1.5 Introdução à tecnologia de objetos

Hoje, como a demanda por software novo e mais poderoso está aumentando, construir softwares de maneira rápida, correta e econômica continua a ser um objetivo indefinido. *Objetos* ou, mais precisamente, as *classes* de onde os objetos vêm são essencialmente componentes *reutilizáveis* de software. Há objetos data, objetos data/hora, objetos áudio, objetos vídeo, objetos automóvel, objetos pessoas etc. Quase qualquer *substantivo* pode ser razoavelmente representado como um objeto de software em termos dos *atributos* (por exemplo, nome, cor e tamanho) e *comportamentos* (por exemplo, calcular, mover e comunicar). Grupos de desenvolvimento de software podem usar uma abordagem modular de projeto e implementação orientados a objetos para que sejam muito mais produtivos do que com as técnicas anteriormente populares como "programação estruturada" — programas orientados a objetos são muitas vezes mais fáceis de entender, corrigir e modificar.

1.5.1 O automóvel como um objeto

Para ajudar a entender objetos e seus conteúdos, vamos começar com uma analogia simples. Suponha que você *queira guiar um carro e fazê-lo andar mais rápido pisando no pedal acelerador*. O que deve acontecer antes que você possa fazer isso? Bem, antes de poder dirigir um carro, alguém tem de *projetá-lo*. Um carro tipicamente começa como desenhos de engenharia, semelhantes a *plantas* que descrevem o projeto de uma casa. Esses desenhos incluem o projeto do pedal do acelerador. O pedal *oculta* do motorista os complexos mecanismos que realmente fazem o carro ir mais rápido, assim como o pedal de freio "oculta" os mecanismos que diminuem a velocidade do carro e a direção "oculta" os mecanismos que mudam a direção dele. Isso permite que pessoas com pouco ou nenhum conhecimento sobre como motores, freios e mecanismos de direção funcionam consigam dirigir um carro facilmente.

Assim como você não pode cozinhar refeições na planta de uma cozinha, não pode dirigir os desenhos de engenharia de um carro. Antes de poder guiar um carro, ele deve ser *construído* a partir dos desenhos de engenharia que o descrevem. Um carro pronto tem um pedal de acelerador *real* para fazê-lo andar mais rápido, mas mesmo isso não é suficiente — o carro não acelerará por conta própria (tomara!), então o motorista deve *pressionar* o pedal do acelerador.

1.5.2 Métodos e classes

Vamos usar nosso exemplo do carro para introduzir alguns conceitos fundamentais da programação orientada a objetos. Para realizar uma tarefa em um programa é necessário um **método**. O método armazena as declarações do programa que, na verdade, executam as tarefas; além disso, ele oculta essas declarações do usuário, assim como o pedal do acelerador de um carro oculta do motorista os mecanismos para fazer o veículo ir mais rápido. No Java, criamos uma unidade de programa chamada **classe** para armazenar o conjunto de métodos que executam as tarefas dela. Por exemplo, uma classe que representa uma conta bancária poderia conter um método para *fazer depósitos* de dinheiro, outro para *fazer saques* e um terceiro para *perguntar* qual é o saldo atual. Uma classe é similar em termos do conceito aos desenhos de engenharia de um carro, que armazenam o projeto de um pedal de acelerador, volante etc.

1.5.3 Instanciação

Assim como alguém tem de *fabricar um carro* a partir dos desenhos de engenharia antes que possa realmente dirigi-lo, você deve *construir um objeto* de uma classe antes que um programa possa executar as tarefas que os métodos da classe definem. O processo para fazer isso é chamado *instanciação*. Um objeto é então referido como uma **instância** da sua classe.

1.5.4 Reutilização

Assim como os desenhos de engenharia de um carro podem ser *reutilizados* várias vezes para fabricar muitos carros, você pode *reutilizar* uma classe muitas vezes para construir vários objetos. A reutilização de classes existentes ao construir novas classes e programas economiza tempo e esforço. Também ajuda a construir sistemas mais confiáveis e eficientes, porque classes e componentes existentes costumam passar por extensos *testes*, *depuração* e *ajuste de desempenho*. Assim como a noção das *partes intercambiáveis* foi crucial para a Revolução Industrial, classes reutilizáveis são fundamentais para a revolução de software que foi estimulada pela tecnologia de objetos.

Observação de engenharia de software 1.1

Utilize uma abordagem de bloco de construção para criar seus programas. Evite reinventar a roda — use as peças de alta qualidade existentes sempre que possível. Essa reutilização de software *é um dos principais benefícios da programação orientada a objetos.*

1.5.5 Mensagens e chamadas de método

Ao dirigir um carro, o ato de pressionar o acelerador envia uma *mensagem* para o veículo realizar uma tarefa — isto é, ir mais rápido. Da mesma forma, você *envia mensagens para um objeto*. Cada mensagem é implementada como uma **chamada de método** que informa a um método do objeto a maneira de realizar sua tarefa. Por exemplo, um programa pode chamar o método *depósito* de um objeto conta bancária para aumentar o saldo da conta.

1.5.6 Atributos e variáveis de instância

Um carro, além de ter a capacidade de realizar tarefas, também tem *atributos*, como cor, número de portas, quantidade de gasolina no tanque, velocidade atual e registro das milhas totais dirigidas (isto é, a leitura do odômetro). Assim como suas capacidades, os atributos do carro são representados como parte do seu projeto nos diagramas de engenharia (que, por exemplo, incluem um odômetro e um medidor de combustível). Ao dirigir um carro real, esses atributos são incorporados a ele. Cada carro mantém seus *próprios* atributos. Cada carro sabe a quantidade de gasolina que há no seu tanque, mas desconhece quanto há no tanque de *outros* carros.

Um objeto, da mesma forma, tem atributos que ele incorpora à medida que é usado em um programa. Esses atributos são especificados como parte da classe do objeto. Por exemplo, um objeto conta bancária tem um *atributo saldo* que representa a quantidade de dinheiro disponível. Cada objeto conta bancária sabe o saldo que ele representa, mas *não* os saldos de *outras* contas bancárias. Os atributos são especificados pelas **variáveis de instância** da classe.

1.5.7 Encapsulamento e ocultamento de informações

Classes (e seus objetos) **encapsulam**, isto é, contêm seus atributos e métodos. Os atributos e métodos de uma classe (e de seu objeto) estão intimamente relacionados. Os objetos podem se comunicar entre si, mas eles em geral não sabem como outros objetos são implementados — os detalhes de implementação permanecem *ocultos* dentro dos próprios objetos. Esse **ocultamento de informações**, como veremos, é crucial à boa engenharia de software.

1.5.8 Herança

Uma nova classe de objetos pode ser criada convenientemente por meio de **herança** — ela (chamada **subclasse**) começa com as características de uma classe existente (chamada **superclasse**), possivelmente personalizando-as e adicionando aspectos próprios.

Na nossa analogia do carro, um objeto da classe "conversível" decerto *é um* objeto da classe mais *geral* "automóvel", mas, *especificamente,* o teto pode ser levantado ou baixado.

1.5.9 Interfaces

O Java também suporta **interfaces** — coleções de métodos relacionados que normalmente permitem informar aos objetos *o que* fazer, mas não *como* fazer (veremos uma exceção a isso no Java SE 8). Na analogia do carro, uma interface das capacidades "básicas de dirigir" consistindo em um volante, um pedal de acelerador e um pedal de freio permitiria que um motorista informasse ao carro *o que* fazer. Depois que você sabe como usar essa interface para virar, acelerar e frear, você pode dirigir muitos tipos de carro, embora os fabricantes possam *implementar* esses sistemas *de forma diferente.*

Uma classe **implementa** zero ou mais interfaces — cada uma das quais pode ter um ou mais métodos —, assim como um carro implementa interfaces separadas para as funções básicas de dirigir, controlar o rádio, controlar os sistemas de aquecimento, ar-condicionado e afins. Da mesma forma que os fabricantes de automóveis implementam os recursos *de forma distinta,* classes podem implementar métodos de uma interface de maneira *diferente.* Por exemplo, um sistema de software pode incluir uma interface de "backup" que ofereça os métodos *save* e *restore.* As classes podem implementar esses métodos de modo diferente, dependendo dos tipos de formato em que é feito o backup, como programas, textos, áudios, vídeos etc., além dos tipos de dispositivo em que esses itens serão armazenados.

1.5.10 Análise e projeto orientados a objetos (OOAD)

Logo você estará escrevendo programas em Java. Como criará o **código** (isto é, as instruções do programa) para seus programas? Talvez, como muitos programadores, simplesmente ligará seu computador e começará a digitar. Essa abordagem pode funcionar para pequenos programas (como os apresentados nos primeiros capítulos deste livro), mas e se você fosse contratado para criar um sistema de software para controlar milhares de caixas automáticos de um banco importante? Ou se fosse trabalhar em uma equipe de 1.000 desenvolvedores de software para construir a próxima geração de sistema de controle de tráfego aéreo dos Estados Unidos? Para projetos tão grandes e complexos, não sentaria e simplesmente começaria a escrever programas.

Para criar as melhores soluções, você deve seguir um processo de **análise** detalhado a fim de determinar os **requisitos** do projeto (isto é, definir *o que* o sistema deve fazer) e desenvolver um **design** que os atenda (isto é, especificar *como* o sistema deve fazê-lo). Idealmente, você passaria por esse processo e revisaria cuidadosamente o projeto (e teria seu projeto revisado por outros profissionais de software) antes de escrever qualquer código. Se esse processo envolve analisar e projetar o sistema de um ponto de vista orientado a objetos, ele é chamado de **processo de análise e projeto orientados a objetos (*object-oriented analysis and design* — OOAD)**. Linguagens como Java são orientadas a objetos. A programação nessa linguagem, chamada **programação orientada a objetos (*object-oriented programming* — OOP)**, permite-lhe implementar um projeto orientado a objetos como um sistema funcional.

1.8 Java

A contribuição mais importante até agora da revolução dos microprocessadores é que ela permitiu o desenvolvimento de computadores pessoais. Os microprocessadores estão tendo um impacto profundo em dispositivos eletrônicos inteligentes de consumo popular. Reconhecendo isso, a Sun Microsystems, em 1991, financiou um projeto de pesquisa corporativa interna chefiado por James Gosling, que resultou em uma linguagem de programação orientada a objetos chamada C++, que a empresa chamou de Java.

Um objetivo-chave do Java é ser capaz de escrever programas a serem executados em uma grande variedade de sistemas computacionais e dispositivos controlados por computador. Isso às vezes é chamado de "escreva uma vez, execute em qualquer lugar".

Por uma feliz casualidade, a web explodiu em popularidade em 1993 e a Sun viu o potencial de utilizar o Java para adicionar *conteúdo dinâmico*, como interatividade e animações, às páginas da web. O Java chamou a atenção da comunidade de negócios por causa do interesse fenomenal pela web. Ele é agora utilizado para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores da web (os computadores que fornecem o conteúdo que vemos em nossos navegadores), fornecer aplicativos para dispositivos voltados ao consumo popular (por exemplo, telefones celulares, smartphones, televisão, *set-up boxes* etc.) e para muitos outros propósitos. Ainda, ele também é a linguagem-chave para desenvolvimento de aplicativos Android adequados a smartphones e tablets. A Sun Microsystems foi adquirida pela Oracle em 2010.

Bibliotecas de classe do Java

Você pode criar cada classe e método de que precisa para formar seus programas Java. Porém, a maioria dos programadores Java tira proveito das ricas coleções de classes existentes e métodos nas **bibliotecas de classe Java**, também conhecidas como **Java APIs (*application programming interfaces*)**.

Dica de desempenho 1.1

Utilizar as classes e os métodos da Java API em vez de escrever suas próprias versões pode melhorar o desempenho de programa, porque eles são cuidadosamente escritos para executar de modo eficiente. Isso também diminui o tempo de desenvolvimento de programa.

1.9 Um ambiente de desenvolvimento Java típico

Agora explicaremos os passos para criar e executar um aplicativo Java. Normalmente, existem cinco fases: editar, compilar, carregar, verificar e executar. Nós as discutiremos no contexto do Java SE 8 Development Kit (JDK). Consulte a seção *"Antes de começar" (nas páginas iniciais do livro) para informações sobre como baixar e instalar o JDK no Windows, Linux e OS X.*

Fase 1: criando um programa

A Fase 1 consiste em editar um arquivo com um *programa editor*, muitas vezes conhecido simplesmente como um *editor* (Figura 1.6). Você digita um programa Java (em geral referido como **código-fonte**) utilizando o editor, faz quaisquer correções necessárias e salva o programa em um dispositivo de armazenamento secundário, como sua unidade de disco. Arquivos de código-fonte Java recebem um nome que termina com a **extensão `.java`**, que indica um arquivo contendo código-fonte Java.

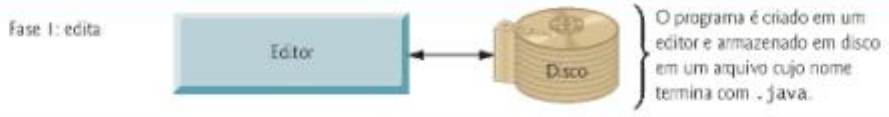

Figura 1.6 | Ambiente típico de desenvolvimento Java — fase de edição.

Dois editores amplamente utilizados nos sistemas Linux são `vi` e `emacs`. O Windows fornece o Bloco de Notas. Já o OS X fornece o TextEdit. Também há muitos editores freeware e shareware disponíveis on-line, incluindo Notepad++ (`notepad-plus-plus.org`), EditPlus (`www.editplus.com`), TextPad (`www.textpad.com`) e jEdit (`www.jedit.org`).

Ambientes de desenvolvimento integrado (IDEs) fornecem ferramentas que suportam o processo de desenvolvimento de software, como editores e depuradores para localizar **erros lógicos** (que fazem programas serem executados incorretamente) e outros. Há muitos IDEs Java populares, incluindo:

- Eclipse (`www.eclipse.org`)
- NetBeans (`www.netbeans.org`)
- IntelliJ IDEA (`www.jetbrains.com`)

No site dos autores (Seção Antes de começar, nas páginas iniciais do livro) estão os vídeos Dive Into®, que mostram como executar os aplicativos Java desta obra e como desenvolver novos aplicativos Java com o Eclipse, NetBeans e IntelliJ IDEA.

Fase 2: compilando um programa Java em bytecodes

Na Fase 2, utilize o comando `javac` (o **compilador Java**) para **compilar** um programa (Figura 1.7). Por exemplo, a fim de compilar um programa chamado `Welcome.java`, você digitaria

```
javac Welcome.java
```

na janela de comando do seu sistema (isto é, o Prompt do MS-DOS, no Windows, ou o aplicativo Terminal, no Mac OS X) ou em um shell Linux (também chamado Terminal em algumas versões do Linux). Se o programa compilar, o compilador produz um arquivo `.class` chamado `Welcome.class` que contém a versão compilada. IDEs tipicamente fornecem um item de menu, como Build ou Make, que chama o comando `javac` para você. Se o compilador detectar erros, você precisa voltar para a Fase 1 e corrigi-los. No Capítulo 2, discutiremos com detalhes os tipos de erro que o compilador pode detectar.

O compilador Java converte o código-fonte Java em **bytecodes** que representam as tarefas a serem executadas na fase de execução (Fase 5). O **Java Virtual Machine (JVM)** — uma parte do JDK e a base da plataforma Java — executa bytecodes. A **máquina virtual (*virtual machine* — VM)** é um aplicativo de software que simula um computador, mas oculta o sistema operacional e o hardware subjacentes dos programas que interagem com ela. Se a mesma máquina virtual é implementada em muitas plataformas de computador, os aplicativos escritos para ela podem ser utilizados em todas essas plataformas. A JVM é uma das máquinas virtuais mais utilizadas. O .NET da Microsoft utiliza uma arquitetura de máquina virtual semelhante.

Diferentemente das instruções em linguagem de máquina, que são *dependentes de plataforma* (isto é, de hardware específico de computador), instruções bytecode são *independentes de plataforma*. Portanto, os bytecodes do Java são **portáveis** — sem recompilar o código-fonte, as mesmas instruções em bytecodes podem ser executadas em qualquer plataforma contendo uma JVM que entende a versão do Java na qual os bytecodes foram compilados. A JVM é invocada pelo comando `java`. Por exemplo, para executar um aplicativo Java chamado `Welcome`, você digitaria

```
java Welcome
```

em uma janela de comando para invocar a JVM, que então iniciaria os passos necessários a fim de executar o aplicativo. Isso começa a Fase 3. IDEs tipicamente fornecem um item de menu, como Run, que chama o comando `java` para você.

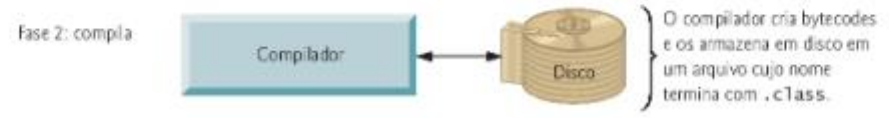

Figura 1.7 | Ambiente típico de desenvolvimento Java — fase de compilação.

Fase 3: carregando um programa na memória

Na Fase 3, a JVM armazena o programa na memória para executá-lo — isso é conhecido como **carregamento** (Figura 1.8). O **carregador de classe** da JVM pega os arquivos `.class` que contêm os bytecodes do programa e os transfere para a memória primária. Ele também carrega qualquer um dos arquivos `.class` fornecidos pelo Java que seu programa usa. Os arquivos `.class` podem ser carregados a partir de um disco em seu sistema ou em uma rede (por exemplo, sua faculdade local ou rede corporativa ou a internet).

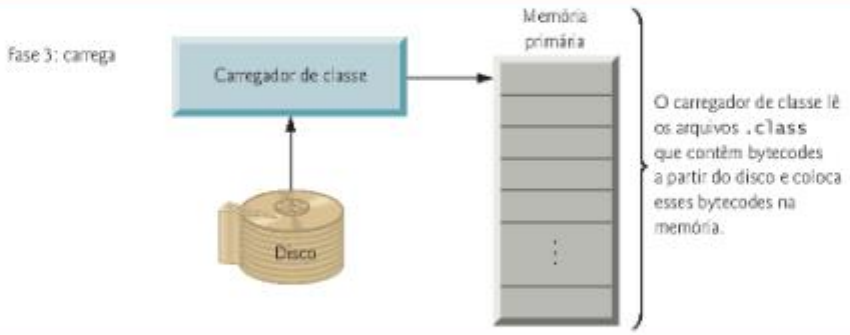

Figura 1.8 | Ambiente típico de desenvolvimento Java — fase de carregamento.

Fase 4: verificação de bytecode

Na Fase 4, enquanto as classes são carregadas, o **verificador de bytecode** examina seus bytecodes para assegurar que eles são válidos e não violam restrições de segurança do Java (Figura 1.9). O Java impõe uma forte segurança para certificar-se de que os programas Java que chegam pela rede não danificam os arquivos ou o sistema (como vírus e worms de computador).

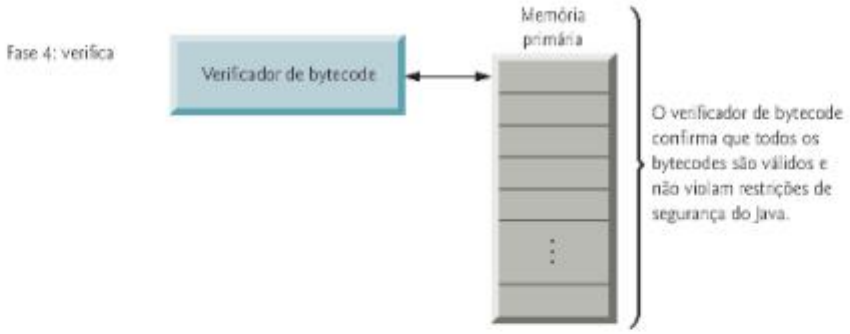

Figura 1.9 | Ambiente típico de desenvolvimento Java — fase de verificação.

Fase 5: execução

Na Fase 5, a JVM **executa** os bytecodes do programa, realizando, assim, as ações especificadas por ele (Figura 1.10). Nas primeiras versões do Java, a JVM era simplesmente um *interpretador* para bytecodes. A maioria dos programas Java executava lentamente, porque a JVM interpretava e executava um bytecode de cada vez. Algumas arquiteturas modernas de computador podem executar várias instruções em paralelo. Em geral, as JVMs atuais executam bytecodes utilizando uma combinação de interpretação e a chamada **compilação *just in time* (JIT)**. Nesse processo, a JVM analisa os bytecodes à medida que eles são interpretados, procurando *hot spots* (*pontos ativos*) — partes dos bytecodes que executam com frequência. Para essas partes, um **compilador *just in time* (JIT)**, como o **compilador Java HotSpot™** da Oracle, traduz os bytecodes para a linguagem de máquina do computador subjacente. Quando a JVM encontra de novo essas partes compiladas, o código de linguagem de máquina mais rápido é executado. Portanto, os programas Java realmente passam por *duas* fases de compilação: uma em que o código-fonte é traduzido em bytecodes (para a portabilidade entre JVMs em diferentes plataformas de computador) e outra em que, durante a execução, os *bytecodes* são traduzidos em *linguagem de máquina* para o computador real no qual o programa é executado.

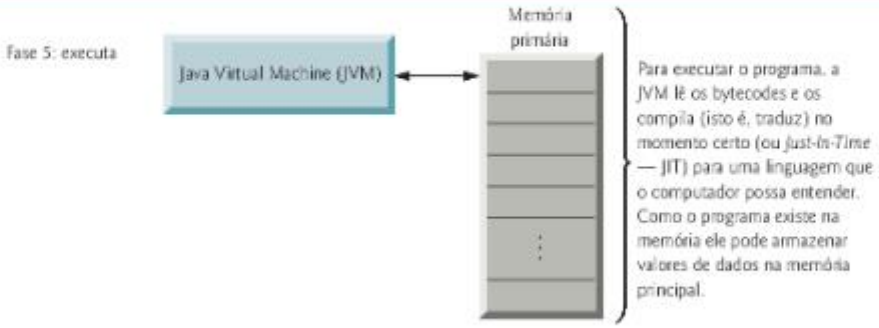

Figura 1.10 | Ambiente típico de desenvolvimento Java — fase de execução.

Problemas que podem ocorrer no tempo de execução

Os programas podem não funcionar na primeira tentativa. Cada uma das fases anteriores pode falhar por causa de vários erros que discutiremos ao longo dos capítulos. Por exemplo, um programa executável talvez tente realizar uma operação de divisão por zero (uma operação ilegal para a aritmética de número inteiro em Java). Isso faria o programa Java imprimir uma mensagem de erro. Se isso ocorresse, você teria de retornar à fase de edição, realizar as correções necessárias e passar novamente pelas demais fases para determinar se as correções resolveram o(s) problema(s). [***Observação:*** a maioria dos programas Java realiza entrada ou saída de dados. Quando afirmamos que um programa exibe uma mensagem, normalmente queremos dizer que ele a apresenta pela tela do computador. As mensagens e outros dados podem ser enviados a outros dispositivos de saída, como discos e impressoras, ou até mesmo uma rede para transmissão a outros computadores.]

Erro comum de programação 1.1

Os erros como divisão por zero ocorrem enquanto um programa executa, então são chamados ***runtime errors*** *ou* ***erros de tempo de execução****. Erros de tempo de execução fatais fazem os programas serem imediatamente encerrados sem terem realizado seus trabalhos com sucesso. Erros de tempo de execução não fatais permitem que os programas executem até sua conclusão, produzindo frequentemente resultados incorretos.*