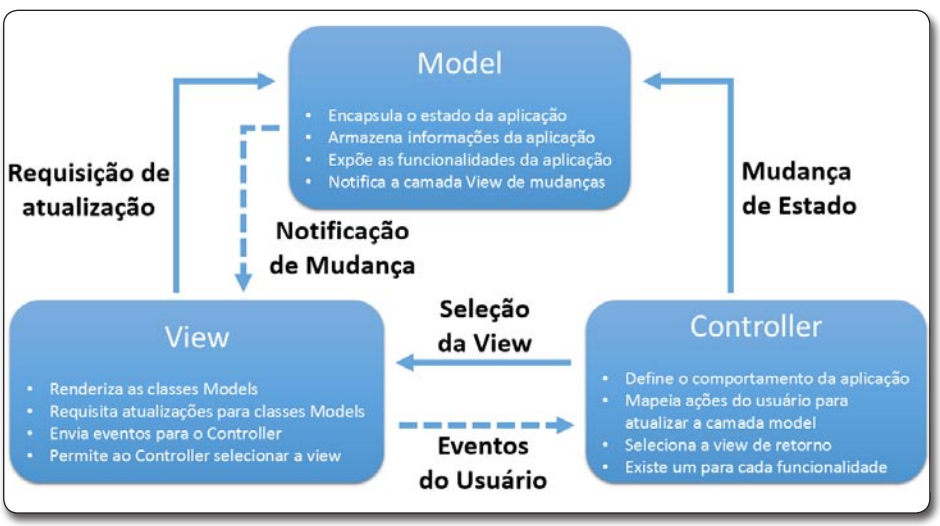

Figura 8. Diagrama apresentando as camadas Model, View e Controller

tinentes ao sistema. No ciclo de vida da execução de um software, a camada View preenche instâncias de classes da camada Model com dados fornecidos pelos usuários. Essas instâncias, após algum evento do usuário, como o clique em um botão, são enviadas através da camada View para outras camadas mais centrais do software, como a camada Controller.

O retorno das informações dessas camadas mais centrais, por sua vez, exerce influência sobre a camada View. Por exemplo, ao clicar em um link haverá o processamento em camadas mais internas e, a partir de mudanças realizadas no estado da aplicação, a página (camada View) pode passar por mudanças para representar esse novo estado ou mesmo ocorrer o redirecionamento para a exibição de outra página.

Já a camada Controller possui a responsabilidade de definir o comportamento da aplicação, se comunicando com as classes da camada View conforme a necessidade. Um aspecto que pode estar dentro de uma classe da camada de Controller é a validação das regras de negócio realizada sobre as instâncias das classes da camada Model. Assim, é possível realizar verificações sobre estas instâncias a fim de obter dados inválidos ou incompletos como, por exemplo, um CEP não preenchido ou que não existe. Por causa dessa característica, é uma prática de mercado modificar o nome desta camada para Business Controller.

Ademais, a camada de Controller deve possuir uma classe para cada classe da camada Model, aumentando dessa forma o reuso de validações, ao dispor em um único lugar as validações correspondentes a cada classe da camada Model.

Outro aspecto que merece destaque é que no modelo MVC não há o armazenamento explícito de informações. Isto é realizado internamente, pela camada Model e suas instâncias. Por este motivo, grandes empresas de software passaram a adicionar uma nova camada, que representa a camada de persistência do software ou a sua ligação com o banco de dados. Essa camada possui o nome de *Data Access Object*, e é mais conhecida por suas classes internas, que são chamadas de classes DAO. Estas classes têm a responsabilidade de prover a inserção/atualização/remoção/recuperação de informações do banco de dados. Com a definição dessa camada, o MVC passou a ter uma extensão, chamada de MVC-DAO.

Com um bom conhecimento das regras de negócio que existem em um projeto, é possível estabelecer um diagrama de classes UML conciso e coerente com o que o domínio necessita. Para isso, no entanto, além de um bom conhecimento sobre o domínio, também é importante saber projetar um software tendo como base princípios como o da responsabilidade única, analisando se as classes elaboradas estão satisfazendo as condições indicadas.

Autor

Everson Mauda
mauda@mauda.com.br – www.mauda.com.br
Bacharel em Ciência da Computação pela Universidade Federal do Paraná (UFPR), Mestre em Informática pela Pontifícia Universidade Católica do Paraná (PUC-PR). Analista e Desenvolvedor Java desde 2004, trabalhou em várias empresas multinacionais como GVT, HSBC, SERPRO e SITA. Baterista por hobby desde 2002 e entusiasta por ensinar aos outros desde que se conhece por gente.

Programando com o Java Collections Framework

Conheça e adote os vários tipos de coleções Java em seus sistemas

A implementação de coleções na linguagem Java, antes do Java 2 (JDK 1.2), incluía poucas classes, como **Vector** e **HashTable**, e não tinha a forma e organização de um framework. Para o lançamento do Java 2, no entanto, foram criadas novas classes e interfaces para organizar os mais diversos tipos de coleções agora existentes, originando assim o *Java Collections Framework*.

Com este framework, o Java ganhou componentes comuns – reutilizáveis para diversas aplicações – que usam coleções de dados para atender as mais variadas finalidades de coleções, como listas, conjuntos, pilhas, vetores, entre outros.

Algum tempo depois, já no Java 5, foram adicionados diversos recursos ao framework de coleções devido à criação de **Generics**, o que permitiu criar coleções "tipadas", isto é, coleções cujos elementos são restritos a um determinado tipo em vez de **Object**. Além disso, com o Java 5, foi criado também o laço "for melhorado" (*enhanced for*), possibilitando percorrer qualquer coleção facilmente.

Passados alguns anos, com o lançamento do Java 7, mais melhorias foram realizadas no framework de coleções e foi criado o operador diamante, que simplifica a instanciação de objetos (coleções) usando Generics.

No Java Collections Framework, através de interfaces, são definidos e organizados os mais diversos tipos de coleções, de forma que são oferecidas classes que implementam os mais variados padrões de coleções, como mostra na **Figura 1**.

A interface **java.util.Iterable** está no topo da hierarquia de interfaces do framework e sua principal função é definir que qualquer coleção filha seja iterável pelo laço de repetição "for melhorado". Para isso, ela disponibiliza alguns métodos, sendo **iterator()** o método principal, que retorna um objeto que implementa a interface **Iterator**.

Através da interface **java.util.Iterator** pode-se percorrer qualquer coleção facilmente, pois além da vantagem de se usar o "for melhorado", é uma interface que define alguns métodos que permitem percorrer qualquer tipo de coleção. Os principais métodos são:

- **boolean hasNext():** retorna **true** se existem mais elementos a serem acessados na coleção vinculada a esse **Iterator**;
- **E next():** retorna o próximo elemento disponível na coleção vinculada a esse **iterator**;
- **void remove():** remove da coleção o último elemento acessado através desse **Iterator**.

Observe que tanto **Iterable** como **Iterator** fazem uso de **Generics**. Sendo assim, em vez de apenas criar coleções para armazenar objetos do tipo **Object**, é possível que sejam criadas coleções "tipadas", restringindo o tipo de objeto que a coleção pode armazenar (este recurso é denotado pelo **<E>** na **Figura 1**), aumentando assim o controle da coleção pelo programador, uma vez que esse recurso permite descobrir erros de programação em tempo de compilação.

Como resultado do uso de **Generics** na interface **Iterable**, a sua interface filha **java.util.Collection** também pode ser parametrizada com um tipo usando **Generics**. Como **Collection** é a interface mãe de quase todos os tipos de coleções, as funcionalidades conferidas a ela através de **Generics** são automaticamente herdadas

Fique por dentro

Este artigo apresenta os principais tipos de coleções disponíveis no framework de coleções da linguagem Java, que inclui listas, vetores, mapas, pilhas, conjuntos, entre outros. Através de uma hierarquia de interfaces, a linguagem Java implementa e disponibiliza classes que definem os mais variados tipos de coleções a fim auxiliar no desenvolvimento de aplicações.

As classes do framework de coleções do Java livram o programador da necessidade de implementar complicadas estruturas de dados para gerenciar coleções de objetos facilitando, portanto, o desenvolvimento de aplicações, acelerando o desenvolvimento e simplificando o intercâmbio de dados ao viabilizar uma implementação mais rápida, eficiente e padronizada com o reuso de coleções que são amplamente adotadas no mundo Java.

por todas as coleções filhas. A interface **Collection** tem a função de definir muitas das funcionalidades comuns a quase todas as coleções e será por ela que iniciaremos esse artigo.

A interface Collection

A maioria das coleções em Java derivam da interface **java.util.Collection**, que define métodos comuns entre todas elas com o objetivo de padronizar muitas das operações básicas suportadas em cada tipo de coleção, a saber:

- **boolean add(E e):** adiciona um elemento à coleção;
- **boolean addAll(Collection<? extends E> c):** adiciona à coleção todos os elementos da coleção passada como parâmetro;
- **void clear():** Esvazia essa collection, mas não elimina da memória os objetos que ela referenciava, a não ser que não haja mais nenhuma outra referência para os mesmos;
- **boolean contains(Object o):** retorna **true** se a coleção contém o elemento informado como parâmetro;
- **boolean containsAll(Collection<?> c):** retorna **true** se a coleção contém todos os elementos da coleção informada como parâmetro;
- **boolean isEmpty():** retorna **true** se a coleção não tem elementos;
- **Iterator<E> iterator():** retorna um **Iterator** para percorrer os elementos da coleção. Uma coleção pode ser percorrida por um **Iterator** usando o **for** melhorado;
- **boolean remove(Object o):** remove da coleção o elemento informado como parâmetro;
- **boolean removeAll(Object o):** remove da coleção todos os elementos da coleção informada como parâmetro;
- **int size():** Retorna a quantidade de elementos da coleção;
- **Object[] toArray():** converte essa coleção para um array de **Object**.

Como a interface **Collection** define o tipo mais alto na hierarquia de coleções, as suas interfaces filhas definem tipos mais especializados, como as interfaces **List**, **Set**, **SortedSet**, **Queue**, **Deque**, entre outras, que automaticamente herdam todos os seus métodos. As principais interfaces filhas de **Collection** serão apresentadas a seguir.

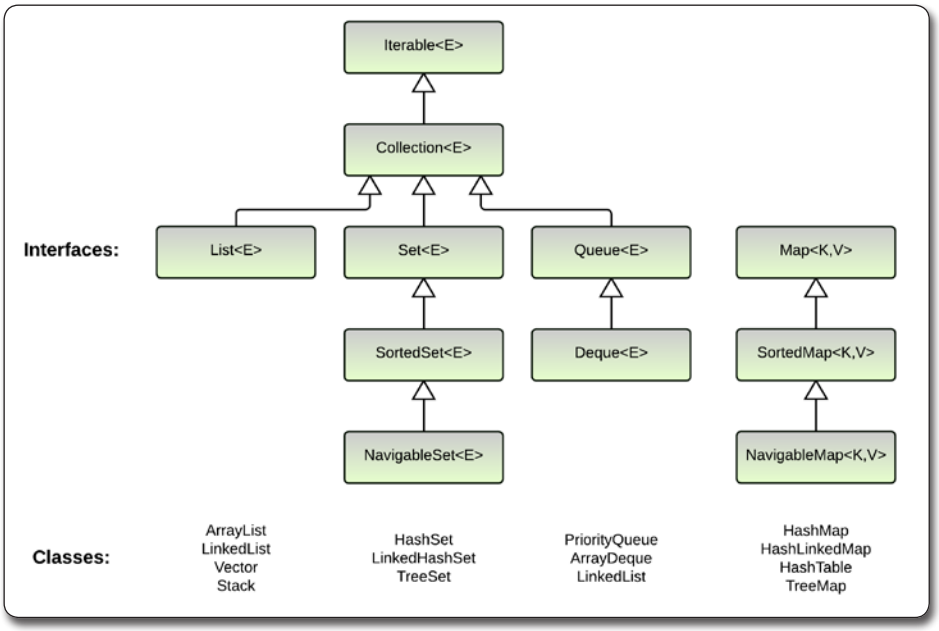

Figura 1. Estrutura das principais interfaces e classes do framework de coleções

A interface List

A interface **java.util.List** tem o objetivo de definir coleções que têm a função de servirem como arrays de tamanho dinâmico, de forma que cada elemento seja acessível por um índice, que é a posição do elemento no array. Além disso, novos elementos podem ser criados ou removidos em qualquer posição e pode haver elementos duplicados.

As principais classes que descendem da interface **List** são **ArrayList**, **Vector**, **LinkedList** e **Stack**, sendo seus principais métodos listados a seguir (observe que foram omitidos os métodos herdados de **Collection**):

- **void add(int index, E element):** Insere o elemento especificado na posição passada como parâmetro;
- **E get (int index):** Retorna o elemento de uma posição especificada;
- **Int indexOf(Object o):** Retorna a posição do objeto passado como parâmetro. Se o objeto não existe, retorna -1;
- **E remove(int index):** Remove o elemento presente no índice indicado como parâmetro;
- **E set(int index, E element):** Faz com que o objeto informado como parâmetro ocupe a posição informada no parâmetro **index**. Se a posição já estiver ocupada, substitui o valor presente;
- **void sort(Comparator<? super E> c):** Ordena o **List** de acordo com o critério implementado no **Comparator** passado como parâmetro;
- **List<E> subList(int fromIndex, int toIndex):** Retorna um **List** que contém a sublista de elementos presentes entre os índices indicados.

Conhecendo as classes ArrayList e Vector

A classe **java.util.ArrayList** representa uma lista dinâmica não protegida de acesso concorrente, ou seja, não apresenta sincronização na invocação de seus métodos por *threads* concorrentes. A classe **java.util.Vector** se torna a alternativa correta quando se deseja uma lista que seja acessada de forma sincronizada por *threads* concorrentes, entretanto tem um custo maior de processamento nas suas operações devido à sincronização.

Um **ArrayList** contém um array interno de forma a simular uma lista com tamanho dinâmico para o usuário. No entanto, quando a última posição disponível for

preenchida, **ArrayList** aumenta o tamanho do vetor interno em 50%. A classe **Vector** é praticamente igual a **ArrayList**, quase tendo em comum todas as assinaturas de métodos, porém **Vector** inclui sincronização nas assinaturas e mais métodos para gerenciar a capacidade do array interno.

Outra diferença importante entre **ArrayList** e **Vector** é que quando o array interno de um **Vector** fica cheio, este dobra o tamanho do array, demorando assim mais tempo para encher o array interno novamente. Os principais construtores e métodos da classe **ArrayList** (e **Vector**) são:

- **ArrayList():** Constrói um **ArrayList** sem elementos que contém inicialmente 10 posições livres no array interno;
- **ArrayList(Collection<? extends E> c):** Constrói um **ArrayList** com todos os elementos da coleção passada como parâmetro;
- **ArrayList(int initialCapacity):** Constrói um **ArrayList** sem elementos que contém inicialmente o número indicado de posições livres no array interno. Esse construtor permite um uso melhor da memória quando é possível se ter uma previsão da quantidade de elementos que o **ArrayList** necessitará;
- **boolean add(E e):** adiciona o elemento passado como parâmetro ao final do **ArrayList**;
- **void add(int index, E element):** adiciona o elemento passado como parâmetro na posição indicada na lista por **index**. Se esta posição já estiver ocupada, desloca todos os elementos seguintes para acomodar o novo elemento;
- **boolean addAll(Collection<? extends E> c)**: Adiciona todos os elementos da collection passada como parâmetro ao final do **ArrayList**;
- **boolean addAll(index, Collection<? extends E> c):** Adiciona todos os elementos da collection passada como parâmetro, iniciando a inserir na posição indicada pelo parâmetro **index**;
- **void clear():** Esvazia essa collection, mas não elimina da memória os objetos que ela referenciava, a não ser que não haja mais nenhuma referência para esses objetos;
- **Object clone():** Retorna uma cópia do **ArrayList**, de forma que a cópia contenha novas referências aos objetos apontados por esse **ArrayList**;
- **boolean contains(Object o):** retorna **true** se o **ArrayList** contém o elemento informado como parâmetro;
- **void ensureCapacity(int minCapacity):** se o array interno tem capacidade menor que **minCapacity**, aumenta o tamanho dele até ficar igual a **minCapacity**. Esse método evita a fragmentação da memória, uma vez que diminui a quantidade de vezes que internamente o array é aumentado;
- **E get(int index)**: Retorna o elemento presente no índice indicado.
- **int indexOf (Object o)**: Retorna a posição de um elemento no **ArrayList** dada a referência para o mesmo;
- **boolean isEmpty():** retorna **true** se o **ArrayList** não contém elementos;
- **Iterator<E> iterator():** retorna um **Iterator** para percorrer os elementos do ArrayList;
- **boolean remove(Object o):** remove do **ArrayList** o elemento informado como parâmetro;
- **E set(int index, E element)**: Faz com que o objeto informado como parâmetro ocupe a posição informada no parâmetro **index**. Se a posição já estiver ocupada, substitui o elemento que já estava presente;
- **int size():** Retorna a quantidade de elementos do **ArrayList**;
- **void sort(Comparator<? super E> c)**: Ordena os elementos do **ArrayList** de acordo com o **Comparator** indicado;
- **List<E> subList(int fromIndex, int toIndex):** Retorna uma sublista com todos os elementos que estão entre os índices informados;
- **Object[] toArray():** Converte esse **ArrayList** em um **array** de **Object**;
- **void trimToSize()**: Compacta o **array** interno descartando posições não ocupadas, otimizando assim o uso da memória.

Observe que como **ArrayList** é filha de **Collection**, ela automaticamente herda todos os seus métodos, que são: **size()**, **isEmpty()**, **contains()**, **iterator()**, **toArray()**, **add()**, **remove()**, **addAll()**, **clear()**, entre outros.

Um **ArrayList** pode ser criado de várias formas diferentes. A primeira delas que vamos mostrar é a mais antiga, disponível desde o Java 2, como mostrado a seguir:

```
ArrayList lista = new ArrayList();
```

Essa é a forma padrão de se criar um **ArrayList**, sendo, neste exemplo, criado um **ArrayList** com nome **lista** que armazena qualquer objeto descendente da classe **Object**.

Depois disso, com a criação de **Generics** no Java 5, foi adicionada uma nova opção de se instanciar ArrayLists, de forma que é especificado um tipo para o **ArrayList** usando **Generics**. Assim, o **ArrayList** só vai aceitar objetos daquele tipo, o tornando mais consistente e possibilitando detectar erros durante a compilação que só poderiam ser verificados em tempo de execução. No código a seguir é mostrado um exemplo de instanciação de **ArrayList** usando **Generics**. Observe que o **ArrayList** criado permite que sejam adicionados somente objetos do tipo **Integer**:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
```

Felizmente, com o lançamento do Java 7, foi facilitado o uso do operador diamante **<>** (*diamond*), de forma que não é mais necessário informar o tipo no lado direito da expressão, como expõe o código a seguir, que tem a mesma funcionalidade que o código anterior:

```
ArrayList<Integer> lista = new ArrayList<>();
```

Na **Listagem 1** é apresentado um exemplo de uso de **ArrayList**, onde um é criado usando o operador diamante (linha 5) e em seguida são criados seis objetos do tipo **Integer** (linhas 7 a 12). Logo após, os cinco primeiros números são adicionados ao **ArrayList** usando o método **add()** (linhas 14 a 18).

Listagem 1. Exemplo com ArrayList.

```
01. import java.util.ArrayList;
02. public class TesteArrayList {
03.  public static void main (String[] args) {
04.
05.  ArrayList<Integer> lista = new ArrayList<>();
06.
07.  Integer numero0 = new Integer(0);
08.  Integer numero1 = new Integer(1);
09.  Integer numero2 = new Integer(2);
10.  Integer numero3 = new Integer(3);
11.  Integer numero4 = new Integer(4);
12.  Integer numero5 = new Integer(5);
13.
14.  lista.add(numero0);
15.  lista.add(numero1);
16.  lista.add(numero2);
17.  lista.add(numero3);
18.  lista.add(numero4);
19.
20.  System.out.println(lista);
21.  lista.remove(2);
22.  System.out.println(lista);
23.  System.out.println(lista.indexOf(numero2));
24.  lista.set(0,  numero2);
25.  System.out.println(lista);
26.  lista.add(0, numero5);
27.  System.out.println(lista);
28.  }
29. }
```

Feito isso, na linha 20 é impresso o objeto **lista**, obtendo como resultado: [0, 1, 2, 3, 4], que são os elementos do **ArrayList** na ordem em que foram inseridos. Continuando a execução, na linha 21 é removido o elemento de índice 2 (que corresponde ao objeto **numero2**) e na linha 22, o **ArrayList** é impresso novamente, obtendo: [0, 1, 3, 4]. Depois disso, na linha 23 é invocado **lista .indexOf(numero2)**, entretanto, como **numero2** não faz mais parte desse **ArrayList**, é impresso "-1" na tela.

Na linha 24, o elemento de índice 0 é substituído pelo objeto **numero2** usando o comando **lista.set(0, numero2)**, imprimindo [2, 1, 3, 4] na linha 25. Por fim, é adicionado o objeto **numero5** a posição 0, deslocando todos os outros elementos para as posições seguintes, imprimindo [5, 2, 1, 3, 4] na linha 27.

Como as classes **Vector** e **ArrayList** têm praticamente os mesmos métodos, o código declarado na **Listagem 2** é equivalente ao código na **Listagem 1**, porém usando **Vector** em vez de **ArrayList**.

Aplicando Comparator para ordenar um ArrayList

Um dos métodos mais importantes da classe **ArrayList** é o **void sort(Comparator<? super E> c)**. Este método tem a função de ordenar um **ArrayList** segundo o critério definido por uma classe que implementa a interface **java.util.Comparator**.

Quando a interface **Comparator** é implementada em uma classe filha, codifica-se o método **int compare(Object obj1, Object obj2)**. Este recebe como parâmetro dois objetos, ficando a cargo do programador especificar a forma de comparação entre eles, o que vai refletir na ordem com que os mesmos são dispostos no vetor final ordenado, sendo que o mesmo deve retornar zero quando os objetos forem iguais (implicando na mesma ordem entre ambos), retornar 1 quando o primeiro objeto é menor que o segundo (fazendo com que o primeiro objeto seja posicionado no vetor final antes do segundo), e -1 caso contrário. Na **Listagem 3** é apresentado um exemplo de **Comparator** que ordena números inteiros.

Listagem 2. Exemplo com Vector.

```
01. import java.util.ArrayList;
02. public class TesteVector {
03.  public static void main (String[] args) {
04.
05.  Vector<Integer> lista = new Vector<>();
06.
07.  Integer numero0 = new Integer(0);
08.  Integer numero1 = new Integer(1);
09.  Integer numero2 = new Integer(2);
10.  Integer numero3 = new Integer(3);
11.  Integer numero4 = new Integer(4);
12.  Integer numero5 = new Integer(5);
13.
14.  lista.add(numero0);
15.  lista.add(numero1);
16.  lista.add(numero2);
17.  lista.add(numero3);
18.  lista.add(numero4);
19.
20.  System.out.println(lista);
21.  lista.remove(2);
22.  System.out.println(lista);
23.  System.out.println(lista.indexOf(numero2));
24.  lista.set(0,  numero2);
25.  System.out.println(lista);
26.  lista.add(0, numero5);
27.  System.out.println(lista);
28.  }
29. }
```

Listagem 3. Exemplo de Comparator.

```
01. import java.util.ArrayList;
02. import java.util.Comparator;
03. public class IntegerComparator<E> implements Comparator<E> {
04.  public int compare (E obj1, E obj2) {
05.   return (((Integer) obj1)).compareTo((Integer)obj2);
06. }
```

Como **Comparator** usa **Generics**, **IntegerComparator** é definido também com **Generics** e compara dois objetos distintos de tipo genérico **E**. Neste caso, **E** representará o tipo **Integer**, que é necessário para nosso exemplo. Na implementação do método **compare()** (vide linha 5), ambos os objetos são convertidos para o tipo **Integer**, possibilitando assim invocar o **compareTo()**, já disponível na classe **Integer**. Note que ele se enquadra totalmente às nossas necessidades, pois compara dois objetos do tipo **Integer**, nos poupando o trabalho de reimplementar o mesmo comportamento.

Na **Listagem 4** é apresentado um trecho de código onde é realizada a ordenação de um **ArrayList** que contém elementos desordenados e repetidos usando o **Comparator** definido anteriormente.

Listagem 4. Exemplo de Comparator.

```
01. public static void main (String[] args) {
02.
03.  ArrayList<Integer> lista = new ArrayList<>();
04.
05.  Integer numero0 = new Integer(0);
06.  Integer numero1 = new Integer(1);
07.  Integer numero2 = new Integer(2);
08.  Integer numero3 = new Integer(3);
09.  Integer numero4 = new Integer(4);
10.  Integer numero5 = new Integer(5);
11.
12.  lista.add(numero4);
13.  lista.add(numero2);
14.  lista.add(numero3);
15.  lista.add(numero0);
16.  lista.add(numero1);
17.  lista.add(numero5);
18.  lista.add(numero3);
19.  lista.add(numero1);
20.
21.  System.out.println(lista);
22.
23.  lista.sort(new IntegerComparator<Integer>());
24.
25.  System.out.println(lista);
26. }
```

Neste exemplo, entre as linhas 5 e 10 são adicionados diversos elementos desordenados e alguns repetidos no **ArrayList**, que é impresso na linha 21. Como resultado temos a saída [4, 2, 3, 0, 1, 5, 3, 1]. Depois disso, a ordenação dos elementos ocorre na linha 21, usando o método **sort()**, onde é informado como parâmetro uma instância do **Comparator IntegerComparator**. Em seguida, na linha 25, é impresso novamente o **ArrayList**, agora ordenado, resultando em: [0, 1, 1, 2, 3, 3, 4, 5].

Percorrendo um ArrayList

Nos primeiros anos da linguagem Java, a forma de se percorrer um **ArrayList** era usando um laço de repetição que incrementa um índice, conseguindo assim acessar cada elemento do mesmo, como exemplificado na **Listagem 5**. Neste código, primeiramente é instanciado um **ArrayList** de nome **lista** (linha 1) e são inseridos seis elementos (linhas 3 a 8). Em seguida, nas linhas 17 e 18, o objeto **lista** é percorrido usando a variável contadora **cont**. Observe que o elemento corrente é sempre acessado usando o método **get()**, informando **cont** como índice.

Entretanto, essa forma de percorrer os elementos pode ser simplificada se considerarmos que na maioria das vezes não há necessidade de se usar um índice, a não ser para acessar o elemento corrente do **ArrayList**. Portanto, podemos usar **Iterator** para percorrer a lista mais facilmente desde o Java 2, como mostrado na **Listagem 6**.

Essa opção é completamente diferente da codificação apresentada na **Listagem 5**, pois agora cada elemento do **ArrayList** é acessado sem se conhecer a sua posição. Observando em detalhes, podemos ver que na linha 1 é obtido um **Iterator** usando o método **iterator()**. Dessa forma, o objeto **iter** permitirá percorrer de forma simples o **ArrayList** usando o laço de repetição **while** (linha 2), cuja condição de parada é dada pelo método **hasNext()** de **Iterator**, ou seja, enquanto houver um próximo elemento a ser acessado no **ArrayList**, o laço de repetição continua.

Listagem 5. Percorrendo um ArrayList.

```
01. ArrayList<Integer> lista = new ArrayList<>();
02.
03. Integer numero0 = new Integer(0);
04. Integer numero1 = new Integer(1);
05. Integer numero2 = new Integer(2);
06. Integer numero3 = new Integer(3);
07. Integer numero4 = new Integer(4);
08. Integer numero5 = new Integer(5);
09.
10. lista.add(numero0);
11. lista.add(numero1);
12. lista.add(numero2);
13. lista.add(numero3);
14. lista.add(numero4);
15. lista.add(numero5);
16.
17. for (int cont = 0; cont < lista.size(); cont++)
18.  System.out.print(lista.get(cont));
```

Listagem 6. Percorrendo um ArrayList com Iterator.

```
01. Iterator<Integer> iter = lista.iterator();
02. while(iter.hasNext())
03.  System.out.print(iter.next());
```

Na linha 3 o elemento corrente é acessado utilizando-se o método **next()**, também de **Iterator**, permitindo então varrer todos os elementos do **ArrayList**. Entretanto, com o lançamento do Java 5, foi introduzido o comando "for melhorado". A partir dele temos mais uma opção para se percorrer o **ArrayList**, mas usando uma sintaxe ainda mais simples, como expõe o código a seguir.

```
01. for (Integer valor : lista)
02.  System.out.print(valor);
```

Observe que com o uso de **Generics** em todos os exemplos, não é necessário realizar nenhum **cast**, pois o tipo **Integer** é usado diretamente. Na linha 1 observa-se que a variável **valor** sempre vai conter o elemento corrente, visto que ela é alimentada pelo "for melhorado".

Conhecendo a classe LinkedList

Disponível desde o Java 2, a classe **java.util.LinkedList** é outra filha de **List** e representa um tipo de coleção que pode ser muito útil, pois sua funcionalidade é semelhante a uma lista de tamanho dinâmico, porém, internamente é usada uma lista dinâmica (em vez de um **array**) de forma que cada elemento da lista contém uma referência ao elemento anterior e ao próximo.

Embora **LinkedList** use mais memória que **ArrayList**, **LinkedList** apresenta melhor performance em algumas operações, como no caso da inserção e remoção de elementos – vantagem que é dada pela lista dinâmica interna. Além disso, a classe **LinkedList**

implementa a interface **Deque**, que disponibiliza métodos para manipular o início e o fim da lista. Seus principais métodos e construtores são:

• **LinkedList():** Constrói uma lista vazia.
• **LinkedList(Collection<? extends E> c)**: Cria uma **LinkedList** com todos os elementos da coleção informada como parâmetro. Os elementos são inseridos na ordem com que eles são retornados pelo **Iterator** da coleção informada;
• **boolean add(E e):** Adiciona o objeto **e** ao fim da lista;
• **void add(int index, E element):** Insere o objeto indicado na posição informada;
• **boolean addAll(Collection<? extends E> c)**: Adiciona todos os elementos da coleção passada como parâmetro no final do **LinkedList**;
• **void clear():** Esvazia essa collection, mas não elimina da memória os objetos que ela referenciava, a não ser que não haja mais nenhuma outra referência para esses objetos;
• **Object clone():** Retorna uma cópia do **LinkedList**, de forma que essa cópia contenha novas referências aos objetos apontados pelo **LinkedList**;
• **boolean contains(Object o):** Retorna **true** se o **LinkedList** contém o elemento informado como parâmetro;
• **Iterator<E> descendingIterator():** Retorna um **Iterator** que percorre a lista na ordem contrária;
• **E element():** Retorna a cabeça da lista (primeiro elemento), mas não o remove;
• **E get(int index):** Retorna o elemento presente na posição indicada por parâmetro;
• **E getFirst():** Retorna o primeiro elemento da lista;
• **E getLast():** Retorna o último elemento da lista;
• **int indexOf(Object o)**: Retorna o índice do elemento indicado. Se não for encontrado, retorna -1;
• **boolean offer(E e):** Adiciona o objeto passado como parâmetro como sendo a cauda da lista (último elemento);
• **linkFirst(E e):** Define o objeto informado como sendo o primeiro elemento da lista;
• **linkLast(E e):** Define o objeto informado como sendo o último elemento da lista;
• **linkBefore(E e, Node<E> succ):** Insere o objeto informado logo antes do elemento também informado por parâmetro;
• **E peek():** Acessa o primeiro elemento da lista, mas não o remove;
• **E poll():** Acessa e remove a cabeça da lista;
• **E pop():** Remove o elemento que está no início da lista;
• **void push(E e):** Insere um elemento no início da lista, deslocando os demais;
• **boolean remove(Object o)** (Definido em **Collection**): Remove a primeira ocorrência do objeto indicado na lista. Usa o método **equals()** para encontrar o elemento indicado;
• **E remove(int index):** Remove o elemento com o índice especificado por parâmetro;
• **E removeFirst(int index):** Remove e retorna o primeiro elemento da lista;
• **E removeLast(int index):** Remove e retorna o último elemento da lista;
• **E set(int index, E element):** Redefine o elemento da posição indicada;
• **int size():** Retorna a quantidade de elementos do **LinkedList**;
• **Object[] toArray():** Converte o **LinkedList** em um **array** de **Object**.

É importante saber que **LinkedList** não possui sincronização em seus métodos, portanto, se necessário, deve-se adicionar sincronização externa à chamada destes métodos. O uso de **LinkedList** ocorre de forma muito semelhante a **ArrayList**, porém algumas facilidades podem ser percebidas em função de termos internamente uma lista dinâmica com cabeça (elemento inicial) e cauda (elemento final), como notado na **Listagem 7**.

Listagem 7. Exemplo com LinkedList.

```
01. import java.util.LinkedList;
02. public class TesteLinkedList {
03.   public static void main(String[] args) {
04.     LinkedList<Integer> lista = new LinkedList<>();
05.     lista.add(41);
06.     lista.add(31);
07.     lista.add(67);
08.     lista.add(10);
09.     lista.add(99);
10.     System.out.println(lista);
11.     lista.removeFirst();
12.     lista.removeLast();
13.     System.out.println("   " + lista);
14.     lista.addFirst(0);
15.     lista.addLast(100);
16.     System.out.println(" " + lista);
17.   }
18. }
```

Neste código, inicialmente são adicionados cinco elementos ao **LinkedList** (linhas 5 a 9). Em seguida, ele é impresso na tela (linha 10), obtendo como resultado: [41, 31, 67, 10, 99]. Observe que a ordem dos elementos é a mesma na qual foram inseridos. Como se trata de um **LinkedList**, a cabeça da lista corresponde ao elemento 41 (primeiro elemento que foi inserido) e a cauda corresponde ao elemento 99 (último elemento que foi inserido). Em seguida, nas linhas 11, 12 e 13 são removidas a cabeça e a cauda da lista e é impresso na tela o resultado: [31, 67, 10]. Para completar, é inserido o elemento 0 no início da lista e 100 no final. O resultado desse processamento é apresentado na tela quando a linha 16 é chamada: [0, 31, 67, 10, 100].

Conhecendo a classe Stack

A classe **java.util.Stack** é mais uma descendente da interface **List** e tem a função de representar um outro tipo de coleção: as pilhas. Ela contém todos os métodos de sua superclasse **Vector** e, adicionalmente, pode ser usada como uma pilha. Como a classe **Stack** representa uma pilha LIFO (last-in-first-out, ou seja, último a entrar, primeiro a sair), ela apresenta métodos para gerenciar uma

pilha de forma que novos elementos são inseridos e removidos do topo. Seus principais métodos, sem contar os de **Vector**, são:

- **boolean empty():** Verifica se a pilha está vazia;
- **E peek():** Retorna o elemento do topo da pilha, sem o remover;
- **E pop():** Retorna e remove o elemento do topo da pilha;
- **E push(E item):** Coloca um novo item no topo da pilha;
- **int search(Object o):** Retorna a primeira posição em que o objeto passado como parâmetro é encontrado nessa pilha, usando o critério de busca do método **equals()**.

Para conseguir usar um conjunto de operações mais abrangentes envolvendo pilhas, o framework de coleções da linguagem Java oferece a interface **Deque**.

A interface Set

Set é a interface que define os tipos de coleções que não contêm elementos duplicados, ou seja, os conjuntos. A fim de criar um padrão reutilizável, **java.util.set** define a unicidade dos elementos através do método **equals()**, que deve ser implementado pelos objetos pertencentes ao conjunto de forma que se dois objetos **obj1** e **obj2** são iguais, então **obj1.equals(obj2)** deve retornar **true**.

É importante saber que ao usar **Set**, se o valor de um objeto for alterado, a coleção pode ficar inconsistente, pois dessa forma o critério do **equals()** pode falhar, visto que **Set** não monitora alterações em seus elementos. As principais classes que implementam a interface **Set** são **HashSet** e **TreeSet** e os principais métodos de **Set** são apresentados a seguir:

- **boolean add(E e):** Adiciona um elemento ao conjunto caso ele não se encontre nesse conjunto pelo critério do método **equals()**;
- **boolean addAll(Collection<? extends E> c):** Adiciona todos os elementos da coleção passada como parâmetro ao conjunto (quando já não existem no conjunto);
- **void clear():** Esvazia a collection, mas não elimina da memória os objetos que essa collection referenciava, a não ser que não haja mais nenhuma outra referência para esses objetos;
- **boolean contains(Object o):** Retorna **true** se o conjunto contém o elemento informado como parâmetro pelo critério do método **equals()**;
- **boolean containsAll(Collection<?> c):** Retorna **true** se o conjunto contém todos os elementos da coleção informada como parâmetro;
- **boolean isEmpty():** Retorna **true** se o conjunto não tem elementos;
- **Iterator<E> iterator():** Retorna um **Iterator** para percorrer os elementos desse conjunto;
- **boolean remove(Object o):** Remove da coleção o elemento informado como parâmetro;
- **boolean removeAll(Object o):** Remove do conjunto todos os elementos da coleção informada como parâmetro;
- **int size():** Retorna a quantidade de elementos do conjunto;
- **Object[] toArray():** Converte o conjunto para um **array** de **Object**.

Conhecendo a classe HashSet

A classe **java.util.HashSet** é um tipo de implementação de **Set** oferecida desde o Java 2, que usa uma tabela de *hash* para organizar os elementos em um **array** interno. Uma tabela de *hash* garante maior velocidade em certas operações, conquanto que o **HashSet** seja utilizado adequadamente. Pelo fato de **HashSet** usar uma tabela de *hash*, não é garantida a ordenação dos elementos com o passar do tempo, uma vez que repetidas operações de inserção e remoção poderão levar a um estado não otimizado do vetor interno.

Uma função de *hash* é usada para dispor os elementos no **array** interno, sendo que ela recebe como parâmetro um objeto e determina a posição do mesmo nesse **array** através de uma equação que avalia os atributos do objeto para gerar uma estimativa para a posição em que o mesmo deve ser inserido. No caso de **HashSet**, a função de *hash* é implementada no método **hashCode()** dos objetos, com a finalidade de gerar índices que disponham os elementos adequadamente no **array** interno, de forma espaçada e sem conflitos.

Uma função de *hash* ótima sempre mapeia cada objeto para uma posição desocupada no **array** interno. Entretanto, tal função não existe na prática, pois podemos apenas criar aproximações subótimas. Por outro lado, ao adotar uma função de *hash* ruim, com o tempo o **array** interno pode ficar desorganizado em virtude da função de *hash* causar a concentração de muitos objetos em um pequeno intervalo de posições do vetor interno, o que implica em aumento do tempo de processamento nas operações de inserção, remoção e consulta de elementos.

Na inserção de um novo objeto, tal situação pode fazer com que a função de *hash* designe o novo elemento a uma posição que já esteja ocupada, obrigando **HashSet** a alocar o objeto na próxima posição livre seguinte ao índice que a função de *hash* determinou. Isso diminui a organização na tabela de *hash* e causa um impacto negativo na eficiência de **HashSet**, pois em uma operação de consulta, por exemplo, **HashSet** precisará buscar o elemento nas posições seguintes à posição indicada pela função de *hash*.

Considerando o caso de uma função de *hash* ótima (que nunca mapeie objetos distintos para a mesma posição no array interno), a complexidade das operações de inserção, remoção e consulta de elementos é constante, pois **hashCode()** vai sempre retornar exatamente o índice do elemento no **array**, possibilitando o localizar instantaneamente.

Entretanto, se o **array** interno estiver muito cheio, a função de *hash* empregada vai perder a sua eficiência, passando a indicar a posição aproximada de cada elemento, em vez da posição exata, e assim, o bom desempenho de operações como inserção, remoção e consulta de elementos será prejudicado. Por sua vez, uma função que posiciona elementos muito espaçadamente no **array** interno pode levar ao desperdício de memória, pois poucos objetos são armazenados em vetores muito grandes, o que também pode ser prejudicial.

Outra observação importante sobre **HashSet** é que como ela não é sincronizada, não garante a integridade das informações

sob acesso concorrente. Apesar disso, ela pode ser sincronizada externamente, como qualquer objeto. Adicionalmente, se um **Iterator** for criado para percorrer os elementos de um **HashSet** e o **HashSet** for alterado posteriormente, será lançada a exceção **ConcurrentModificationException** ao acessar os elementos do **Iterator**, pois a sincronização não é garantida. Finalmente, a classe **HashSet** aceita um elemento nulo no máximo, ao contrário de **ArrayList**, que suporta qualquer quantidade de elementos nulos (lembre-se que **ArrayList** aceita elementos repetidos). Seus principais métodos são:

- **boolean add(E e):** Adiciona um elemento ao conjunto se ele atualmente já não se encontra nesse conjunto;
- **boolean addAll(Collection<? extends E> c):** Adiciona todos os elementos da coleção passada como parâmetro ao conjunto (quando já não existem no conjunto);
- **void clear():** Esvazia a collection, mas não elimina da memória os objetos que essa collection referenciava, a não ser que não haja mais nenhuma outra referência para esses objetos;
- **Object clone():** Retorna uma cópia desse **HashSet**, de forma que a cópia contenha novas referências aos objetos apontados por esse **HashSet**;
- **boolean contains(Object o):** Retorna **true** se o conjunto contém o elemento informado como parâmetro;
- **boolean containsAll(Collection<?> c):** Retorna **true** se o conjunto contém todos os elementos da coleção informada como parâmetro;
- **boolean isEmpty():** Retorna **true** se o conjunto não tem elementos;
- **Iterator<E> iterator():** Retorna um **Iterator** para percorrer o conjunto;
- **boolean remove(Object o):** Remove do conjunto o elemento informado como parâmetro;
- **boolean removeAll(Object o):** Remove do conjunto todos os elementos da coleção informada como parâmetro;
- **int size():** Retorna a quantidade de elementos do conjunto;
- **Object[] toArray():** Converte o conjunto em um **array** de **Object**.

Observe que o método **hashCode()** deve ser implementado pela classe dos elementos de forma a definir um código representativo para aquele objeto, o **hashcode**. Na **Listagem 8** é apresentado um exemplo de classe cujos objetos podem ser adicionados a um **HashSet** eficientemente.

Observe que a classe **ElementoInteiro** tem um atributo do tipo **Integer**, representando o valor desse objeto, que pode ser passado como parâmetro no construtor (linhas 5 a 7), sendo o mesmo acessível pelo método **getValor()** (linhas 9 a 11). O método **equals()** é definido nas linhas 13 a 18, sendo que se o objeto passado como parâmetro for de outro tipo que não **ElementoInteiro**, é retornado **false**, e caso contrário, o atributo **valor** de ambos os objetos são comparados usando o método **equals()** da classe **Integer** (linha 17).

Dessa forma é garantida a consistência do método **equals()**. Em seguida, nas linhas 20 a 24, é definido o método **hashCode()**, que tem a função de determinar a posição de cada elemento no **array** interno, sendo definido na implementação que a posição será o quociente da divisão do **valor** por 10. No caso da posição gerada já se encontrar preenchida, será ocupada a próxima posição livre para acomodar o elemento.

Listagem 8. Exemplo com HashSet – Código da classe ElementoInteiro.

```
01. public class ElementoInteiro {
02.
03.  private Integer valor;
04.
05.  public ElementoInteiro(Integer valor) {
06.   this.valor = valor;
07.  }
08.
09.  public Integer getValor() {
10.   return valor;
11.  }
12.
13.  public boolean equals(Object obj) {
14.   if (! (obj instanceof ElementoInteiro))
15.    return false;
16.   else
17.    return valor.equals(((ElementoInteiro)obj).getValor());
18.  }
19.
20.  public int hashCode() {
21.   int hashValue = valor.intValue() / 10;
22.   System.out.println("Valor " + valor + " hashcode=" + hashValue);
23.   return hashValue;
24.  }
25.
26.  public String toString() {
27.   return valor.toString();
28.  }
29. }
```

Na **Listagem 9** é apresentado um exemplo em que é criado um **HashSet** cujos elementos são restritos ao tipo **ElementoInteiro** (linha 5) e em seguida são adicionados sete elementos (linhas 7 a 13). Observe que a cada vez que um objeto é inserido no conjunto, o método **hashCode()** do objeto é invocado, imprimindo na tela o atributo **valor** e o **hashcode** gerado, conforme a declaração do método **hashCode()** na linha 20 da **Listagem 8**.

O resultado completo impresso no console é apresentado na **Listagem 10**. Assim, podemos verificar que o valor 11 gerou o *hashcode* 1, o valor 20 gerou o *hashcode* 2, 50 gerou o *hashcode* 5, 12 gerou o *hashcode* 1 (o mesmo de 11), 66 gerou o *hashcode* 6, 65 gerou o *hashcode* 6 (que é o mesmo de 66) e o valor 20 gerou o *hashcode* 2 (sendo que 20 é um elemento repetido e, portanto, não é adicionado).

Como houve elementos com *hashcodes* repetidos, as posições dos elementos 12 e 65 no **array** interno serão as posições seguintes à dos elementos que já ocupavam essas posições. Portanto, como resultado, temos o conjunto: [11, 12, 20, 50, 66, 65] (vide linha 8 da **Listagem 10**).

Continuando a explicação do exemplo na **Listagem 9**, é verificado se o conjunto contém algum elemento com valor 20 (linha 16). Isto é feito utilizando-se o método **contains()** da classe **HashSet**, passando como parâmetro um novo objeto que contém o **valor** 20. Como resultado, o objeto é encontrado, pois **HashSet** procura

elementos se baseando no método **equals()**, e como este compara os valores dos objetos, ele tem sucesso na busca, imprimindo **true** (veja a linha 10 da **Listagem 10**).

Em seguida, na linha 17 da **Listagem 9**, é procurado o elemento com **valor** 19. Como ele não existe pelo critério do **equals()**, é impresso **false** na saída padrão (veja a linha 12 da **Listagem 10**). Por fim, na linha 19 da **Listagem 9** é removido o elemento de **valor** 66. Note que novamente o **equals()** entra em ação, encontrando o objeto passado como parâmetro. Depois, o método **remove()** o exclui. O **array** resultante é: [11, 12, 20, 50, 65].

Listagem 9. Exemplo com HashSet.

```
01. import java.util.HashSet;
02. public class TesteHashSet {
03.  public static void main(String[] args) {
04.
05.   HashSet<ElementoInteiro> conjunto = new HashSet<>();
06.
07.   conjunto.add(new ElementoInteiro(11));
08.   conjunto.add(new ElementoInteiro(20));
09.   conjunto.add(new ElementoInteiro(50));
10.   conjunto.add(new ElementoInteiro(12));
11.   conjunto.add(new ElementoInteiro(66));
12.   conjunto.add(new ElementoInteiro(65));
13.   conjunto.add(new ElementoInteiro(20));
14.
15.   System.out.println(conjunto);
16.   System.out.println(conjunto.contains(new ElementoInteiro(20)));
17.   System.out.println(conjunto.contains(new ElementoInteiro(19)));
18.
19.   conjunto.remove(new ElementoInteiro(66));
20.   System.out.println(conjunto);
21.  }
22. }
```

Listagem 10. Resultado impresso na saída padrão.

```
01. Valor 11 hashcode=1
02. Valor 20 hashcode=2
03. Valor 50 hashcode=5
04. Valor 12 hashcode=1
05. Valor 66 hashcode=6
06. Valor 65 hashcode=6
07. Valor 20 hashcode=2
08. [11, 12, 20, 50, 66, 65]
09. Valor 20 hashcode=2
10. true
11. Valor 19 hashcode=1
12. false
13. Valor 66 hashcode=6
14. [11, 12, 20, 50, 65]
```

Conhecendo a classe TreeSet

TreeSet é outra classe para o tipo de coleção que oferece suporte a conjuntos (filha de **Set**). Sua principal vantagem é ser ordenada, ou seja, os elementos em uma **java.util.TreeSet** permanecem ordenados de acordo com a ordenação natural deles, que é fornecida por uma implementação de **Comparator**.

Como internamente é mantida uma árvore binária, a complexidade das operações de inserção, remoção e consulta são da ordem de **log(n)**, ou seja, é mais rápida do que o caso de complexidade **n**, que percorre todos os itens da coleção para encontrar um determinado elemento.

Outra informação importante sobre **TreeSet** é que ela não é sincronizada e se um **TreeSet** for alterado, os *iterators* que foram criados antes da mudança vão lançar a exceção **CuncurrentModificationException** se utilizados. Os principais construtores e métodos de **TreeSet**:

- **TreeSet():** Cria um novo **TreeSet** usando a ordenação natural dos elementos, ou seja, de acordo com a implementação de **Comparator** que os elementos apresentam;
- **TreeSet(Collection<? extends E> c):** Cria um **TreeSet** adicionando todos os elementos da coleção informada;
- **TreeSet(Comparator<? super E> comparator):** Cria um **TreeSet** que usa o **Comparator** informado;
- **boolean add(E e):** Adiciona um elemento ao conjunto caso ele não se encontre no mesmo pelo critério do método **equals()**;
- **boolean addAll(Collection<? extends E> c):** Adiciona todos os elementos da coleção passada como parâmetro a esse conjunto (quando já não existem no conjunto);
- **E ceiling(E e):** Retorna o menor elemento no conjunto que seja maior ou igual ao objeto informado como parâmetro ou nulo caso não exista um elemento que se enquadre nesse critério;
- **void clear():** Esvazia a collection, mas não elimina da memória os objetos que ela referenciava, a não ser que não haja mais nenhuma outra referência para esses objetos;
- **Object clone():** Retorna uma cópia de **TreeSet**, de forma que ela contenha novas referências aos objetos apontados por esse **TreeSet**;
- **Comparator<? super E> comparator():** Retorna o **Comparator** usado no conjunto. Se a ordenação é natural, retorna **null**;
- **boolean contains(Object o):** Retorna **true** se o conjunto contém o elemento informado como parâmetro pelo critério do método **equals()**;
- **boolean containsAll(Collection<?> c):** Retorna **true** se o conjunto contém todos os elementos da coleção informada como parâmetro;
- **E first():** Retorna o primeiro elemento do conjunto, ou seja, o menor elemento de acordo com o **Comparator** informado a esse **TreeSet**;
- **E floor(E e):** Retorna o maior elemento que seja menor ou igual ao elemento informado como parâmetro;
- **SortedSet<E> headSet(E toElement):** Retorna um **SortedSet** com todos os elementos menores que o elemento informado. **SortedSet** é uma classe filha de **TreeSet** e é usada para selecionar intervalos de elementos em coleções do tipo **TreeSet**;
- **E higher(E e):** Retorna o menor elemento que é maior que o elemento informado;
- **boolean isEmpty():** Retorna **true** se o conjunto não tem elementos;
- **Iterator<E> iterator():** Retorna um **Iterator** que permite percorrer a coleção;
- **E last():** Retorna o maior elemento do conjunto (o último).
- **E lower(E e):** Retorna o maior elemento que seja menor que o elemento informado como parâmetro;
- **boolean remove(Object o):** Remove da coleção o elemento informado como parâmetro;

• **boolean removeAll(Object o):** Remove do conjunto todos os elementos da coleção informada como parâmetro.
• **int size():** Retorna a quantidade de elementos do conjunto.
• **Object[] toArray():** Converte o conjunto em um **array** de **Object**.

A interface Queue

Criada com o Java 5, **java.util.Queue** representa uma fila, e assim sendo, ela tem um início (também chamado de cabeça) e novos elementos são inseridos no fim da fila (também chamado de cauda). Como todas as coleções apresentadas anteriormente, **Queue** é filha de **Collection** e seus principais métodos são (além dos métodos herdados de **Collection**):
• **E element():** Retorna, mas não remove o elemento inicial da fila. O método **element()** lança uma exceção se a lista não tem elemento inicial;
• **boolean offer(E e):** Insere o elemento no final da fila;
• **E peek():** recupera, mas não remove o elemento inicial da fila. O método **peek()** retorna **null** se a lista não tem elemento inicial;
• **E poll():** Recupera e remove o elemento do início da fila. Retorna **null** se a lista estiver vazia;
• **E remove():** Recupera e remove o elemento do início da fila.

As principais coleções filhas de **Queue** são **PriorityQueue**, **ArrayDeque** e **LinkedList**, cada uma oferecendo uma implementação especializada de fila. **PriorityQueue** representa uma fila que possui a ordenação dada por uma implementação de **Comparator** e **ArrayDeque** representa uma fila em que se pode inserir e remover elementos no final ou no início.

A interface Map

A interface **Map** não descende de **Collection**, porém faz parte do framework de coleções da linguagem Java, pois **java.util.Map<K,V>** representa uma lista de chaves com valores, onde **K** representa as chaves e **V** representa os valores.

Como uma das suas principais características, uma coleção que implementa **Map** não conterá chaves duplicadas, uma vez que **Map** mapeia chaves **K** para valores **V** e não permite duplicidade da chave **K**. Para se ter um uso eficiente de **Map**, deve-se evitar o uso chaves que sejam objetos mutáveis, pois se tais objetos forem alterados, as relações de chave e valor mantidas por **Map** não serão atualizadas automaticamente, ou seja, ele fica com um mapeamento inconsistente das chaves para os valores. Os principais métodos de **Map** são:
• **void clear():** Elimina todos os mapeamentos do mapa;
• **boolean containsKey(Object key):** Retorna **true** se existe um mapeamento para essa chave no mapa;
• **boolean containsValue(Object value):** Retorna **true** se existe uma ou mais chaves que contêm esse valor no mapa;
• **V get(Object key):** Retorna o valor a qual essa chave é mapeada;

• **boolean isEmpty():** Retorna **true** se o mapa não tem elementos;
• **V remove(Object key):** Remove o mapeamento dado pela chave informada;
• **boolean remove (Object key, Object value)**: Remove o mapeamento com a chave informada somente se esta chave aponta para o valor indicado;
• **boolean replace (K key, V value)**: Substitui o valor da chave pelo informado;
• **int size():** Retorna a quantidade de chaves do mapa;
• **Collection<V> values():** Retorna uma lista de valores do mapa.

As principais classes filhas de **Map** são **HashMap**, **HashLinkedMap**, **HashTable** e **TreeMap**, sendo que uma das mais utilizadas, a **HashMap**, será apresentada a seguir.

Conhecendo a classe HashMap

A classe **HashMap** herda todos os métodos da interface **Map** e serve para criar coleções em que chaves são associadas a valores específicos, pois **HashMap** gerencia pares de registros, onde cada chave está ligada a seu respectivo valor. Como regra, os objetos usados como chaves devem implementar os métodos **equals()** (para verificar se a chave se encontra na coleção) e **hashCode()** (que é uma função de *hash* que determina a posição dos elementos no **array** interno).

Uma observação importante é que **HashMap** é muito semelhante a **HashTable**, porém existem algumas diferenças entre elas. Primeiramente, **HashTable** pode ser compreendida como uma versão sincronizada de **HashMap**. Outra diferença é que **HashMap** permite somente uma chave nula e qualquer quantidade de valores nulos, enquanto **HashTable** não permite chaves nem valores nulos. Por fim, **HashTable**, além de implementar a interface **HashMap**, é filha da classe **Dictionary**, que é desaprovada (não recomendada para ser usada), o que não é o caso de **HashMap** (filha de **AbstractMap**). Portanto, **HashMap** é indicada para ser usada no lugar de **HashTable** em novas aplicações.

Os principais métodos de **HashMap** são:
• **void clear():** Elimina todos os mapeamentos do mapa;
• **Object clone():** Retorna uma cópia do **HashMap**, de forma que ela contenha novas referências aos objetos apontados por esse **HashMap**;
• **boolean contains(Object value):** Retorna **true** se o mapa contém uma chave que aponte para o valor informado;
• **boolean containsKey(Object key):** Retorna **true** se existe um mapeamento para essa chave no mapa;
• **boolean containsValue(Object value):** Retorna **true** se existe uma ou mais chaves que contêm esse valor no mapa;
• **V get(Object key):** Retorna o valor a qual essa chave é mapeada;
• **boolean isEmpty():** Retorna **true** se o mapa não tem elementos;
• **V put(K key, V value):** Mapeia o valor especificado para a chave informada;
• **V remove(Object key):** Remove o mapeamento dado pela chave informada;
• **boolean remove (Object key, Object value)**: Remove o mapeamento com a chave e valores indicados;
• **boolean replace (K key, V value)**: Substitui o valor da chave pelo informado;
• **int size():** Retorna a quantidade de chaves do mapa;
• **Collection<V> values():** Retorna uma lista de valores do mapa.

É de grande importância a todo programador dominar o framework de coleções, para assim saber adotar o tipo de coleção mais adequado às necessidades de cada projeto. O uso inadequado de coleções pode causar um atraso nas operações básicas e perda de informação.

Ademais, como o framework de coleções do Java viabiliza um conjunto de classes e interfaces reutilizável e empregado nas mais diversas aplicações, a sua adoção pode, inclusive, simplificar o desenvolvimento e a manutenção de código, por ser uma solução padronizada e de conhecimento de todos.

Autor

John Soldera
johnsoldera@gmail.com
É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.

Links:

Javadoc da plataforma Java SE 8.
https://docs.oracle.com/javase/8/docs/api/

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486