Tutorial

Fundamentos da linguagem Java

Programação orientada a objetos na plataforma Java

Por: J Steven Perry

Atualizado 26/07/2021 | Publicado 19/07/2010

Antes de começar

Descubra o que esperar deste tutorial e como obter o máximo dele.

Sobre este tutorial

O tutorial de duas partes *Introdução à programação Java* destina-se aos desenvolvedores de software que conhecem pouco da tecnologia Java. Execute as duas partes para o funcionamento com a programação orientada a objeto (OOP) e desenvolvimento de aplicativo real usando a linguagem e plataforma Java.

Esta primeira parte é uma introdução etapa a etapa à OOP, usando a linguagem Java. O tutorial começa com uma visão geral da plataforma e linguagem Java, seguida de instruções para configurar um ambiente de desenvolvimento que consiste de uma Java Development Kit (JDK) e Eclipse IDE. Depois da introdução aos componentes do ambiente de desenvolvimento, você começa a aprender a prática de sintaxe básica Java.

Parte 2 abrange recursos de linguagem mais avançados, incluindo expressões regulares, genéricas, E/S e serialização. Os exemplos de programação na Parte 2 criam o objeto `Person` que você começa a desenvolver na Parte 1.

Objetivos

Ao concluir a Parte 1, você estará familiarizado com a sintaxe de linguagem básica Java e capaz de escrever programas Java simples. Acompanhe "*Introdução à programação Java*, Parte 2: Construções para aplicativos reais" para criar esta base.

Pré-requisitos

Este tutorial destina-se a desenvolvedores de software que ainda não possuem experiência no código ou plataforma Java. O tutorial inclui uma visão geral dos conceitos de OOP.

Requisitos do sistema

Para concluir os exercícios neste tutorial, instale e configure um ambiente de desenvolvimento que consiste de:

- JDK 8 em Oracle
- Eclipse IDE for Java Developers

As instruções de download e instalação para ambos estão incluídas no tutorial.

A configuração do sistema recomendada é:

- Um sistema que suporte Java SE 8 com pelo menos 2 GB de memória. Java 8 é suportado em Linux®, Windows®, Solaris® e Mac OS X.
- Pelo menos 200 MB de espaço em disco para instalar os componentes de software e exemplos.

Visão geral da plataforma Java

A tecnologia Java é usada para desenvolver aplicativos para uma ampla variedade de ambientes, de dispositivos consumidores a sistemas corporativos heterogêneos. Nesta seção, obtenha uma visualização de alto nível da plataforma Java e seus componentes.

A linguagem Java

Como qualquer linguagem de programação, a linguagem Java tem sua própria estrutura, regras de sintaxe e paradigma de programação. O paradigma de programação da linguagem Java baseia-se no conceito de OOP, que os recursos da linguagem suportam.

A linguagem Java deriva da linguagem C, portanto suas regras de sintaxe assemelham-se às regras de C. Por exemplo, os blocos de códigos são modularizados em métodos e delimitados por chaves ({ e }) e variáveis são declaradas antes que sejam usadas.

Estruturalmente, a linguagem Java começa com *pacotes*. Um pacote é o mecanismo de namespace da linguagem Java. Dentro dos pacotes estão as classes e dentro das classes estão métodos, variáveis, constantes e mais. Neste tutorial você aprende sobre as partes da linguagem Java.

O compilador Java

Quando você programa na plataforma Java, escreve seu código-fonte em arquivos .java e depois os compila. O compilador verifica seu código nas regras de sintaxe da linguagem e depois grava *bytecode* em arquivos .class. Bytecode é um conjunto de instruções destinadas a executar em uma Java virtual machine (JVM). Ao incluir esse nível de abstração, o compilador Java difere-se de outros compiladores de linguagem, que escrevem instruções adequadas para o chipset de CPU no qual o programa é executado.

A JVM

No tempo de execução, a JVM lê e interpreta arquivos .class e executa as instruções do programa na plataforma de hardware nativa para qual a JVM foi escrita. A JVM interpreta o bytecode como uma CPU interpretaria instruções de linguagem assembly. A diferença é que a JVM é uma parte do software escrita especificamente para uma determinada plataforma. A JVM é o núcleo do princípio “gravação única, execução em qualquer local” da linguagem Java. Seu código pode executar em qualquer chipset para o qual a implementação da JVM adequada está disponível. JVMs estão disponíveis para principais plataformas, como Linux e Windows, e subconjuntos de linguagem Java foram implementados nas JVMs para telefones celulares e chips hobbyist.

O coletor de lixo

Em vez de forçá-lo a manter a alocação de memória (ou usar uma biblioteca de terceiros para isso), a plataforma Java fornece gerenciamento de memória fora do padrão. Quando seu aplicativo Java cria uma instância de objeto no tempo de execução, a JVM aloca automaticamente espaço de memória para esse objeto a partir de um conjunto de memória *heap* — reservado para uso de seu programa. O *coletor de lixo* Java é executado em segundo plano, mantendo o controle de quais objetos o aplicativo não necessita mais e recuperando memória deles. Essa abordagem para manipulação de memória é chamada de *gerenciamento implícito de memória* porque não exige a gravação de qualquer código de manipulação de memória. A coleta de lixo é um dos recursos essenciais para o desempenho da plataforma Java.

O Java Development Kit

Ao fazer o download de um Java Development Kit (JDK), você obtém, — além do compilador e de outras ferramentas, — uma biblioteca de classe completa de utilitários de pré-construção que o ajuda a realizar tarefas de desenvolvimento de aplicativo mais comuns. A melhor forma de obter uma ideia do escopo dos pacotes e bibliotecas JDK é verificar a documentação da API JDK .

O Java Runtime Environment

O Java Runtime Environment (JRE; também conhecido como o tempo de execução Java) inclui a JVM, bibliotecas de códigos e componentes necessários para executar programas que são escritos na linguagem Java. O JRE está disponível para diversas plataformas.É possível redistribuir livremente o JRE com seus aplicativos, de acordo com os termos da licença do JRE, para fornecer aos usuários do aplicativo uma plataforma na qual executar seu software. O JRE está incluído no JDK.

Configurando seu ambiente de desenvolvimento Java

Nesta seção, você fará o download e instalará o JDK e a liberação atual do Eclipse IDE e configurará seu ambiente de desenvolvimento Eclipse.

Se você já tiver o JDK e Eclipse IDE instalados, poderá desejar pular para a seção "Introdução ao Eclipse" ou para aquela depois dela, "Conceitos de programação orientada a objeto."

Seu ambiente de desenvolvimento

O JDK inclui um conjunto de ferramentas de linha de comandos para compilar e executar seu código Java, incluindo uma cópia completa do JRE. Embora seja possível usar essas ferramentas para desenvolver seus aplicativos, a maioria dos desenvolvedores apreciam a funcionalidade adicional, o gerenciamento de tarefas e a interface visual de um IDE.

Eclipse é um IDE de software livre popular para desenvolvimento Java. O Eclipse manipula tarefas básicas, como a compilação e depuração de códigos, portanto, você pode focar na escrita e teste de códigos. Além disso, é possível usar o Eclipse para organizar seus arquivos de código-fonte em projetos, compilar e testar esses projetos e armazenar arquivos de projetos em qualquer número de repositórios de origem. É necessário ter um JDK instalado para usar Eclipse para desenvolvimento Java. Se você fizer o download de um dos pacotes configuráveis Eclipse, ele já virá com o JDK.

Instale o JDK

Siga estas etapas para fazer o download e instalar o JDK:

1. Navegue para Downloads do Java SE e clique na caixa **Plataforma Java (JDK)** para exibir a página de download da última versão do JDK.
2. Concorde com os termos da licença.
3. Em **Java SE Development Kit**, escolha o download que corresponda a seu sistema operacional e arquitetura de chip.

Windows

1. Salve o arquivo em sua unidade de disco rígido quando solicitado.
2. Quando o download estiver concluído, execute o programa de instalação. Instale o JDK em sua unidade de disco rígido em um local fácil de lembrar, como C:\home\Java\jdk1.8.0_60. É uma boa ideia codificar o número de atualização no nome do diretório de instalação escolhido.

OS X

1. Quando o download estiver concluído, dê um clique duplo nele para montá-lo.

2. Execute o programa de instalação. Você não escolhe onde o JDK será instalado. É possível executar `/usr/libexec/java_home -1.8` para ver o local do JDK 8 em seu Mac. O caminho isdisplay é semelhantea /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home.

Consulte Instalação de JDK 8 e JRE 8 para obter mais informações.

Agora você possui um ambiente Java em seu computador. Depois, instalará o Eclipse IDE.

Instale o Eclipse

Para fazer o download e instalar o Eclipse, siga estas etapas:

1. Navegue até Página de downloads do Eclipse IDE .
2. Clique em **Eclipse IDE for Java Developers**.
3. Em Links de download à direita, escolha sua plataforma (o site já pode ter encontrado seu tipo de sistema operacional).
4. Clique no espelho do qual deseja fazer download; depois salve o arquivo em sua unidade de disco rígido.
5. Extraia o conteúdo do arquivo .zip em um local em sua unidade de disco rígido do qual se lembre facilmente (como C:\home\eclipse no Windows ou ~/home/eclipse em Mac ou Linux).

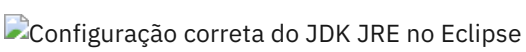
Configure o Eclipse

O Eclipse IDE opera sobre o JDK como um conceito de abstração útil, mas ainda precisa acessar o JDK e suas várias ferramentas. Antes de poder usar o Eclipse para escrever código Java, é necessário informar a ele onde o JDK está localizado.

Para configurar seu ambiente de desenvolvimento Eclipse:

1. Ative o Eclipse clicando duas vezes em eclipse.exe (ou no executável equivalente em sua plataforma).
2. O Ativador de área de trabalho é aberto, permitindo escolher uma pasta raiz para seus projetos Eclipse. Use uma pasta da qual possa se lembrar facilmente, como C:\home\workspace em Windows ou ~/home/workspace em Mac ou Linux.
3. Feche a janela Bem-vindo ao Eclipse.
4. Clique em **Janela > Preferências > Java > JREs instalados**. Figura 1 mostra esta seleção destacada na janela de configuração do Eclipse para o JRE.

   Configurando o JDK que o Eclipse usa

   
   Configuração correta do JDK JRE no Eclipse

5. O Eclipse aponta para um JRE instalado. Você deve usar o JRE que transferiu por download com o JDK. Se o Eclipse não detectar automaticamente o JDK instalado, clique em **Incluir...** e, na próxima caixa de diálogo, clique em **VM padrão** e clique em **Avançar**.

6. Especifique o diretório inicial do JDK (como C:\home\jdk1.8.0_60 em Windows) e clique em **Concluir**.

7. Confirme se o JDK que você deseja usar está selecionado e clique em **OK**.

O Eclipse está agora configurado e pronto para você criar projetos e compilar e executar código Java. A próxima seção o familiariza com Eclipse.

Introdução ao Eclipse

O Eclipse é mais que um IDE; ele é um ecossistema de desenvolvimento completo. Esta seção é uma breve introdução prática para usar o Eclipse para desenvolvimento Java.

O ambiente de desenvolvimento Eclipse

O ambiente de desenvolvimento Eclipse tem quatro principais componentes:

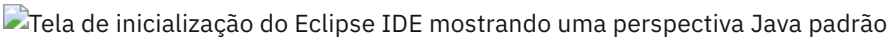
- Área de trabalho
- Projetos
- Perspectivas
- Visualizações

A unidade primária de organização no Eclipse é a *área de trabalho*. Uma área de trabalho contém todos os seus *projetos*. Uma *perspectiva* é uma forma de consulta a cada projeto (consequentemente o nome) e dentro de uma perspectiva há uma ou mais *visualizações*.

A perspectiva Java

A Figura 2 mostra a perspectiva Java, que é a perspectiva padrão para Eclipse. Você deverá ver essa perspectiva ao iniciar o Eclipse.

Perspectiva Eclipse Java


Tela de inicialização do Eclipse IDE mostrando uma perspectiva Java padrão

A perspectiva Java contém as ferramentas necessárias para começar a escrever aplicativos Java. Cada janela tabulada mostrada na Perspectiva Eclipse Java é uma visualização da perspectiva Java. O Package Explorer e o Outline são duas visualizações especialmente úteis.

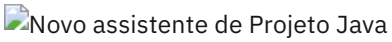
O ambiente Eclipse é altamente configurável. Cada visualização é adaptável, portanto, é possível movê-la na perspectiva Java e colocá-la onde desejar. Por enquanto, no entanto, aderiremos à perspectiva padrão e configuração de visualização.

Crie um projeto

Siga estas etapas para criar um novo projeto Java:

1. Clique em **Arquivo > Novo > Projeto Java...** para iniciar o assistente para Novo projeto Java, mostrado da Figura 3.

   Assistente de Novo projeto Java

   
   Novo assistente de Projeto Java

2. Insira `Tutorial` como o nome do projeto e clique em **Concluir**.

3. Se desejar modificar as configurações padrão do projeto, clique em **Avançar** (recomendado *apenas* se você tiver experiência com o Eclipse IDE).
4. Clique em **Concluir** para aceitar a configuração do projeto e criá-lo.

Agora você criou um novo projeto Eclipse Java e uma pasta de origem. Seu ambiente de desenvolvimento está pronto para ação. No entanto, um entendimento do paradigma OOP — abrangido nas próximas duas seções deste tutorial — é essencial. Se você estiver familiarizado com os conceitos e princípios de OOP, pode desejar pular para a seção "Introdução à linguagem Java".

Conceitos de programação orientada a objeto

A linguagem Java é (principalmente) orientada a objetos. Se você não usou uma linguagem orientada a objetos antes, os conceitos de OOP podem parecer estranhos à primeira vista. Esta seção é uma rápida introdução aos conceitos de linguagem OOP, usando programação estruturada como um ponto de contraste.

O que é um objeto?

Linguagens de programação estruturas, como C e COBOL, seguem um paradigma de programação diferente daquelas orientadas a objetos. O paradigma de programação estruturada é altamente orientado a dados: você possui estruturas de dados e instruções do programa atuam nesses dados. Linguagens orientadas a objetos, como a linguagem Java, combinam instruções de dados e programas em *objetos*.

Um objeto é uma entidade autocontida que contém atributos e comportamento, e nada mais. Em vez de ter uma estrutura de dados com campos (atributos) e transmite essa estrutura em toda a lógica do programa que atua nela (comportamento), em uma linguagem orientada a objetos, dados e lógica de programa são combinados. Essa combinação pode ocorrer em níveis completamente diferentes, de objetos com baixa granularidade, como `Number`, a objetos com alta granularidade, como um serviço `FundsTransfer` em um aplicativo financeiro amplo.

Objetos pai e filho

Um *objeto pai* é aquele que serve como base estrutural para derivação de *objetos-filho mais complexos*. Um objeto-filho assemelha-se ao seu pai, mas é mais especializado. Com o paradigma orientado a objetos, é possível reutilizar os atributos comuns e o comportamento do objeto pai,

incluindo nesses objetos-filho atributos e comportamentos diferentes. (Você aprende mais sobre *herança* na próxima seção deste tutorial.)

Comunicação e coordenação de objetos

Os objetos se comunicam enviando mensagens (*chamadas de métodos* na linguagem Java). Além disso, em um aplicativo orientado a objetos, o código do programa coordena as atividades entre objetos para executar tarefas dentro do contexto do domínio de aplicativo específico.

Resumo de objeto

Um objeto bem escrito:

- Tem limites nítidos
- Executa um conjunto finito de atividades
- Conhece apenas sobre seus dados e quaisquer outros objetos necessários para realizar suas atividades

Em essência, um objeto é uma entidade discreta que possui apenas as dependências necessárias em outros objetos para executar suas tarefas.

Agora, você vê como um objeto se parece.

O objeto `Person`

Eu começo com um exemplo que baseia-se em um cenário de desenvolvimento de aplicativo comum: um indivíduo sendo representado por um objeto `Person`.

Voltando à definição de um objeto, você sabe que um objeto tem dois elementos primários: atributos e comportamento. Eu mostro como eles se aplicam a `Person`.

Atributos

Quais atributos uma pessoa pode ter? Alguns atributos comuns incluem:

- Name
- Age
- Height
- Weight
- Eye color
- Gender

Você provavelmente pensará em outros (e pode sempre incluir mais atributos posteriormente), mas esta lista é um bom começo.

Comportamento

Uma pessoa real pode fazer todo o tipo de ação, mas comportamentos de objetos geralmente estão relacionados a algum tipo de contexto de aplicativo. Em um contexto de aplicativo de negócios, por exemplo, você pode querer perguntar ao seu objeto `Person`: “Qual é sua idade?” Em resposta, `Person` informaria o valor de seu atributo `Age`.

Lógica mais complexa pode estar oculta dentro do objeto `Person` —, por exemplo calcular um Índice de massa corpórea (IMC) de uma pessoa para um aplicativo de saúde, — mas, por enquanto, suponha que `Person` tenha o comportamento de responder a estas questões:

- Qual é seu nome?
- Qual é sua idade?
- Qual é sua altura?
- Qual é seu peso?
- Qual é a cor de seus olhos?
- Qual é seu sexo?

Estado e sequência

Estado é um conceito importante em OOP. Um estado de objeto é representado a qualquer momento pelo valor de seus atributos.

No caso de `Person`, seu estado é definido por atributos como nome, idade, altura e peso. Se você quisesse apresentar uma lista de vários desses atributos, poderia fazer isso usando uma classe `String`, da qual eu falarei mais posteriormente no tutorial.

Usando os conceitos de estado e sequência juntos, é possível dizer para `Person`, “Diga-me quem você é fornecendo-me uma listagem (ou `String`) de seus atributos.”

Princípios de OOP

Se você veio de um plano de fundo de programação estruturada, a proposição de valor de OOP pode ainda não estar muito clara. Afinal, os atributos de uma pessoa e qualquer lógica para recuperar (e converter) esses valores podem ser escritos em C ou COBOL. Esta seção esclarece os benefícios do paradigma de OOP explicando seus princípios de definição: *encapsulamento*, *herança* e *polimorfismo*.

Encapsulamento

Lembre-se de que um objeto é, acima de tudo, discreto ou autocontido. Essa característica é o princípio do *encapsulamento* em funcionamento. *Ocultação* é outro termo que às vezes é usado para expressar a natureza autocontida e protegida de objetos.

Independentemente da terminologia, o que é importante é que o objeto mantém um limite entre seu estado e o comportamento e o mundo externo. Como objetos no mundo real, objetos usados na

programação de computador possuem vários tipos de relacionamentos com diferentes categorias de objetos nos aplicativos que os usam.

Na plataforma Java, é possível usar *modificadores de acesso* (que eu apresento posteriormente no tutorial) para variar a natureza dos relacionamentos de objetos de *público* para *privado*. O acesso público é muito aberto, considerando que acesso privado significa que os atributos de objetos estão acessíveis apenas dentro do próprio objeto.

O limite público/privado impinge o princípio de encapsulamento orientado a objetos. Na plataforma Java, é possível variar a intensidade desse limite em uma base de objeto por objeto, dependendo de um sistema de confiança. O encapsulamento é um recurso poderoso da linguagem Java.

Herança

Na programação estruturada, é comum copiar uma estrutura, nomeá-la e incluir ou modificar os atributos que torna a nova entidade (como um registro `Account`) diferente de sua fonte original. Com o tempo, essa abordagem gera uma grande questão de código duplicado, o que pode criar problemas de manutenção.

O OOP apresenta o conceito de *herança*, pelo qual classes especializadas — sem código adicional — podem "copiar" os atributos e o comportamento de classes de origem na qual elas se especializam. Se alguns desses atributos ou comportamentos precisarem mudar, você os substitui. O único código-fonte a ser mudado é o código necessário para criar classes especializadas. Como vimos na seção "Conceitos de programação orientada a objeto", o objeto de origem é chamado de *pai* e a nova especialização é chamada de *filho*.

Herança em funcionamento

Suponha que você esteja programando um aplicativo de recursos humanos e queira usar a classe `Person` como base (chamada de *superclasse*) para uma nova classe chamada `Employee`. Sendo filha de `Person`, `Employee` teria todos os atributos de uma classe `Person`, junto com aqueles adicionais, como:

- Número de identificação de contribuinte
- Número de matrícula
- Salário

A herança facilita a criação da nova classe `Employee` sem a necessidade de copiar todo o código `Person` manualmente.

Você verá muitos exemplos de herança na programação Java posteriormente no tutorial, especialmente na Parte 2.

Polimorfismo

O polimorfismo é um conceito mais difícil de compreender do que o encapsulamento e a herança. Em essência, isso significa que objetos que pertencem à mesma ramificação de uma herança, quando enviam a mesma mensagem (ou seja, quando efetuam a mesma ação), podem manifestar esse comportamento de forma diferente.

Para entender como o polimorfismo se aplica a um contexto de aplicativo de negócios, retorne ao exemplo de `Person`. Lembre-se de dizer a `Person` para formatar seus atributos em uma `String`? O polimorfismo torna possível que `Person` represente seus atributos em uma variedade de formas, dependendo do tipo que `Person` é.

O polimorfismo é um dos conceitos mais complexos que você encontrará no OOP na plataforma Java e está além do escopo de um tutorial introdutório.

Introdução à linguagem Java

Seria impossível apresentar toda a sintaxe da linguagem Java em um único tutorial. O lembrete da Parte 1 foca nos fundamentos da linguagem, deixando você com conhecimento suficiente e prática para escrever programas simples. OOP trata exclusivamente de objetos, portanto, esta seção começa com dois tópicos especialmente relacionados a como a linguagem Java os manipula: palavras reservadas e a estrutura de um objeto Java.

A linguagem Java: não puramente orientada a objetos

Com a linguagem Java, é possível criar objetos de primeira classe, mas nem *tudo* na linguagem é um objeto. Duas qualidades diferenciam a linguagem Java das linguagens puramente orientadas a objetos, como Smalltalk. Primeiro, a linguagem Java é uma mistura de objetos e tipos primitivos. Segundo, com Java, é possível escrever código que expõe os trabalhos internos de um objeto a qualquer outro objeto que o usa.
A linguagem Java fornece a você as ferramentas necessárias para seguir os princípios de som do OOP e produzir código orientado a objetos de som. Como Java não é puramente orientada a objetos, é necessário ter alguma disciplina sobre como escrever o código A linguagem não o força a fazer a tarefa corretamente, portanto, você deve fazê-la por si mesmo. (A última seção deste tutorial, "Escrevendo um bom código Java," fornece dicas.)

Palavras reservadas

Como qualquer linguagem de programação, a linguagem Java designa determinadas palavras que o compilador reconhece como especiais. Por essa razão, você não pode usá-las para nomear suas construções Java. A lista de palavras reservadas é surpreendentemente curta:

- `abstract`
- `assert`
- `boolean`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `class`
- `const`
- `continue`
- `default`
- `do`
- `double`
- `else`
- `enum`
- `extends`
- `final`
- `finally`
- `float`
- `para`

- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

Observe que `true`, `false` e `null` são tecnicamente palavras não reservadas. Embora sejam literais, eu as inclui nesta lista porque você não pode usá-las para nomear construções Java.

Uma vantagem da programação com um IDE é que ela pode usar a coloração de sintaxe para palavras reservadas, como eu mostro posteriormente neste tutorial.

Estrutura de uma classe Java

Uma classe é um blueprint para uma entidade discreta (objeto) que contém atributos e comportamentos. A classe define a estrutura básica do objeto e, no tempo de execução, seu aplicativo cria uma instância do objeto. Um objeto tem um limite e um estado nítidos e pode fazer ações quando corretamente solicitado. Cada linguagem orientada a objetos possui regras sobre como definir uma classe.

Na linguagem Java, classes são definidas conforme mostrado na Listagem 1:

Definição de classe

```
package packageName;
import ClassNameToImport; accessSpecifier class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName ([argumentList]) {
    methodStatement(s)
  }
  // This is a comment
  /* This is a comment too */
  /* This is a
   multiline
     comment */
}
```

Mostrar mais

Definição de classe contém vários tipos de construções, que eu diferenciei com a formatação de fonte. As construções mostradas em negrito (que você localizará na lista de Palavras reservadas) são literais; em qualquer definição de objeto, elas devem ser exatamente o que estão na listagem. Os nomes fornecidos para outras construções descrevem os conceitos que eles representam. Eu expliquei todas as construções detalhadamente no restante desta seção.

Observação: Na Definição de classe e em alguns outros exemplos de códigos nesta seção, colchetes indicam que as construções dentro deles não são requeridas. Os colchetes (diferente de { e }) não fazem parte da sintaxe Java.

Comentários no código

Observe que Definição de classe também inclui algumas linhas de comentário:

```
// This is a comment
/* This is a comment too */
/* This is a
multiline
comment */
```

Mostrar mais

Com informações apenas sobre cada linguagem de programação, os programadores podem incluir comentários para ajudar a documentar o código. A sintaxe Java permite comentários de única linha e multilinhas. Um comentário de única linha deve estar contido em uma linha, embora seja possível usar comentários de única linha adjacentes para formar um bloco. Um comentário de multilinhas começa com `/*`, deve terminar com `*/` e pode abranger qualquer número de linhas.

Você obtém mais informações sobre comentários quando chega à seção “Escrevendo um bom código Java” deste tutorial.

Empacotando classes

Com a linguagem Java, é possível escolher os nomes de suas classes, como `Account`, `Person` ou `LizardMan`. Às vezes, é possível terminar usando o mesmo nome para expressar dois conceitos um

pouco diferentes. Esta situação é chamada de *colisão de nomes* e ocorre com frequência. A linguagem Java usa *pacotes* para resolver esses conflitos.

Um pacote Java é um mecanismo que fornece um *namespace* — de uma área dentro da qual nomes são exclusivos, mas fora dela eles podem não ser. Para identificar uma construção exclusivamente, é necessário qualificá-la incluindo seu namespace.

Pacotes também fornecem a você uma boa forma de criar aplicativos mais complexos com unidades de funcionalidade discretas.

Definição de pacote

Para definir um pacote, você usa a palavra-chave `package` seguida de um nome de pacote legal, terminando com um ponto e vírgula. Geralmente os nomes de pacotes são separados por pontos e seguem este esquema padrão *de fato*:

```
package orgType.orgName.appName.compName;
```

Mostrar mais

Esta definição de pacote é dividida como a seguir:

- `orgType` é o tipo de organização, como `com`, `org` ou `net`.
- `orgName` é o nome do domínio da organização, como `makotojava`, `oracle` ou `ibm`.
- `appName` é o nome do aplicativo, abreviado.
- `compName` é o nome do componente.

A linguagem Java não força você a seguir esta convenção de pacote. De fato, não é necessário especificar um pacote, nesse caso, todas as suas classes deverão ter nomes exclusivos e residirão no pacote padrão. Como melhor prática, eu recomendo que você defina todas as suas classes Java em pacotes nomeados conforme descrito aqui. Você segue essa convenção em todo este tutorial.

Instruções de importação

Próxima à definição de classe (referindo-se novamente à Definição de classe) está a *instrução de importação*. Uma instrução de importação informa ao compilador Java onde localizar classes às quais você fez referência dentro de seu código. Qualquer classe não trivial usa outras classes para alguma funcionalidade e a instrução de importação é como você informa ao compilador Java sobre elas.

Uma instrução de importação geralmente assemelha-se ao seguinte:

```
import ClassNameToImport;
```

Mostrar mais

O Eclipse simplifica as importações

Ao escrever código no editor de Eclipse, você pode digitar o nome de uma classe que deseja usar, seguido por

Eclipse localizar duas classes com o mesmo

nome, ele exibirá uma caixa de diálogo perguntando em qual classe você deseja incluir importações.

Você especifica a palavra-chave `import`, seguida pela classe que deseja importar, seguida por um ponto e vírgula. O nome da classe deve ser *completo*, o que significa que ele deve incluir seu pacote.

Para importar todas as classes dentro de um pacote, é possível inserir `.*` após o nome do pacote. Por exemplo, essa instrução importa cada classe no pacote `com.makotojava`:

```
import com.makotojava.*;
```

Mostrar mais

A importação de um pacote integral pode tornar seu código menos legível, portanto, eu recomendo importar apenas as classes necessárias, usando seus nomes completos.

Declaração de classe

Para definir um objeto na linguagem Java, você deve declarar uma classe. Novamente, pense em uma classe como um modelo para um objeto, como um cortador de cookie.

Definição de classe inclui esta declaração de classe:

```
accessSpecifier class ClassName {
  accessSpecifier dataType variableName [= initialValue];
    accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName([argumentList]) {
    methodStatement(s)
  }
}
```

Mostrar mais

Um `accessSpecifier` da classe pode ter diversos valores, mas ele geralmente é `public`. Você logo consulta outros valores de `accessSpecifier`.

Convenções de nomenclatura de classe

É possível nomear classes basicamento como você realmente deseja, mas a convenção é usar *camel case*: comece com uma letra maiúscula, altere para letras maiúsculas concatenada e torne todas as outras letras minúsculas. Nomes de classes devem conter apenas letras e números. A aderência a estas diretrizes garante que seu código seja mais acessível a outros desenvolvedores que estão seguindo as mesmas convenções.

Classes podem ter dois tipos de membros: *variáveis* e *métodos*.

Variáveis

Os valores de variáveis de uma classe específica distinguem cada instância daquela classe e definem seu estado. Esses valores geralmente são referidos como *variáveis de instância*. Uma variável possui:

- Um `accessSpecifier`
- Um `dataType`
- Um `variableName`
- Opcionalmente, um `initialValue`

Os valores possíveis de `accessSpecifier` são:

- `public`: qualquer objeto em qualquer pacote pode ver a variável. (Não use de forma alguma este valor; consulte a barra lateral Variáveis públicas.)
- `protected`: qualquer objeto definido no mesmo pacote ou uma subclasse (definida em qualquer pacote) pode ver a variável.
- Nenhum especificador (também chamado de acesso *amigável* ou *pacote privado*): apenas objetos cujas classes são definidas no mesmo pacote podem ver a variável.
- `private`: apenas a classe contendo a variável pode vê-la.

Variáveis públicas

Nunca é uma boa ideia usar variáveis públicas, mas em casos extremamente raros, elas podem ser necessárias, portanto, existe a opção. A plataforma Java não limita seus casos de uso, portanto cabe a você ser disciplinado sobre o uso de boas convenções de codificação, mesmo quando instigado a fazer de outra forma.

Um `dataType` de variável depende do que a variável é, — ela pode ser um tipo primitivo ou outro tipo de classe (novamente, mais sobre isso posteriormente).

O `variableName` está ativo para você, mas, por convenção, nomes de variáveis usam a convenção de camel case que eu descrevi em "Convenções de nomenclatura de classe," exceto pelo fato de começarem com uma letra minúscula. (Este estilo é às vezes chamado de *Camel Case iniciado por letra minúscula.*)

Não se preocupe com o `initialValue` por enquanto; apenas saiba que é possível inicializar uma variável de instância ao declará-la. (Caso contrário, o compilador gerará um padrão para você que será configurado quando a classe for instanciada.)

Exemplo: definição de classe para `Person`

Antes de falarmos de métodos, veja um exemplo que resume o que você aprendeu até aqui. Listagem 2 é uma definição de classe para `Person`.

Definição básica de classe para `Person`

```
package com.makotojava.intro;

public class Person {
   private String name;
   private int age;
   private int height;
   private int weight;
   private String eyeColor;
   private String gender;
}
```

Mostrar mais

A definição básica de classe para `Person` não é útil neste ponto, pois define apenas seus atributos (e os atributos privados que estão nela).

Para que seja mais interessante, a classe `Person` precisa de comportamento — e isso significa *métodos*.

Métodos

Os métodos de uma classe definem seu comportamento.

Métodos se enquadram em duas categorias principais: *construtores* ou todos os outros métodos, dos quais existem muitos tipos. Um método construtor é usado apenas para criar uma instância de uma classe. Outros tipos de métodos podem ser usados virtualmente para qualquer comportamento do aplicativo.

A definição de classe na Definição de classe mostra a forma de definir a estrutura de um método, que inclui elementos como:

- `accessSpecifier`
- `returnType`
- `methodName`
- `argumentList`

A combinação desses elementos estruturais em uma definição de método é chamada de *assinatura de método*.

Depois, você consulta mais detalhadamente os dois tipos de métodos, começando com construtores.

Métodos construtores

Você usa construtores para especificar como instanciar uma classe. Definição de classe mostra a sintaxe de declaração do construtor de forma abstrata; novamente:

```
accessSpecifier ClassName([argumentList]) {
  constructorStatement(s)
}
```

Mostrar mais

Um `accessSpecifier` do construtor é igual ao das variáveis. O nome do construtor deve corresponder ao nome da classe. Portanto, se você chamar sua classe de `Person`, o nome do construtor também deverá ser `Person`.

Para qualquer construtor que não seja o construtor padrão, você transmite `argumentList`, que é um ou mais dos seguintes:

Construtores são opcionais

Se você não fornecer um construtor, o compilador fornecerá um para você, chamado de construtor

```
argumentType argumentName
```

Mostrar mais

Argumentos em um `argumentList` são separados por vírgulas e nenhum dos dois argumentos pode ter o mesmo nome. `argumentType` é um tipo primitivo ou outro tipo de classe (igual com tipos de variáveis).

um construtor sem argumento (ou *no-arg*), o compilador não gerará automaticamente um para você.

Definição de classe com um construtor

Agora, você vê o que acontece quando inclui a capacidade para criar um objeto `Person` de duas formas: usando um construtor no-arg e inicializando uma lista parcial de atributos.

Listagem 3 mostra como criar construtores e também como usar `argumentList`:

`Person`: definição de classe com um construtor

```
package com.makotojava.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int  weight;
  private String eyeColor;

  private String gender;
  public Person() {
    // Nothing to do...
  }

  public Person(String name, int age, int height, int weight String eyeColor, St
    this.name = name;
```

Mostrar mais

Observe o uso da palavra-chave `this` ao fazer designações de variáveis na Person: definição de classe com um construtor. A palavra-chave `this` é uma abreviação Java para “this object,” e você deve usá-la ao fazer referência para duas variáveis com o mesmo nome. Neste caso, age é um parâmetro de construtor e uma variável de classe, portanto, a palava-chave `this` ajuda o compilador a desambiguar a referência.

O objeto `Person` está ficando mais interessante, mas precisa de mais comportamento. E, para isso, você precisa de mais métodos.

Outros métodos

Um construtor é um tipo particular de método com uma função específica. Similarmente, muitos outros tipos de métodos executam funções específicas em programas Java. A exploração de outros tipos de métodos começa nesta seção e continua pelo tutorial.

De volta à Definição de classe, eu mostrei como declarar um método:

```
accessSpecifier returnType methodName ([argumentList]) {
  methodStatement(s)
```

```
}
```

Mostrar mais

Outros métodos assemelham-se muito aos construtores, com algumas exceções. Primeiro, é possível nomear outros métodos como deseja (embora, claro, determinadas regras se apliquem). Eu recomendo as seguintes convenções:

- Comece com uma letra minúscula.
- Evite números, a menos que eles sejam absolutamente necessários.
- Use apenas caracteres alfabéticos.

Segundo, diferente dos construtores, outros métodos possuem um tipo de retorno opcional.

`Outros métodos de Person`

Munido destas informações básicas, é possível ver na Listagem 4 o que ocorre quando você inclui alguns métodos a mais em `Person`. (Para resumir, eu omiti os construtores.)

`Person` com alguns novos métodos

```
package com.makotojava.intro;

public class Person {
   private String name;
   private int age;
   private int height;
   private int  weight;
   private String eyeColor;
   private String gender;

   public String getName() { return name; }
   public void setName(String value) { name = value; }
   // Other getter/setter combinations...
}
```

Mostrar mais

Observe o comentário na `Person com alguns novos métodos` sobre “combinações de getter/setter.” Você trabalhará mais com getters e setters posteriormente no tutorial. Por enquanto, tudo o que você precisa saber é que um *getter* é um método para recuperar o valor de um atributo e um *setter* é um método para modificar esse valor. A Listagem 4 mostra apenas uma combinação de getter/setter (para o atributo `Name`), mas é possível definir mais de uma forma semelhante.

Observe na `Person com alguns novos métodos` que se um método não retornar um valor, você deverá informar ao compilador especificando o tipo de retorno `void` em sua assinatura.

Métodos estáticos e de instância

Geralmente, dois tipos de métodos (sem construtor) são usados: *métodos de instância* e *métodos estáticos*. Métodos de instância dependem do estado de uma instância de objeto específica para seus comportamentos. Métodos estáticos também são às vezes chamados de *métodos de classes*, pois seus

comportamentos não dependem de qualquer estado do objeto. O comportamento de um método estático ocorre no nível de classe.

Métodos estáticos são amplamente usados para utilidade; você pode pensar neles como sendo métodos globais (à la C) enquanto mantém o código para o método com a classe que o define.

Por exemplo, neste tutorial, você usa a classe JDK `Logger` para informações de saída para o console. Para criar uma instância da classe `Logger`, você não instancia uma classe `Logger`; ao contrário, você chama um método estático chamado `getLogger()`.

A sintaxe para chamar um método estático em uma classe é diferente da sintaxe usada para chamar um método em um objeto. Você também usa o nome da classe que contém o método estático, conforme mostrado nesta chamada:

```
Logger l = Logger.getLogger("NewLogger");
```

Mostrar mais

Neste exemplo, `Logger` é o nome da classe e `getLogger(...)` é o nome do método. Assim, para chamar um método estático, não é necessário uma instância de objeto, apenas o nome da classe.

Sua primeira classe Java

É hora de compilar tudo o que você aprender nas seções anteriores e começar a escrever algum código. Esta seção o orienta pela declaração de uma classe e inclui variáveis e métodos usando o Eclipse Package Explorer. Você aprende a usar a classe `Logger` para observar o comportamento de seu aplicativo e também aprende a usar um método `main()` como uma rotina de teste.

Criando um pacote

Se você ainda não estiver neste ponto, vá para a visualização do Package Explorer (na perspectiva Java) no Eclipse usando **Janela > Perspectiva > Abrir perspectiva**. Você obterá a configuração para criar sua primeira classe Java. A primeira etapa é criar um local para a classe residir. Pacotes são construções de namespace e também são mapeados de forma conveniente diretamente para a estrutura de diretórios do sistema de arquivos.

Em vez de usar o pacote padrão (quase sempre uma má ideia), você cria um especificamente para o código que está escrevendo. Clique em **Arquivo > Novo > Pacote** para iniciar o assistente Java Package, mostrado na Figura 4:

O assistente Eclipse Java Package


Assistente do Pacote Eclipse

Digite `com.makotojava.intro` na caixa de texto Nome e clique em **Concluir**. Você verá o novo pacote criado no Package Explorer.

Declarando a classe

É possível criar uma classe a partir do Package Explorer de mais de uma forma, mas a forma mais fácil é clicar com o botão direito no pacote que acabou de criar e escolher **Nova > Classe...**. A caixa de diálogo Nova classe é aberta.

Na caixa de texto Nome, digite `Person` e clique em **Concluir**.

A nova classe é exibida em sua janela de edição. Eu recomendo fechar algumas das visualizações (Problemas, Javadoc e outras) abertas por padrão na perspectiva Java na primeira vez que abri-la para facilitar a visualização de seu código-fonte. (O Eclipse lembra que você não deseja ver aquelas visualizações na próxima vez que abrir o Eclipse e vai para a perspectiva Java.) Figura 5 mostra uma área de trabalho com as visualizações essenciais abertas.

Uma área de trabalho bem ordenada


Janela de edição do assistente do Pacote Eclipse

O Eclipse gera uma classe shell para você e inclui a instrução `package` no início. Você só precisa implementar a classe agora. É possível configurar como o Eclipse gerará novas classes usando **Janela > Preferências > Java > Estilo de código > Modelos de código**. Para facilitar, acompanhe a geração de códigos simples de instalar do Eclipse.

Na Uma área de trabalho bem ordenada, observe o asterisco (*) próximo ao novo nome de arquivo de código-fonte, indicando que eu fiz uma modificação. E observe que o código não foi salvo. Depois, observe que eu cometi um erro ao declarar o atributo `Name`: eu declarei o tipo de `Name` como se fosse `Strin`. O compilador não pôde localizar uma referência para esse tipo de classe e o sinalizou como um erro de compilação (a linha ondulada vermelha abaixo de `Strin`). Claro, eu posso corrigir meu erro incluindo um g ao final de `Strin`. Esta é uma pequena demonstração do poder do uso de um IDE em vez de ferramentas de linha de comandos para desenvolvimento de software. Continue e corrija o erro mudando o tipo para `String`.

Incluindo variáveis de classe

Na `Person`: definição de classe com um construtor, você começa a implementar a classe `Person`, mas eu não expliquei muito da sintaxe. Agora, eu formalmente defino como incluir variáveis de classe.

Lembre-se de que uma variável possui um `accessSpecifier`, um `dataType`, um `variableName` e, opcionalmente, um `initialValue`. Anteriormente, você viu brevemente como definir `accessSpecifier` e `variableName`. Agora, verá o `dataType` que uma variável pode ter.

Um `dataType` pode ser um tipo primitivo ou uma referência para outro objeto. Por exemplo, observe que `Age` é um `int` (um tipo primitivo) e que `Name` é um `String` (um objeto). O JDK é fornecido cheio de classes úteis como `java.lang.String` e aquelas no pacote `java.lang` não precisam ser importadas (uma pequena cortesia do compilador Java). Mas se o `dataType` for uma classe JDK como `String` ou uma classe definida pelo usuário, a sintaxe será essencialmente a mesma.

A Tabela 1 mostra os oito tipos de dados primitivos que você provavelmente verá regularmente, incluindo os valores padrão que essas primitivas obtêm se você não inicializar explicitamente um valor de variável do membro.

Tipos de dados primitivos

Tipo	Tamanho	Valor padrão	Faixa de valores
`boolean`	n/a	false	true ou false
`byte`	8 bits	0	-128 a 127
`char`	16 bits	(não designado)	\u0000′ \u0000′ a \uffff’ ou 0 a 65535
`short`	16 bits	0	-32768 a 32767
`int`	32 bits	0	-2147483648 a 2147483647
`long`	64 bits	0	-9223372036854775808 a 9223372036854775807
`float`	32 bits	0,0	1.17549435e-38 a 3.4028235e+38
`double`	64 bits	0,0	4.9e-324 a 1.7976931348623157e+308

Criação de log integrada

Antes de continuar na codificação, você precisa saber como seus programas informam o que estão fazendo.

A plataforma Java inclui o pacote `java.util.logging`, um mecanismo de criação de log integrado para reunir informações do programa de uma forma legível. Criadores de log são entidades nomeadas que você cria usando uma chamada de método estática para a classe `Logger`:

```
import java.util.logging.Logger;
//...
Logger l = Logger.getLogger(getClass().getName());
```

Mostrar mais

Ao chamar o método `getLogger()`, você transmite a ele um `String`. Por enquanto, apenas tenha o hábito de transmitir o nome da classe na qual o código que está escrevendo está localizado. De qualquer método regular (ou seja, não estático), o código anterior sempre faz referência ao nome da classe e o transmite para `Logger`.

Se você estiver fazendo uma chamada de `Logger` dentro de um método estático, faça referência ao nome da classe na qual está:

```
Logger l = Logger.getLogger(Person.class.getName());
```

Mostrar mais

Neste exemplo, o código no qual você está é a classe `Person`, portanto, você faz referência a um literal especial chamado `class` que recupera o objeto `Class` (mais sobre isso posteriormente) e obtém seu atributo `Name`.

A seção "Escrevendo um bom código Java" deste tutorial inclui uma dica sobre como *não* efetuar a criação de log.

Antes de chegarmos ao corpo do teste, primeiro vá ao editor de código-fonte do Eclipse para `Person` e inclua esse código depois de `public class Person {` a partir da `Person`: definição de classe com um construtor, de modo que fique semelhante ao seguinte:

```
package com.makotojava.intro;

public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
}
```

Mostrar mais

O Eclipse tem um gerador de código útil para gerar getters e setters (entre outras coisas). Para experimentar o gerador de código, coloque o cursor do mouse na definição de classe `Person` (ou seja, na palavra `Person` na definição de classe) e clique em **Origem > Gerar getters e setters...**. Quando a caixa de diálogo abrir, clique em **Selecionar tudo**, conforme mostrado na Figura 6.

Eclipse gerando getters e setters

Para o ponto de inserção, escolha **Último membro** e clique em **OK**.

Agora, inclua um construtor em `Person` digitando o código a partir da Listagem 5 em sua janela de origem, logo abaixo da parte superior da definição de classe (a linha imediatamente abaixo de `public class Person ()`).

`Construtor Person`

```
public Person(String name, int age, int height, int weight, String eyeColor, String gender) {
  this.name = name;
  this.age = age;
  this.height = height;
  this.weight = weight;
  this.eyeColor = eyeColor;
  this.gender = gender;
}
```

Mostrar mais

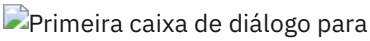
Certifique-se de não ter nenhuma linha ondulada indicando erros de compilação.

Gere um caso de teste JUnit

Agora você gera um caso de teste JUnit no qual instanciar um `Person`, usando o construtor na Listagem 5, e depois imprime o estado do objeto para o console. Nesse sentido, o "teste" certifica-se de que a ordem dos atributos na chamada do construtor esteja correta (ou seja, que eles estejam configurados para os atributos corretos).

No Package Explorer, clique com o botão direito em sua classe `Person` e depois clique em **Novo > Caso de teste JUnit**. A primeira página do assistente Novo caso de teste JUnit é aberta, conforme mostrado na Figura 7.

Criando um caso de teste JUnit


Primeira caixa de diálogo para criar um caso de teste do JUnit

Aceite os padrões clicando em **Avançar**. Você vê a caixa de diálogo Métodos de teste, mostrada na Figura 8.

Selecione métodos para o assistente para gerar casos de teste


Caixa de diálogo Métodos de teste para criar um caso de teste do JUnit

Nesta caixa de diálogo, selecione o método ou os métodos para os quais deseja que o assistente faça testes. Neste caso, selecione apenas o construtor, conforme mostrado na Figura 8. Clique em **Concluir** e o Eclipse gerará o caso de teste JUnit.

Depois, abra `PersonTest`, vá para o método `testPerson()` e deixe-o semelhante à Listagem 6.

O método `testPerson()`

```
@Test
public void testPerson() {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  Logger l = Logger.getLogger(Person.class.getName());
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
  assertEquals("Joe Q Author", p.getName());
  assertEquals(42, p.getAge());
  assertEquals(173, p.getHeight());
  assertEquals(82, p.getWeight());
  assertEquals("Brown", p.getEyeColor());
```

Mostrar mais

Usando main() como uma rotina de teste

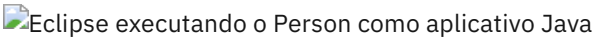
`main()` é um método especial que você pode incluir em qualquer classe para que o JRE possa executar seu código. Uma classe não precisa ter um método `main()` de fato, a maioria nunca terá e pode ter no máximo um `main()`. `main()` é um método útil para se ter, pois fornece uma rotina de teste rápida para a classe. No desenvolvimento corporativo, você usaria bibliotecas de teste como JUnit, mas usar `main()` como sua rotina de teste pode ser uma forma mais rápida e simples de criar uma rotina de teste.

Não se preocupe com a classe `Logger` por enquanto. Apenas insira o código como o vê na O método `testPerson()`. Agora você está pronto para executar seu primeiro programa Java (e caso de teste JUnit).

Executando código em Eclipse

Para executar um aplicativo Java de dentro do Eclipse, selecione a classe que deseja executar, depois clique no ícone **Executar** (que está em verde e tem uma pequena seta triangular apontando para direita). O Eclipse é suficientemente inteligente para reconhecer que a classe que você deseja executar é um caso de teste JUnit e, portanto, ativa JUnit. Agora, sente-se e observe. A Figura 9 mostra o que acontece.

Veja a execução de `Person`


Eclipse executando o Person como aplicativo Java

A visualização Console abre automaticamente e mostra a saída `Logger`. A visualização JUnit também é aberta para mostrar os resultados do teste.

Incluindo comportamento em uma classe Java

`Person` parece muito bom até agora, mas algum comportamento adicional pode ser usado para torná-lo mais interessante. Criando meios de comportamento incluindo métodos. Esta seção aparece mais detalhadamente em *métodos acessadores* —, os getters e setters já vistos em ação.

Métodos acessadores

Para conter dados de uma classe de outros objetos, você declara suas variáveis como sendo `private` e depois fornece métodos acessadores. Como visto, um getter é um método acessador para recuperar o valor de um atributo; um setter é um método acessador para modificar esse valor. A nomenclatura dos acessadores segue uma convenção estrita conhecida como o padrão JavaBeans, pelo qual qualquer atributo `Foo` tem um getter chamado `getFoo()` e um setter chamado `setFoo()`.

O padrão JavaBeans é tão comum que o suporte para ele é desenvolvido em Eclipse IDE. Você já o viu em ação — quando gerou getters e setters para `Person` na seção anterior.

Os acessadores seguem estas diretrizes:

- O próprio atributo é sempre declarado com acesso `private`.
- O especificador de acesso para getters e setters é `public`.
- Getters não obtêm nenhum parâmetro e retornam um valor cujo tipo é igual ao do atributo que eles acessam.
- As configurações apenas obtêm um parâmetro, do tipo de atributo, e não retornam um valor.

Declarando acessadores

De longe a forma mais fácil de declarar acessadores é permitir que o Eclipse faça isso para você, como mostrado na Eclipse gerando getters e setters. Mas você deve também saber como codificar manualmente um par de getter e setter. Suponha que você tenha um atributo, `Foo`, cujo tipo seja `java.lang.String`. Uma declaração completa para ele (seguindo as diretrizes do acessador) seria:

```
private String foo;
public String getFoo() {
  return foo;
}
public void setFoo(String value) {
  foo = value;
}
```

Mostrar mais

É possível observar que o valor de parâmetro transmitido ao setter foi nomeado de forma diferente daquele gerado pelo Eclipse. A nomenclatura segue sua própria convenção, que eu recomendo para outros desenvolvedores. Nos raros casos que eu codifico manualmente um setter, sempre uso o nome `value` como o valor de parâmetro para o setter. Esse destaque me lembra que eu codifiquei manualmente o setter. Como geralmente eu permito que o Eclipse gere getters e setters para mim, quando isso não ocorre, é por uma boa razão. Usar `value` como o valor de parâmetro de setter lembra-me de que esse setter é especial. (Comentários de códigos também fazem isso.)

Chamando métodos

Chamar métodos é fácil. Você viu na O método `testPerson()` como chamar os diversos getters de `Person` para retornar seus valores. Agora, eu formalizo os mecanismos de chamadas de métodos.

Chamada de método com e sem parâmetros

Para chamar um método em um objeto, é necessário uma referência a esse objeto. A sintaxe de chamada de método abrange a referência de objeto, um ponto literal, o nome do método e todos os parâmetros que precisam ser transmitidos:

```
objectReference.someMethod();
objectReference.someOtherMethod(parameter);
```

Mostrar mais

A seguir, uma chamada de método sem parâmetros:

```
Person p = /*obtain somehow */;
p.getName();
```

Mostrar mais

E, aqui, uma chamada de método com parâmetros (acessando o atributo `Name` de `Person`):

```
Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

Mostrar mais

Lembre-se de que construtores são métodos também. E é possível separar os parâmetros com espaços e novas linhas. O compilador Java não se importa. Essas duas próximas chamadas de métodos são idênticas:

```
new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

Mostrar mais

```
new Person("Joe Q Author",// Name
  42,     // Age
  173,    // Height in cm
  82,     // Weight in kg
  "Brown",// Eye Color
  "MALE");// Gender
```

Mostrar mais

Observe como os comentários na segunda chamada do construtor fornece capacidade de leitura para o próximo desenvolvedor. Rapidamente essa pessoa pode dizer ao que cada parâmetro se destina.

Chamada de método aninhada

Chamadas de métodos também podem ser aninhadas:

```
Logger l = Logger.getLogger(Person.class.getName());
l.info("Name: " + p.getName());
```

Mostrar mais

Aqui você está transmitindo o valor de retorno `Person.class.getName()` para o método `getLogger()`. Lembre-se de que a chamada de método `getLogger()` é uma chamada de método estática, portanto sua sintaxe difere um pouco. (Não é necessário uma referência de `Logger` para fazer a chamada; no lugar, você usa o nome da classe do lado esquerdo da chamada.)

Isso é realmente tudo o que há para chamada de método.

Strings e operadores

O tutorial até agora apresentou várias variáveis do tipo `String`, mas sem muita explicação. Você aprendeu mais sobre variáveis string nesta seção e também descobriu quando e como usar operadores.

Strings

Lidar com variáveis string em C é muito trabalhoso, pois elas são matrizes com terminação nula de caracteres de 8 bits que você deve manipular. Na linguagem Java, variáveis string são objetos de primeira classe do tipo `String`, com métodos que ajudam a manipulá-los. (O código Java mais próximo à linguagem C em relação a variáveis string é o tipo de dado primitivo `char`, que pode manter um único caractere Unicode, como *a*.)

Você já viu como instanciar um objeto `String` e configurar seu valor (voltando à `Person` com alguns novos métodos), mas há várias outras formas de fazer isso. Há várias formas de criar uma instância `String` com um valor de `hello`:

```
String greeting = "hello";
```

Mostrar mais

```
greeting = new String("hello");
```

Mostrar mais

Como `String`s são objetos de primeira classe na linguagem Java, você pode usar `new` para instanciá-los. A configuração de uma variável do tipo `String` tem o mesmo resultado, pois a linguagem Java cria um objeto `String` para manter o literal, depois designa esse objeto à variável de instância.

Concatenando strings

É possível efetuar várias ações com `String` e a classe tem muitos métodos úteis. Mesmo sem usar um método, você já fez algo interessante com dois `String`s concatenando-os ou combinando-os:

```
l.info("Name: " + p.getName());
```

Mostrar mais

O sinal de soma (+) é uma abreviação para concatenar `String`s na linguagem Java. (Pode haver penalidade no desempenho ao fazer esse tipo de concatenação dentro de um loop, mas por enquanto, não é necessário se preocupar com isso.)

Exemplo de concatenação

Agora, você pode tentar concatenar `String`s dentro de `Person`. Neste ponto, você possui uma variável de instância `name`, mas seria adequado ter um `firstName` e `lastName`. É possível então concatená-los quando outro objeto solicitar o nome completo de `Person`.

A primeira coisa a fazer é incluir as novas variáveis de instância (no mesmo local do código-fonte no qual `name` está atualmente definido):

```
//private String name;
private String firstName;
private String lastName;
```

Mostrar mais

Não é mais necessário ter `name`; ele foi substituído por `firstName` e `lastName`.

Encadeando chamadas de métodos

Agora, é possível gerar getters e setters para `firstName` e `lastName` (conforme mostrado na Eclipse gerando getters e setters), remover o método `setName()` e mudar `getName()` para se assemelhar ao seguinte:

```
public String getName() {
  return firstName.concat(" ").concat(lastName);
}
```

Mostrar mais

Este código ilustra o *encadeamento* de chamadas de métodos. O encadeamento é uma técnica comumente usada com objetos imutáveis como `String`, na qual uma modificação para um objeto imutável sempre retorna a modificação (mas não muda o original). Você opera então com o valor retornado, mudado.

Operadores

Como você pode esperar, a linguagem Java pode ser aritmética e você já viu como designar variáveis. Agora, daremos uma rápida olhada em alguns operadores de linguagem Java que você precisará à medida que suas habilidades melhoram. A linguagem Java usa dois tipos de operadores:

- *Unário*: apenas um operando é necessário.
- *Binário*: dois operandos são necessários.

Os operadores aritméticos da linguagem Java estão resumidos na Tabela 2.

Operadores aritméticos da linguagem Java

Operador	Uso	Descrição
+	a + b	Inclui a e b
+	+a	Promove a para `int` se ele for um `byte`, `short` ou `char`
-	a - b	Subtrai b de a
-	-a	Nega aritmeticamente a
*	a * b	Multiplica a e b
/	a / b	Divide a por b

Operador	Uso	Descrição
`%`	`a % b`	Retorna o restante da divisão de `a` por `b` (o operador de módulo)
`++`	`a++`	Incrementa `a` por 1; calcula o valor de `a` antes de incrementar
`++`	`++a`	Incrementa `a` por 1; calcula o valor de `a` depois de incrementar
`--`	`a--`	Decrementa `a` por 1; calcula o valor de `a` antes do decremento de
`--`	`--a`	Decrementa `a` por 1; calcula o valor de `a` após decremento de
`+=`	`a += b`	Abreviação de `a = a + b`
`-=`	`a -= b`	Abreviação de `a = a - b`
`*=`	`a *= b`	Abreviação de `a = a * b`
`%=`	`a %= b`	Abreviação de `a = a % b`

Operadores adicionais

Além dos operadores na Tabela 2, você viu diversos outros símbolos que são chamados de operadores na linguagem Java, incluindo:

- Ponto (`.`), que qualifica nomes de pacotes e chama métodos
- Parênteses (`()`), que delimita uma lista de parâmetros separados por vírgula para um método
- `new`, que (quando seguido por um nome de construtor) instancia um objeto

A sintaxe de linguagem Java também inclui diversos operadores que são usados especificamente para programação condicional — ou seja, programas que respondem de forma diferente com base na entrada diferente. Você vê sobre eles na próxima seção.

Operadores condicionais e instruções de controle

Nesta seção, você aprende sobre as diversas instruções e operadores que pode usar para informar aos programas Java como deseja que eles atuem com base em entrada diferente.

Operadores relacionais e condicionais

A linguagem Java fornece operadores e instruções de controle que podem ser usados para a tomada de decisões em seu código. Mais frequentemente, uma decisão no código inicia com uma *expressão booleana* (ou seja, uma que seja avaliada como true ou false). Essas expressões usam *operadores relacionais*, que compraram um operando ou uma expressão a outra, e *operadores condicionais*.

Tabela 3 lista os operadores relacionais e condicionais da linguagem Java.

Operadores relacionais e condicionais

Operador	Uso	Retorna true se...
>	a > b	a for maior que b
>=	a >= b	a é maior que ou igual a b
<	a < b	a é menor que b
<=	a <= b	a é menor que ou igual a b
==	a == b	a é igual a b
!=	a != b	a não é igual a b
&&	a && b	a e b são true, condicionalmente avalia b (se a for false, b não será avaliado)
`	`	`a	b`	a ou b é true; condicionalmente avalia b (se a for true, b não será avaliado)
!	!a	a é false
&	a & b	a e b são true; sempre avalia b
`	`	`a	b`	a ou b é true; sempre avalia b
^	a ^ b	a e b são diferentes

A instrução `if`

Agora que você tem alguns operadores, é hora de usá-los. Este código mostra o que ocorre quando você inclui alguma lógica para o acessador `getHeignt()` do objeto `Person`:

```
public int getHeight() {
  int ret = height;
```

```
  // If locale of the machine this code is running on is U.S.,
  if (Locale.getDefault().equals(Locale.US))
    ret /= 2.54;// convert from cm to inches
  return ret;
}
```

Mostrar mais

Se o código de idioma atual for Estados Unidos (em que o sistema de métrica não é usado), pode fazer sentido converter o valor interno de `height` (em centímetros) para polegadas. Este exemplo (um pouco controverso) ilustra o uso da instrução `if`, que avalia uma expressão booleana entre parênteses. Se essa expressão for avaliada como true, ela executa a próxima instrução.

Neste caso, você só precisa executar uma instrução se o `Locale` da máquina na qual o código está executando for `Locale.US`. Se for necessário executar mais de uma instrução, será possível usar chaves para formar uma *instrução composta*. Uma instrução composta agrupa muitas instruções em um — e instruções compostas também podem conter outras instruções compostas.

Escopo da variável

Cada variável em um aplicativo Java tem *escopo*, ou namespace localizado, que você pode acessá-lo por nome dentro do código. Fora desse espaço, a variável está *fora do escopo* e você obtém um erro de compilação se tenta acessá-la. Níveis de escopo na linguagem Java são definidos por onde uma variável é declarada, conforme mostrado na Listagem 7.

Escopo da variável

```
public class SomeClass {
  private String someClassVariable;
  public void someMethod(String someParameter) {
    String someLocalVariable = "Hello";

    if (true) {
      String someOtherLocalVariable = "Howdy";
    }
    someClassVariable = someParameter; // legal
    someLocalVariable = someClassVariable; // also legal
    someOtherLocalVariable = someLocalVariable;// Variable out of scope!
  }
  public void someOtherMethod() {
    someLocalVariable = "Hello there";// That variable is out of scope!
  }
```

Mostrar mais

Dentro de `SomeClass`, `someClassVariable` é acessível por todos os métodos de instância (ou seja, não estáticos). Dentro de `someMethod`, `someParameter` é visível, mas fora desse método não é, e o mesmo ocorre para `someLocalVariable`. Dentro do bloco `if`, `someOtherLocalVariable` é declarado e, fora desse bloco `if`, ele está fora do escopo. Por essa razão, dizemos que Java tem *escopo de bloco*, pois blocos (delimitados por { e }) definem os limites do escopo.

O escopo tem muitas regras, mas Escopo da variável mostra aquelas mais comuns. Gaste alguns minutos para se familiarizar com elas.

A instrução `else`

Às vezes, em um fluxo de controle de programa, você deseja efetuar ação apenas se uma determinada expressão falhar ao ser avaliada como true. É quando `else` entre em cena:

```
public int getHeight() {
  int ret;
  if (gender.equals("MALE"))
    ret = height + 2;
  else {
    ret = height;
  Logger.getLogger("Person").info("Being honest about height...");
  }
  return ret;
}
```

Mostrar mais

A instrução `else` funciona da mesma forma que `if`, ou seja, executa apenas a próxima instrução na qual é executada. Neste caso, duas instruções são agrupadas em uma instrução composta (observe as chaves), que o programa então executa.

Também é possível usar `else` para executar uma verificação de `if` adicional:

```
if (conditional) {
  // Block 1
} else if (conditional2) {
  // Block 2
} else if (conditional3) {
  // Block 3
} else {
  // Block 4
} // End
```

Mostrar mais

Se `conditional` for avaliado como true, então `Block 1` será executado e o programa irá para a próxima instrução depois da chave final (que é indicada por `// End`). Se `conditional`não_ for avaliado como true, então `conditional2` será avaliado. Se `conditional2` for true, então `Block 2` será executado e o programa irá para a próxima instrução depois da chave final. Se `conditional2` não for true, o programa irá para `conditional3` e assim por diante. Apenas se todos os três condicionais falharem, `Block 4` será executado.

O operador ternário

A linguagem Java fornece um operador útil para efetuar verificações simples da instrução `if` / `else`. Sua sintaxe é:

```
(conditional) ? statementIfTrue : statementIfFalse;
```

Mostrar mais

Se `conditional` for avaliado como true, então `statementIfTrue` será executado; caso contrário, `statementIfFalse` será executado. Instruções compostas não são permitidas para nenhuma instrução.

O operador ternário será útil quando você souber o que precisa para executar uma instrução como o resultado da avaliação condicional como true e outra caso não saiba. Operadores ternários são usados mais frequentemente para inicializar uma variável (como um valor de retorno), como a seguir:

```
public int getHeight() {
  return (gender.equals("MALE")) ? (height + 2) : height;
}
```

Mostrar mais

Os parênteses após o ponto de interrogação não são estritamente necessários, mas eles tornam o código mais legível.

Loops

Além de ser capaz de aplicar condições para seus programas e ver diferentes resultados com base nos vários cenários `if` / `then`, às vezes você deseja que seu código efetue a mesma ação mais e mais vezes, até que a tarefa seja concluída. Nesta seção, aprenda sobre duas construções usadas para iteração por código ou execute-o mais de uma vez: loops `for` e `while`.

O que é um loop?

Um *loop* é uma construção de programação que executa repetidamente enquanto alguma condição (ou conjunto de condições) é atendida. Por exemplo, você pode solicitar a um programa que leia todos os registros até o final de um arquivo ou pode efetuar loop de todos os elementos em uma matriz, processando cada um. (Você aprende sobre matriz na seção "Coleções Java" deste tutorial.)

`Loops for`

A construção básica de loop na linguagem Java é a instrução `for`, que você pode usar para iterar em um intervalo de valores para determinar quantas vezes executar um loop. A sintaxe abstrata de um loop `for` é:

```
for (initialization; loopWhileTrue; executeAtBottomOfEachLoop) {
  statementsToExecute
}
```

Mostrar mais

No início do loop, a instrução de inicialização é executada (diversas instruções de inicialização podem ser separadas por vírgulas). Desde que `loopWhileTrue` (uma expressão condicional Java que deve ser avaliada como true ou false) seja true, o loop será executado. Na parte inferior do loop, `executeAtBottomOfEachLoop` é executado.

Exemplo de um loop `for`

Se você quiser mudar um método `main()` para executar três vezes, pode usar um loop `for`, conforme mostrado na Listagem 8.

Um loop `for`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  for (int aa = 0; aa < 3; aa++) {
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
  }
}
```

Mostrar mais

A variável local `aa` é inicializada como zero no início da Um loop `for`. Essa instrução executa apenas uma vez, quando o loop é inicializado. O loop continua então três vezes, e sempre que `aa` é incrementado por um.

Como você verá posteriormente, uma sintaxe de loop `for` alternativa está disponível para loop em construções que implementam a interface `Iterable` (como matrizes e outras classes do utilitário Java). Por enquanto, apenas observe o uso da sintaxe de loop `for` na Um loop `for`.

Loops `while`

A sintaxe para um loop `while` é:

```
while (condition) {
  statementsToExecute
}
```

Mostrar mais

Como você pode suspeitar, a condição `while _` é avaliada como true, portanto, o loop é executado. Na parte superior de cada iteração (ou seja, antes da execução de qualquer instrução), a condição é avaliada. Se a condição for avaliada como true, o loop será executado. Portanto, é possível que um loop `while` nunca seja executado se sua expressão condicional não for true pelo menos uma vez.

Consulte novamente o loop `for` na Um loop `for`. Para comparação, Listagem 9 usa um loop `while` para obter o mesmo resultado.

Um loop `while`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
```

```
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    aa++;
  }
,
```

Mostrar mais

Como você pode ver, um loop `while` requer um pouco mais de manutenção que um loop `for`. Você deve inicializar a variável `aa` e também lembrar de incrementá-la na parte inferior do loop.

`Loops do...while`

Se quiser um loop que sempre seja executado uma vez e depois verifique sua expressão condicional, tente usar um loop `do...while`, conforme mostrado na Listagem 10.

Um loop `do...while`

```
int aa = 0;
do {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  l.info("Loop executing iteration# " + aa);
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
  aa++;
} while (aa < 3);
```

Mostrar mais

A expressão condicional (`aa < 3`) não é verificada até o término do loop.

Ramificação de loop

Às vezes, é necessário resgatar um loop antes que a expressão condicional seja avaliada como false. Esta situação pode ocorrer se você estiver procurando uma matriz de `String`s para um determinado valor e, depois de encontrá-la, não se importa com os outros elementos da matriz. Para os momentos nos quais você deseja resgatar, a linguagem Java fornece a instrução `break`, mostrada na Listagem 11.

Uma instrução `break`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    if (aa == 1)
      break;
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

```
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
```

Mostrar mais

A instrução `break` o leva para a próxima instrução executável fora do loop no qual ela está localizada.

Continuação de loop

No exemplo (simplista) na Uma instrução `break`, você só deseja executar o loop apenas uma vez e resgatar. Também é possível ignorar uma única iteração de um loop, mas continuar executando o loop. Para esse propósito, é necessária a instrução `continue`, mostrada na Listagem 12.

Uma instrução `continue`

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    if (aa == 1)
      continue;
    else
    aa++;
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
```

Mostrar mais

Na Uma instrução `continue`, você ignora a segunda iteração de um loop, mas continua na terceira. `continue` será útil quando você estiver, digamos, processando registros e se depara com um registro que definitivamente não deseja processar. É possível ignorar esse registro e ir para o próximo.

Coleções Java

A maioria dos aplicativos reais lidam com coleções de itens como arquivos, variáveis, registros de arquivos ou conjuntos de resultados do banco de dados. A linguagem Java possui uma Estrutura de coleções sofisticada que permite criar e gerenciar coleções de objetos de vários tipos. Esta seção não ensinará sobre Coleções Java, mas apresentará as classes de coleções mais usadas e o introduzirá a elas.

Matrizes

A maioria das linguagens de programação inclui o conceito de uma *matriz* para manter uma coleção de itens e a linguagem Java não é exceção. Uma matriz não é nada mais que uma coleção de elementos do mesmo tipo.

É possível declarar uma matriz de uma das duas formas:

- Crie-a com um determinado tamanho, que é fixo pela vida útil da matriz.
- Crie-a com um determinado conjunto de valores iniciais. O tamanho desse conjunto determina o tamanho da matriz —, se ele é suficientemente largo para manter todos esses valores, e seu tamanho é fixo pela vida útil da matriz.

Observação: os colchetes nestes exemplos de código da seção fazem parte da sintaxe requerida para Coleções Java, *não* são indicadores de elementos opcionais.

Declarando uma matriz

No geral, você declara uma matriz como a seguir:

```
new elementType [arraySize]
```

Mostrar mais

É possível criar uma matriz de número inteiro de elementos de duas formas. Essa instrução cria uma matriz que possui espaço para cinco elementos, mas está vazia:

```
// creates an empty array of 5 elements:
int[] integers = new int[5];
```

Mostrar mais

Esta instrução cria a matriz e a inicializa de uma só vez:

```
// creates an array of 5 elements with values:
int[] integers = new int[] { 1, 2, 3, 4, 5 };
```

Mostrar mais

Os valores iniciais estão entre chaves e são separados por vírgulas.

Outra forma de criar uma matriz é criá-la e depois codificar um loop para iniciá-la:

```
int[] integers = new int[5];
for (int aa = 0; aa < integers.length; aa++) {
  integers[aa] = aa+1;
}
```

Mostrar mais

O código anterior declara uma matriz de número inteiro de cinco elementos. Se você tentar inserir mais que cinco elementos na matriz, o Java Runtime lançará uma *exceção*. Você aprenderá sobre exceções e como lidar com elas na Parte 2.

Carregando uma matriz

Para carregar a matriz, efetue o loop usando números inteiros de 1 ao comprimento da matriz (que você obterá chamando `.length` na matriz — ; mais sobre isso a seguir). Neste caso, você para quando chega a 5.

Depois que a matriz é carregada, você pode acessá-la como antes:

```
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < integers.length; aa++) {
  l.info("This little integer's value is: " + integers[aa]);
}
```

Mostrar mais

Esta sintaxe mais recente (disponível desde o JDK 5) também funciona:

```
Logger l = Logger.getLogger("Test");
for (int i : integers) {
  l.info("This little integer's value is: " + i);
}
```

Mostrar mais

Eu acho a sintaxe mais recente mais fácil de trabalhar e a uso nesta seção.

O índice de elemento

Pense em uma matriz como uma série de depósitos e em cada um deles há um elemento de um determinado tipo. O acesso a cada depósito é obtido usando um *índice*:

```
element = arrayName [elementIndex];
```

Mostrar mais

Para acessar um elemento, é necessário fazer referência à matriz (seu nome) e ao índice no qual o elemento que você deseja reside.

O método `length`

Um método útil, como já foi visto, é `length`. Ele é um método integrado, portanto sua sintaxe não inclui os parênteses usuais. Apenas digite a palavra `length` e ele retornará, — conforme esperado, — o tamanho da matriz.

Matrizes na linguagem Java são baseadas em zero. Portanto, para algumas matrizes denominadas `array`, o primeiro elemento sempre reside em `array[0]` e o último reside em `array[array.length - 1]`.

Uma matriz de objetos

Você viu como as matrizes podem manter tipos primitivos, mas vale mencionar que elas também podem manter objetos. Nesse sentido, a matriz é a coleção mais utilitária da linguagem Java.

Criar uma matriz de objetos `java.lang.Integer` não é muito diferente de criar uma matriz de tipos primitivos. Novamente, você tem duas formas de fazer isso:

```
// creates an empty array of 5 elements:
Integer[] integers = new Integer[5];
```

Mostrar mais

```
// creates an array of 5 elements with values:
Integer[] integers = new Integer[] { Integer.valueOf(1),
Integer.valueOf(2)
Integer.valueOf(3)
Integer.valueOf(4)
Integer.valueOf(5));
```

Mostrar mais

Boxing e unboxing

Cada tipo primitivo na linguagem Java tem uma classe de contraparte JDK, que você pode ver na Tabela 4.

Primitivas e contrapartes JDK

Primitiva	Contraparte JDK
`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`char`	`java.lang.Character`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

Cada classe JDK fornece métodos para analisar e converter de sua representação interna em um tipo de primitiva correspondente. Por exemplo, este código converte o valor decimal 238 em um `Integer`:

```
int value = 238;
Integer boxedValue = Integer.valueOf(value);
```

Mostrar mais

Esta técnica é conhecida como *boxing*, pois você está colocando a primitiva em um wrapper ou caixa.

De modo semelhante, para converter a representação `Integer` novamente a sua contraparte `int`, você *efetua unbox* dela:

```
Integer boxedValue = Integer.valueOf(238);
int intValue = boxedValue.intValue();
```

Mostrar mais

Autoboxing e auto-unboxing

Falando estritamente, não é necessário efetuar box e unbox das primitivas explicitamente. Ao contrário, é possível usar os recursos de autoboxing e auto-unboxing da linguagem Java:

```
int intValue = 238;
Integer boxedValue = intValue;
//
intValue = boxedValue;
```

Mostrar mais

Entretanto, eu recomendo que você evite autoboxing e auto-unboxing, pois isso pode levar a problemas de leitura de código. O código nos fragmentos de boxing e unboxing é mais óbvio e, portanto, mais legível que o código com autoboxing; eu acredito que isso compensa o esforço extra.

Analisando e convertendo tipos com boxing

Você viu como obter um tipo com boxing, mas o que dizer da análise de um `String` que você suspeita ter um tipo com boxing em sua caixa correta? As classes de wrapper JDK possuem métodos para isso também:

```
String characterNumeric = "238";
Integer convertedValue = Integer.parseInt(characterNumeric);
```

Mostrar mais

Também é possível converter o conteúdo de um tipo de wrapper JDK em um `String`:

```
Integer boxedValue = Integer.valueOf(238);
String characterNumeric = boxedValue.toString();
```

Mostrar mais

Observe que ao usar o operador de concatenação em uma expressão `String` (você já viu isso em chamadas de `Logger`), o tipo de primitiva tem autoboxing efetuado e os tipos de wrapper

automaticamente têm `toString()` chamado. Muito útil.

Listas

`List` é uma construção de coleções que é, por definição, uma coleção ordenada, também conhecida como *sequência*. Como `List` é ordenado, você possui controle total sobre o local para onde os itens de `List` irão. Uma coleção Java `List` pode ter apenas objetos e define um contrato estrito sobre seus comportamentos.

`List` é uma interface, portanto, não é possível instanciá-la diretamente. Você trabalhará com sua implementação mais comumente usada, `ArrayList`. Há duas formas de fazer a declaração. Primeiro, usando a sintaxe explícita:

```
List<String> listOfStrings = new ArrayList<String>();
```

Mostrar mais

Segundo, usando o operador “diamond”, apresentado no JDK 7:

```
List<String> listOfStrings = new ArrayList<>();
```

Mostrar mais

Observe que o tipo de objeto na instanciação `ArrayList` não é especificado. Isso ocorre porque o tipo de classe à direita da expressão deve corresponder ao tipo do lado esquerdo. No restante deste tutorial, eu uso os dois tipos, pois você provavelmente verá os dois usos na prática.

Observe que eu designei o objeto `ArrayList` para uma variável do tipo `List`. Com programação Java, é possível designar uma variável de um tipo para outro, desde que a variável que está sendo designada seja uma superclasse ou interface implementada pela variável da qual ela está sendo designada. É possível saber mais sobre como as designações de variáveis são afetadas na Parte 2 na seção “Herança“.

Tipo formal

O `<Object>` no fragmento de código precedente é chamado de *tipo formal*. `<Object>` informa ao compilador que `List` contém uma coleção do tipo `Object`, o que significa que você pode inserir o que desejar em `List`.

Se você quiser deixar as restrições mais rigorosas sobre o que é possível ou não inserir em `List`, poderá definir o tipo formal de forma diferente:

```
List<Person> listOfPersons = new ArrayList<Person>();
```

Mostrar mais

Agora `List` poderá conter apenas instâncias `Person`.

Usando `List`s

Usar `List`s é super fácil, como as coleções Java no geral. A seguir, algumas das tarefas que se pode fazer com `List` s:

- Inserir algo em `List`.
- Perguntar a `List` qual seu tamanho no momento.
- Retirar algo de `List`.

Agora, você pode tentar algumas dessas tarefas. Você já viu como criar uma instância de `List` instanciando seu tipo de implementação `ArrayList`, portanto, pode começar daqui.

Para inserir algo em `List`, chame o método `add()`:

```
List<Integer> listOfIntegers = new ArrayList<>();
listOfIntegers.add(Integer.valueOf(238));
```

Mostrar mais

O método `add()` inclui o elemento ao final de `List`.

Para perguntar a `List` seu tamanho, chame `size()`:

```
List<Integer> listOfIntegers = new ArrayList<>();

listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Current List size: " + listOfIntegers.size());
```

Mostrar mais

Para recuperar um item de `List`, chame `get()` e o transmita ao índice do item que deseja:

```
List<Integer> listOfIntegers = new ArrayList<>();
listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Item at index 0 is: " listOfIntegers.get(0));
```

Mostrar mais

Em um aplicativo real, `List` conteria registros ou objetos de negócios, e você possivelmente desejaria consultá-los como parte de seu processamento. Como fazer isso de uma forma genérica? Você deseja fazer a iteração da coleção, o que pode ser feito porque `List` implementa a interface `java.lang.Iterable`. (Você aprendeu sobre interfaces na Parte 2.)

`Iterable`

Se uma coleção implementar `java.lang.Iterable`, ela será chamada de *coleção com possível iteração*. É possível iniciar em uma extremidade e percorrer a coleção item por item até que os itens se esgotem.

Você já viu a sintaxe especial para iteração de coleções que implementam a interface `Iterable`, na seção "Loops". Novamente aqui:

```
for (objectType varName : collectionReference) {
  // Start using objectType (via varName) right away...
}
```

Mostrar mais

Iterando em `List`

Esse exemplo anterior era abstrato; agora, há um exemplo mais realista:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (Integer i : listOfIntegers) {
  l.info("Integer value is : " + i);
}
```

Mostrar mais

Este pequeno fragmento de código efetua a mesma ação do fragmento maior:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < listOfIntegers.size(); aa++) {
  Integer I = listOfIntegers.get(aa);
  l.info("Integer value is : " + i);
}
```

Mostrar mais

O primeiro fragmento usa sintaxe abreviada: não há nenhuma variável `index` (`aa` neste caso) para inicializar e nenhuma chamada para `get()` de `List`.

Como `List` estende `java.util.Collection`, que implementa `Iterable`, é possível usar a sintaxe abreviada para iteração em qualquer `List`.

Sets

Um `Set` é uma construção de coleções que, por definição, contém elementos exclusivos — ou seja, nenhuma duplicata. Considerando que `List` pode conter o mesmo objeto centenas de vezes, um `Set` só pode conter uma determinada instância uma vez. Uma coleção Java `Set` pode ter apenas objetos e define um contrato estrito sobre seus comportamentos.

Como `Set` é uma interface, ela não pode ser instanciada diretamente, portanto, aqui está uma das minhas implementações favoritas: `HashSet`. `HashSet` é fácil de usar e é semelhante a `List`.

A seguir, algumas das tarefas que se pode fazer com `Set`:

- Inserir algo em `Set`.
- Perguntar a `Set` qual seu tamanho no momento.
- Retirar algo de `Set`.

Usando `Set`s

Um atributo de distinção de `Set` é o que garante exclusividade entre seus elementos, mas a ordem dos elementos não é importante. Considere o seguinte código:

```
Set<Integer> setOfIntegers = new HashSet<Integer>();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(11));
setOfIntegers.add(Integer.valueOf(10));
for (Integer i : setOfIntegers) {
  l.info("Integer value is: " + i);
}
```

Mostrar mais

É possível esperar que `Set` tenha três elementos, mas ele só possui dois porque o objeto `Integer` que contém o valor `10` é incluído apenas uma vez.

Mantenha este comportamento em mente ao fazer iteração com `Set`, como a seguir:

```
Set<Integer> setOfIntegers = new HashSet();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(20));
setOfIntegers.add(Integer.valueOf(30));
setOfIntegers.add(Integer.valueOf(40));
setOfIntegers.add(Integer.valueOf(50));
Logger l = Logger.getLogger("Test");
for (Integer i : setOfIntegers) {
  l.info("Integer value is : " + i);
}
```

Mostrar mais

Os objetos impressos em uma ordem diferente da ordem que você os incluiu são casuais, pois `Set` garante exclusividade, não ordem. É possível ver isso ao colar o código anterior no método `main()` de sua classe `Person` e executá-lo.

Maps

Um `Map` é uma construção de coleção útil que você pode usar para associar um objeto (a *chave*) a outro (o *valor*). Como você pode imaginar, a chave para `Map` deve ser exclusiva e ela é usada para recuperar o valor posteriormente. Uma coleção Java `Map` pode ter apenas objetos e define um contrato estrito sobre seus comportamentos.

Como `Map` é uma interface, ela não pode ser instanciada diretamente, portanto, aqui está uma das minhas implementações favoritas: `HashMap`.

A seguir, algumas das tarefas que se pode fazer com `Map` s:

- Inserir algo em `Map`.
- Retirar algo de `Map`.
- Obter um `Set` de chaves para `Map` —, para iteração nele.

Usando `Map`s

Para inserir algo em `Map`, é necessário ter um objeto que represente sua chave e um objeto que represente seu valor:

```
public Map<String, Integer> createMapOfIntegers() {
  Map<String, Integer> mapOfIntegers = new HashMap<>();
  mapOfIntegers.put("1", Integer.valueOf(1));
  mapOfIntegers.put("2", Integer.valueOf(2));
  mapOfIntegers.put("3", Integer.valueOf(3));
  //...
  mapOfIntegers.put("168", Integer.valueOf(168));
}
```

Mostrar mais

Neste exemplo, `Map` contém `Integer` s, encadeados por `String`, que pode ser suas representações de `String`. Para recuperar um valor `Integer` específico, é necessário sua representação de `String`:

```
mapOfIntegers = createMapOfIntegers();
Integer oneHundred68 = mapOfIntegers.get("168");
```

Mostrar mais

Usando `Set` com `Map`

No momento, é possível que você tenha uma referência a `Map` e queira percorrer todo seu conjunto de conteúdo. Nesse caso, será necessário um `Set` de chaves para `Map`:

```
Set<String> keys = mapOfIntegers.keySet();
Logger l = Logger.getLogger("Test");
for (String key : keys) {
  Integer  value = mapOfIntegers.get(key);
  l.info("Value keyed by '" + key + "' is '" + value + "'");
}
```

Mostrar mais

Observe que o método `toString()` de `Integer` recuperado de `Map` é automaticamente chamado quando usado na chamada `Logger`. `Map` não retorna `List` de suas chaves, pois `Map` está encadeado e cada chave é exclusiva. A exclusividade é a característica de distinção de um `Set`.

Fazendo archive do código Java

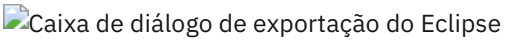
Agora que você aprendeu um pouco sobre como escrever aplicativos Java, pode desejar saber como empacotá-los para que outros desenvolvedores possam usá-los ou como importar o código de outro desenvolvedor em seus aplicativos. Esta seção mostra como fazer isso.

JARs

O JDK é fornecido com uma ferramenta chamada JAR, que é responsável pelo Java archive. Você usa esta ferramenta para criar arquivos JAR. Depois de empacotar seu código em um arquivo JAR, outros desenvolvedores podem descartar o arquivo JAR em seus projetos e configurar seus projetos para usar seu código.

Criar um arquivo JAR em Eclipse é fácil. Em sua área de trabalho, clique com o botão direito no pacote `com.makotojava.intro` e clique em **Arquivo > Exportar**. Você verá a caixa de diálogo mostrada na Figura 10. Escolha **Java > Arquivo JAR** e clique em **Avançar**.

Exportar caixa de diálogo



Quando a próxima caixa de diálogo for aberta, navegue até o local no qual deseja armazenar seu arquivo JAR e nomeie o arquivo da forma que desejar. A extensão .jar é o padrão, que eu recomendo usar. Clique em **Concluir**.

Você verá o arquivo JAR no local selecionado. É possível usar as classes nele a partir de seu código se você inserir o JAR em seu caminho de construção em Eclipse. Fazer isso também é fácil, como verá a seguir.

Usando aplicativos de terceiros

À medida que você se sente mais confortável ao escrever aplicativos Java, pode desejar usar mais e mais aplicativos de terceiros para suportar seu código. Embora o JDK seja muito bom, ele não fornece tudo o que você precisa para escrever um grande código Java. A comunidade de software livre Java fornece muitas bibliotecas para ajudá-lo a eliminar essas lacunas. A título de exemplo, suponha que você deseje usar Commons Lang, uma biblioteca de substituição de JDK para manipular as classes principais Java. As classes fornecidas por Commons Lang o ajudam a manipular matrizes, criar números aleatórios e executar manipulação de sequência.

Assumiremos que você já tenha efetuado o download de Commons Lang, que está armazenada em um arquivo JAR. Para usar as classes, a primeira etapa é criar um diretório lib em seu projeto e eliminar o arquivo JAR:

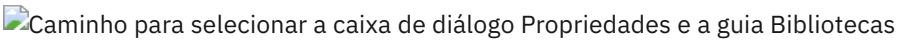
1. Clique com o botão direito na pasta raiz Intro na visualização Explorador de Projetos.
2. Clique em **Nova > Pasta** e chame a pasta `lib`.
3. Clique em **Concluir**.

A nova pasta é mostrada no mesmo nível de src. Agora, copie o arquivo Commons Lang JAR em seu novo diretório lib. Para este exemplo, o arquivo é chamado de commons-lang3.3.4.jar. (Isso é comum ao nomear um arquivo JAR para incluir o número de versão, neste caso 3.4.)

Agora, tudo o que você precisa fazer é solicitar ao Eclipse que inclua as classes no arquivo commons-lang3.3.4.jar em seu projeto:

1. Clique com o botão direito no projeto Intro em sua área de trabalho; em seguida, clique em **Propriedades**.
2. Na caixa de diálogo Propriedades, clique na guia Bibliotecas, conforme mostrado na Figura 11:

Propriedades > Caminho de construção Java



1. Clique no botão **Incluir JARs externos**. Navegue até o diretório lib do projeto, clique no arquivo commons-lang3.3.4.jar e clique em **OK**.

Depois que o código (ou seja, os arquivos de classe) no arquivo JAR são processados pelo Eclipse, eles são disponibilizados para referência (importação) de seu código Java. Observe no Explorador de Projetos que existe uma nova pasta, chamada Bibliotecas referenciadas, que contém o arquivo commons-lang3.3.4.jar.

Escrevendo um bom código Java

Você já obteve conhecimento suficiente na sintaxe Java para escrever programas Java básicos, o que significa que a primeira metade deste tutorial está prestes a concluir. Esta seção final expõe algumas das melhores práticas que podem ajudá-lo a escrever um código Java mais claro, com mais possibilidade de manutenção.

Mantenha classes pequenas

Você criou algumas classes neste tutorial. Depois de gerar pares getter/setter até para o pequeno número de atributos (pelos padrões de uma classe Java real), a classe `Person` possuirá 150 linhas de código. Esta é uma classe pequena. Não é incomum ver classes com 50 ou 100 métodos e milhares de linhas de origem (ou mais). O principal ponto para métodos é manter apenas aquilo que é necessário. Se você precisar de diversos métodos auxiliares que fazem essencialmente a mesma coisa, mas obtêm parâmetros diferentes (como o método `printAudit()`), essa é uma boa escolha. Apenas certifique-se de limitar a lista de métodos ao que você precisa, e não mais.

No geral, as classes representam alguma entidade conceitual em seu aplicativo e seus tamanhos devem refletir apenas a funcionalidade para fazer o que essa entidade necessita. Elas devem permanecer amplamente focadas em efetuar um pequeno número de tarefas e fazê-las bem.

Nomeie métodos cuidadosamente

Um bom padrão de codificação quando se trata de nomes de métodos é o padrão de nomes de métodos *revelação de intenção*. Esse padrão é mais fácil de entender com um simples exemplo. Qual dos nomes de métodos a seguir é mais fácil de decifrar numa visualização rápida?

- `a()`

- `computeInterest()`

A resposta deveria ser óbvia, ainda que, por algum motivo, programadores tenham uma tendência a fornecer nomes de métodos (e variáveis, neste caso) pequenos e abreviados. Certamente, um nome ridiculamente longo pode ser inconveniente, mas um nome que transmita o que um método faz não precisa ser ridiculamente longo. Seis meses após escrever alguns códigos, talvez você não se lembre o que quis fazer com um método chamado `compInt()`, mas é óbvio que um método chamado `computeInterest()`, provavelmente chamará sua atenção.

Mantenha métodos pequenos

Métodos pequenos são tão preferíveis quanto classes pequenas, e por motivos semelhantes. Um padrão que eu tento seguir é manter o tamanho de um método para *uma página*, pois eu o vejo em minha tela. Isso torna minhas classes de aplicativo mais fáceis de manter.

Se um método passar de uma página, eu o *refatoro*. Refatorar é mudar o design do código existente sem mudar seus resultados. O Eclipse tem um conjunto maravilhoso de ferramentas de refatoração. Geralmente, um método longo contém subgrupos de funcionalidades agrupadas. Pegue essa funcionalidade e mova-a para outro método (nomeando-a de acordo) e a transmita nos parâmetros, conforme necessário.

Limite cada método a uma única tarefa. Eu acredito que um método que faça apenas uma tarefa bem feita geralmente não possui mais do que cerca de 30 linhas de código.

Use comentários

Use comentários. As pessoas que o usarão depois (ou até mesmo você, seis meses depois) agradecerão. Talvez você conheça aquele velho ditado *Um código bem escrito é, por si só, uma documentação; então quem precisa de comentários?* Eu darei duas razões para mostrar porque esse ditado é falso:

- A maioria dos códigos não é bem escrita.
- Provavelmente seu código não estará tão bem escrito como você pensa estar.

Portanto, comente seu código. Ponto.

Use um estilo consistente

O estilo de codificação é uma questão de preferência pessoal, mas eu o aconselho a usar a sintaxe Java de chaves:

```
public static void main(String[] args) {
}
```

Mostrar mais

Não use este estilo:

```
public static void main(String[] args)
{
}
```

Mostrar mais

Ou este:

```
public static void main(String[] args)
  {
  }
```

Mostrar mais

Por quê? Bem, isso é padrão, portanto a maioria dos códigos com os quais você se depara (como em código que não escreveu, mas pode ser pago para manter) provavelmente será escrita dessa forma. Dito isso, o Eclipse *permite* definir estilos de códigos e formatar seu código do modo que desejar. A principal questão é que você escolhe um estilo e adere a ele.

Use criação de log integrada

Antes que Java 1.4 apresentasse a criação de log integrada, a forma canônica de descobrir o que seu programa estava fazendo era fazer uma chamada do sistema, como a seguir:

```
public void someMethod() {
  // Do some stuff...
  // Now tell all about it
  System.out.println("Telling you all about it:");
  // Etc...
}
```

Mostrar mais

O recurso de criação de log integrado da linguagem Java (retorne à consulta a "Sua primeira classe Java") é uma melhor alternativa. Eu *nunca* uso `System.out.println()` em meu código e sugiro que você não o use também. Outra alternativa é a biblioteca de substituição log4j comumente usada, parte do projeto Apache umbrella.

Nas pegadas de Fowler

O melhor livro no mercado (em minha opinião, e não estou sozinho) é *Refactoring: Improving the Design of Existing Code* de Martin Fowler et al. Esse livro é até divertido de ler. Os autores falam sobre "code smells" que "imploram" por refatoração e detalham as diversas técnicas para corrigi-los.

A refatoração e a capacidade de escrever código test-first são as aptidões mais importantes para aprendizado para novos programadores. Se todos fossem bons nas duas, a indústria sofreria uma revolução. Se você se torna bom nas duas, finalmente produz código mais claro e aplicativos mais funcionais do que muitos de seus peers.

Conclusão para a parte 1

Neste tutorial, você aprendeu sobre a programação orientada a objeto, descobriu a sintaxe Java que pode usar para criar objetos úteis e se familiarizou com um IDE que o ajuda a controlar seu ambiente de desenvolvimento. Você sabe como criar e executar objetos Java que podem fazer diversas tarefas, incluindo fazer coisas diferentes com base em entradas diferentes. Você também sabe como efetuar JAR de seus aplicativos para que outros desenvolvedores usem em seus programas e obteve algumas das melhores práticas básicas de programação Java.

O que vem a seguir

Na segunda metade deste tutorial, você começa aprendendo sobre algumas das construções de programação Java mais avançadas, embora a discussão geral ainda esteja no escopo introdutório. Os tópicos da programação Java abrangidos naquele tutorial incluem:

- Manipulação de exceções
- Herança e abstração
- Interfaces
- Classes aninhadas
- Expressões regulares
- Genéricos
- Tipos de enumeração
- E/S
- Serialização

Leia "*Introdução à programação Java*, Parte 2: Construções para aplicativos reais."

Aviso