CAPÍTULO 2

Classes e Objetos

As primeiras coisas primeiro, mas não necessariamente nesta ordem.
– Dr. Who, Meglos

A unidade fundamental de programação da linguagem de programação Java é a classe. Classes fornecem a estrutura para os *objetos* e os mecanismos para fabricar objetos a partir de uma definição de classe. Classes definem *métodos*: coleções de código executável que são o foco da computação e que manipulam os dados armazenados em objetos. Métodos fornecem o comportamento dos objetos de uma classe. Embora você possa fazer computações usando somente tipos primitivos – inteiros, ponto-flutuante e assim por diante – quase todos os programas interessantes irão criar e manipular objetos.

A programação orientada a objetos separa claramente a noção de *o que* é feito de *como* é feito. ("O que" é descrito como um conjunto de métodos (e às vezes com dados publicamente disponíveis) e suas semânticas associadas. Esta combinação – métodos, dados e semântica – é muitas vezes descrita como um *contrato* entre o projetista da classe e o programador que a usa, porque ela diz o que acontece quando certos métodos são invocados sobre um objeto. Este contrato define um *tipo* tal que todos os objetos que são instâncias deste tipo são conhecidos por honrar este contrato.

Uma suposição comum é que os métodos declarados em uma classe constituem seu contrato completo. A semântica destas operações também faz parte do contrato, mesmo que ela tenha sido descrita somente na documentação. Dois métodos podem ter o mesmo nome e parâmetros, lançar as mesmas exceções, mas eles não são equivalentes se eles têm semânticas diferentes. Por exemplo, nem todo o método denominado `print` pode ser suposto imprimir uma cópia do objeto. Alguém pode definir um método `print` com a semântica "processar intervalo" ou "priorizar não-terminais" – não que recomendemos isto. O contrato de método, assinatura e semântica juntas, define o que ele significa.

O "como" de um objeto é definido pela sua classe, que define a implementação dos métodos que o objeto suporta. Cada objeto é uma *instância* de uma classe. Quando um método é invocado sobre um objeto, a classe é examinada para encontrar o código a ser executado. Um objeto pode usar outros objetos para realizar o seu trabalho, mas vamos iniciar com classes simples que implementam seus próprios métodos diretamente.

2.1 Uma Classe Simples

Aqui está uma classe simples, chamada `Body`[1] que poderia ser usada para armazenar dados sobre corpos celestes tais como cometas, asteróides, planetas e estrelas:

```
class Body {
   public long idNum;
   public String name;
   public Body orbits;

   public static long nextID = 0;
}
```

Uma classe é declarada usando a palavra-chave `class`, dando um nome à classe e listando os membros da classe entre chaves. Uma declaração de classe cria um nome de tipo, de modo que referências a objetos deste tipo podem ser declaradas com um simples

```
Body mercury;
```

Esta declaração afirma que `mercury` é uma variável que pode guardar uma referência para um objeto do tipo `Body`. A declaração não cria um objeto – ela apenas declara uma referência que pode referir-se a um objeto `Body`. Estes objetos devem ser explicitamente criados. A esse respeito, a linguagem de programação Java é diferente das linguagens nas quais objetos são criados quando você declara variáveis.

Esta primeira versão de `Body` é mal projetada. Isto é intencional: vamos demonstrar o valor de certas características da linguagem à medida que melhoramos esta classe neste capítulo.

2.1.1 Membros de Classe

Uma classe pode ter três espécies de membros:

- *Campos* são as variáveis de dados associadas com uma classe e seus objetos e armazenam o estado de uma classe ou objeto.
- *Métodos* contêm o código executável de uma classe e definem o comportamento de objetos.
- *Classes aninhadas* e *interfaces aninhadas* são declarações de classes ou interfaces que ocorrem aninhadas dentro da declaração de outra classe ou interface.

Neste capítulo vamos nos concentrar nos membros básicos: campos e métodos. Membros aninhados são discutidos no Capítulo 5.

2.1.2 Modificadores de Classes

Uma declaração de classe pode ser precedida por modificadores que dão certas propriedades à classe:

[1] Boa denominação é uma parte importante de projeto de classe, e Body poderia na realidade se chamar `CelestialBody` ou algo deste tipo. Usamos `Body` por brevidade, visto que nos referimos a ela dúzias de vezes ao longo deste livro.

- *annotations* – anotações e tipos de anotação são discutidos no Capítulo 15.
- *public* – uma classe `public` é publicamente acessível: qualquer um pode declarar referências e objetos da classe ou acessar seus membros `public`. Sem um modificador, uma classe é somente acessível dentro de seu próprio pacote. Você vai aprender sobre controle de acesso geral na Seção 2.3 na página 68. Pacotes e questões relacionadas com acessibilidade de classes e membros são discutidos no Capítulo 18.
- *abstract* –uma classe `abstract` é considerada incompleta e nenhuma instância da classe pode ser criada. Geralmente isto se deve ao fato da classe conter métodos abstratos que devem ser implementados por uma subclasse. Você vai aprender sobre isto em "Classes e Métodos Abstratos" nas páginas 110-111.
- *final* – Uma classe `final` não pode ter subclasses. Subclasses são discutidas no Capítulo 3.
- *ponto-flutuante estrito* – uma classe declarada `strictfp` possui toda aritmética de ponto-flutuante avaliada estritamente. Ver "Aritmética de Ponto-flutuante Estrita e Não-estrita" nas páginas 222-223 para detalhes.

Uma classe não pode ser ao mesmo tempo `final` e `abstract`.
Uma declaração de classe pode ser precedida por vários modificadores. Modificadores são permitidos em qualquer ordem, mas recomendamos a você que adote uma ordem consistente para melhorar a legibilidade de seu código. Sempre usamos e recomendamos a ordem listada.
Embora não estejamos preocupados com modificadores de classe neste capítulo, você precisa conhecer um pouco sobre classes `public`. A maioria das ferramentas de desenvolvimento Java exige que uma classe `public` seja declarada em um arquivo com o mesmo nome da classe, o que significa que pode existir uma única classe `public` declarada em um arquivo.

Exercício 2.1: Escreva uma classe simples `Veiculo` que possui campos para (pelo menos) velocidade atual, direção atual em graus e nome do proprietário.

Exercício 2.2: Escreva uma classe `ListaEncadeada` que possui um campo tipo `Object` e uma referência para o próximo elemento `ListaEncadeada` da lista.

2.2 Campos

As variáveis de uma classe são chamadas campos; as variáveis `name` e `orbits` da classe `Body` são exemplos. Uma declaração de campo consiste de um nome de tipo seguido pelo nome do campo e opcionalmente uma *cláusula de inicialização* para fornecer ao campo um valor inicial. Cada objeto `Body` possui suas próprias instâncias específicas dos três campos: um `long` que univocamente distingue o corpo celeste de todos os outros, um `String` que é o seu nome e uma referência a outro `Body` ao redor do qual ele orbita. Dar a cada objeto separado uma instância diferente dos campos significa que cada objeto possui seu próprio e único estado – tais campos são conhecidos como variáveis de instância. Alterar o campo `orbits` em um objeto `Body` não afeta o campo `orbits` de qualquer outro objeto `Body`.

Declarações de campos também podem ser precedidas por modificadores que controlam certas propriedades do campo:

- *annotations* – anotações e tipos de anotação são discutidos no Capítulo 15.
- *modificadores de acesso* – estes são discutidos na Seção 2.3 na página 68.
- *static* – isto é discutido abaixo.
- *final* – isto é discutido abaixo.
- *transient* – isto se relaciona à serialização de objetos e é discutida na Seção 20.8 nas páginas 492-493.
- *volatile* – isto se relaciona à sincronização e questões de modelo de memória e é discutido na Seção 14.10 nas páginas 341-342.

Um campo não pode ser ao mesmo tempo `final` e `volatile`.
Quando diversos modificadores são aplicados à mesma declaração de campo, recomendamos usar a ordem listada acima.

2.2.1 Inicialização de campos

Quando um campo é declarado, ele pode ser inicializado pela atribuição a ele de um valor do tipo correspondente. No exemplo `Body`, o campo `nextID` é inicializado com valor zero. A expressão de inicialização, ou mais simplesmente, o *inicializador*, não precisa, no entanto ser uma constante. Ele pode ser um outro campo, uma invocação de método ou uma expressão envolvendo tudo isso. A única exigência é que o inicializador seja do tipo correto e, se ele invoca um método, nenhuma exceção verificada possa ser lançada porque não existe código adjacente para capturar a exceção. Por exemplo, são válidos os seguintes inicializadores:

```
double zero = 0.0;              // constante
double sum = 4.5 + 3.7;         // expressão constante
double zeroCopy = zero;         // campo
double rootTwo = Math.sqrt(2);     // invocação de método
double someVal = sum + 2*Math.sqrt(rootTwo); // misturado
```

Embora inicializadores forneçam uma grande flexibilidade na maneira como os campos são inicializados, eles somente são adequados para esquemas de inicialização simples. Para esquemas mais complexos necessitamos ferramentas adicionais – como logo veremos.
Se um campo não for inicializado, um valor inicial padrão é atribuído a ele, dependendo de seu tipo:

Tipo	Valor Inicial
`boolean`	`false`
`char`	`'\u0000'`
`byte, short, int, long`	`0`
`float, double`	`+0.0`
referência a objeto	`null`

2.2.2 Campos Estáticos

Algumas vezes você deseja que somente uma instância de um campo seja compartilhada por todos os objetos de uma classe.

Você cria tais campos declarando-os `static`, de modo que eles são chamados de *campos estáticos* ou *variáveis de classe*. Quando você declara um campo estático em uma classe, existe somente uma cópia do campo, sem importar quantas instâncias da classe são criadas.

Em nosso caso, `Body` possui um campo `static`, `nextID`, que contém o próximo identificador de campo celeste a ser usado. O campo `nextID` é inicializado com zero quando a classe é inicializada, após ter sido carregada (ver "Carga de Classes" nas páginas 396-397). Você vai ver que a cada objeto `Body` recém criado será atribuído o valor de `nextID` como seu identificador, e o valor de `nextID` será incrementado. Portanto, queremos somente uma cópia do campo `nextID`, para ser usado ao criar todas as instâncias de `Body`.

Dentro de sua própria classe um campo `static` pode ser referido diretamente, mas quando acessado externamente ele deve geralmente ser acessado usando o nome da classe. Por exemplo, podemos imprimir o valor de `nextID` como segue:

```
System.out.println(Body.nextID);
```

O uso de `System.out` ilustra o acesso a um campo `static`. Quando um membro estático de uma classe é referenciado por outra classe muitas vezes, ou quando diversos membros estáticos de uma classe são referenciados por outra classe, você pode tornar seu código mais claro e talvez economizar digitação, usando um comando *static import* para identificar a classe do membro `static` uma única vez. Isto é discutido com detalhes na Seção 2.9 nas páginas 87-88.

Um membro `static` também pode ser acessado usando uma referência a um objeto desta classe, como em

```
System.out.println(mercury.nextID);
```

Você deve evitar esta forma porque ela dá a falsa impressão que `nextID` é um membro do objeto `mercury`, não um membro da classe `Body`. É o tipo de referência e não o tipo de objeto ao qual se refere, que determina a classe onde a variável `static` será procurada.

Neste livro quando usamos o termo campo, geralmente significa a espécie não-estática. Quando o contexto torna isto ambíguo, usamos o termo não-estático para fins de clareza.

Exercício 2.3: Adicione um campo estático à sua classe `Veiculo` para guardar o número de identificação do próximo veículo, e um campo não-estático para guardar o número de identificação de cada carro.

2.2.3 Campos `final`

Uma variável `final` é aquela cujo valor não pode ser alterado após ter sido inicializado – qualquer tentativa de atribuição a este campo produzirá um erro de compilação. Vimos campos `final` usados para definir constantes denominadas porque constantes não mudam de valor. Em geral, um campo `final` é usado para definir uma propriedade imutável de uma classe ou objeto – uma propriedade que não se altera durante o tempo de vida da classe ou objeto. Campos que são marcados como `final` também possuem uma semântica especial em relação ao acesso concorrente por diversas *threads*; isto é discutido com maiores detalhes no Capítulo 14.

Se um campo `final` não possui um inicializador, ele é denominado de *branco final*. Você usaria um branco final quando uma inicialização simples não é apropriada para um campo. Tais campos devem ser inicializados apenas uma vez quando a classe estiver sendo inicializada (no caso de campos `static final`) ou logo que um objeto da classe tenha sido completamente construído (para campos `final` não-estáticos). O compilador vai assegurar que isto seja feito e recusar compilar uma classe se ele verificar que um campo `final` não foi inicializado.

Um campo `final` de tipo primitivo, ou tipo `String`, que é inicializado com uma expressão constante – isto é, uma expressão cujo valor pode ser determinado durante a compilação – é conhecido como *variável constante*. Variáveis constantes são especiais porque o compilador as trata como valores e não como campos. Se o seu código se refere a uma variável constante, o compilador não gera *bytecodes* para carregar o valor do campo a partir do objeto; ele simplesmente insere o valor diretamente no *bytecode*. Isto pode ser uma otimização útil, mas ela significa que se o valor de seu campo `final` necessita ser alterado, então cada parte do código que se refere a esse campo também deve ser recompilada.

O fato de uma propriedade ser imutável é determinado pela semântica da aplicação para a qual a classe foi projetada. Quando você decide se um campo deve ser `final`, considere três coisas:

- O campo representa uma propriedade imutável do objeto?
- O valor do campo é sempre conhecido no momento em que o objeto é criado?
- É sempre prático e apropriado fixar o valor do campo quando o objeto é criado?

Se a propriedade é simplesmente alterada com pouca freqüência em vez de ser verdadeiramente imutável, então um campo `final` não é apropriado. Se o valor do campo não é conhecido quando o objeto é criado então ele não pode ser tornado `final`, mesmo que ele seja conhecido como logicamente imutável. Por exemplo, em uma simulação de corpos celestiais, um cometa pode ser atraído pela força gravitacional de uma estrela e começar a orbitar essa estrela. Uma vez atraído, o cometa orbita a estrela para sempre ou até que seja destruído. Nesta situação, o campo `orbits` vai guardar um valor imutável uma vez estabelecido, mas este valor não é conhecido quando o objeto cometa é criado e portanto este campo não pode ser `final`. Finalmente, se a inicialização do campo é custosa e o valor do campo é necessário com pouca freqüência, então pode ser mais prático postergar a inicialização do campo até seu valor ser necessário – isto é geralmente denominado de *inicialização tardia* – o que não pode ser feito com um campo `final`.

Existem outras considerações a respeito de campos `final` se o seu objeto deve ser clonável ou serializável. Isto é discutido em "Clonar Objetos" na página 114 e "Serialização de Objetos" nas páginas 492-493, respectivamente.

Exercício 2.4: Considere a sua solução para o Exercício 2.3. Você acha que o campo do número de identificação deve ser final?