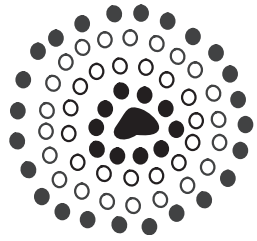
Capítulo

1 Cartilha de Java



Sumário

1.1 Iniciando: classes, tipos e objetos

Construir estruturas de dados e algoritmos requer a comunicação de instruções detalhadas para um computador. Uma excelente maneira de fazer isso é usar uma linguagem de programação de alto nível como Java. Este capítulo apresenta uma visão geral da linguagem Java, assumindo que o leitor esteja familiarizado com alguma linguagem de programação de alto nível. Este livro, entretanto, não fornece uma descrição completa da linguagem Java. Existem aspectos importantes da linguagem que não são relevantes para o projeto de estruturas de dados e que não são incluídos aqui. Inicia-se com um programa que imprime "Hello Universe!" na tela, e que é mostrado e dissecado na Figura 1.1.

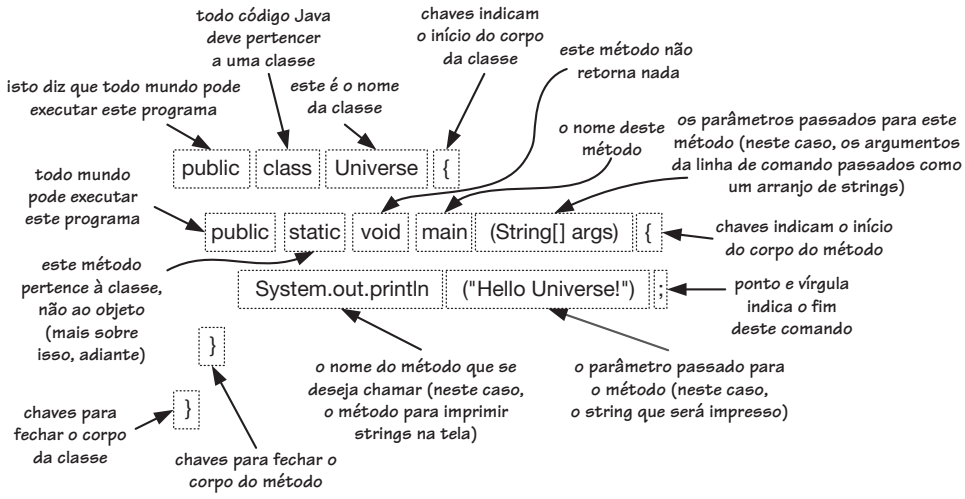


Figura 1.1 O programa "Hello Universe!".

Os principais "atores" em um programa Java são os ***objetos***. Os objetos armazenam dados e fornecem os métodos para acessar e modificar esses dados. Todo objeto é instância de uma ***classe*** que define o ***tipo*** do objeto, bem como os tipos de operações que executa. Os ***membros*** críticos de uma classe Java são os seguintes (classes também podem conter definições de classes aninhadas, mas essa é uma discussão para mais tarde):

- Dados de objetos Java são armazenados em ***variáveis de instância*** (também chamadas de ***campos***). Por essa razão, se um objeto de uma classe deve armazenar dados, então sua classe deve especificar variáveis de instância para esse fim. As variáveis de instância podem ser de tipos básicos (tais como inteiros, números de ponto flutuante ou booleanos) ou podem se referir a objetos de outras classes.
- As operações que podem atuar sobre os dados e que expressam as "mensagens" às quais os objetos respondem são chamadas de ***métodos***, e estes consis-

tem em construtores, subprogramas e funções. Eles definem o comportamento dos objetos daquela classe.

Como as classes são declaradas

Resumindo, um ***objeto*** é uma combinação específica de dados e dos métodos capazes de processar e comunicar esses dados. As classes definem os ***tipos*** dos objetos; por isso, objetos também são chamados de instâncias da classe que os define e usam o nome da classe como seu tipo.

Um exemplo de definição de uma classe Java é apresentado no Trecho de Código 1.1.

```
public class Counter {
   protected int count; // uma simples variável de instância inteira
   /** O construtor default para um objeto Counter */
   Counter( ) { count = 0; }
   /** Um método de acesso para recuperar o valor corrente do contador */
   public int getCount( ) { return count; }
   /** Um método modificador para incrementar o contador */
   public void incrementCount( ) { count++; }
   /** Um método modificador para decrementar o contador */
   public void decrementCount( ) { count--; }
}
```

Trecho de Código 1.1 A classe Counter para um contador simples, que pode ser acessado, incrementado e decrementado.

Nesse exemplo, observa-se que a definição da classe está delimitada por chaves, isto é, começa por um "{" e termina com um "}". Em Java, qualquer conjunto de comandos entre chaves "{" e "}" define um ***bloco*** de programa.

Assim como a classe Universe, a classe Counter é pública, o que significa que qualquer outra classe pode criar e usar um objeto Counter. O Counter tem uma variável de instância – um inteiro chamado de count. Essa variável é inicializada com zero no método construtor, Counter, que é chamado quando se deseja criar um novo objeto Counter (esse método sempre tem o mesmo nome que a classe a qual pertence). Essa classe também tem um método de acesso, getCount, que retorna o valor corrente do contador. Finalmente, essa classe tem dois métodos de atualização – o método incrementCount, que incrementa o contador, e o método decrementCount, que decrementa o contador. Na verdade, esta é uma classe extremamente maçante, mas pelo menos mostra a sintaxe e a estrutura de uma classe Java. Mostra também que uma classe Java não precisa ter um método chamado de main (mas tal classe não consegue fazer nada sozinha).

O nome da classe, método ou variável em Java é chamado de ***identificador*** e pode ser qualquer string de caracteres, desde que inicie por uma letra e seja composto por letras, números e caracteres sublinhados (onde "letra" e "número" podem ser de qualquer língua escrita definida no conjunto de caracteres Unicode). As exceções a essa regra geral para identificadores Java estão na Tabela 1.1.

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

Tabela 1.1 Lista de palavras reservadas Java. Essas palavras não podem ser usadas como nomes de variáveis ou de métodos em Java.

Modificadores de classes

Os modificadores de classes são palavras reservadas opcionais que precedem a palavra reservada **class**. Até agora, foram vistos exemplos que usavam a palavra reservada **public**. Em geral, os diferentes modificadores de classes e seu significado são os seguintes:

- O modificador de classe **abstract** descreve uma classe que possui métodos abstratos. Métodos abstratos são declarados com a palavra reservada **abstract** e são vazios (isto é, não possuem um bloco de comandos definindo o código do método). Se uma classe tem apenas métodos abstratos e nenhuma variável de instância, é mais adequado considerá-la uma interface (ver Seção 2.4), de forma que uma classe **abstract** é normalmente uma mistura de métodos abstratos e métodos verdadeiros. (Discutem-se classes abstratas e seus usos na Seção 2.4.)
- O modificador de classe **final** descreve uma classe que não pode ter subclasses. (Discute-se esse conceito no próximo capítulo.)
- O modificador de classe **public** descreve uma classe que pode ser instanciada ou estendida por qualquer coisa definida no mesmo pacote ou por qualquer coisa que importe a classe. (Isso é mais bem detalhado na Seção 1.8.) Todas as classes públicas são declaradas em arquivo próprio exclusivo nomeado *classname*.`java`, onde "*classname*" é o nome da classe.
- Se o modificador de classe **public** não é usado, então a classe é considerada ***amigável***. Isso significa que pode ser usada e instanciada por qualquer classe do mesmo ***pacote***. Esse é o modificador de classe default.

1.1.1 Tipos básicos

Os tipos dos objetos são determinados pela classe de origem. Em nome da eficiência e da simplicidade, Java ainda oferece os seguintes ***tipos básicos*** (também chamados de ***tipos primitivos***), que não são objetos:

boolean	valor booleano: true ou false
char	caracter Unicode de 16 bits
byte	inteiro com sinal em complemento de dois de 8 bits
short	inteiro com sinal em complemento de dois de 16 bits
int	inteiro com sinal em complemento de dois de 32 bits
long	inteiro com sinal em complemento de dois de 64 bits
loat	número de ponto flutuante de 32 bits (IEEE 754-1985)
double	número de ponto flutuante de 64 bits (IEEE 754-1985)

Uma variável declarada como tendo um desses tipos simplesmente armazena um valor desse tipo, em vez de uma referência para um objeto. Constantes inteiras, tais como 14 ou 195, são do tipo **int**, a menos que seguidas de imediato por um "L"ou "l", sendo, neste caso, do tipo **long**. Constantes de ponto flutuante, como 3.1415 ou 2.158*e*5, são do tipo **double**, a menos que seguidas de imediato por um "F" ou um "f", sendo, neste caso, do tipo **float**. O Trecho de Código 1.2 apresenta uma classe simples que define algumas variáveis locais de tipos básicos no método main.

```
public class Base {
   public static void main (String[ ] args) {
      boolean flag = true;
      char ch = 'A';
      byte b = 12;
      short s = 24;
      int i = 257;
      long l = 890L;                          // Observar o uso do "L" aqui
      float f = 3.1415F;                      // Observar o uso do "F" aqui
      double d = 2.1828;
      System.out.println ("flag = " + flag); // o "+" indica concatenação de strings
      System.out.println ("ch = " + ch);
      System.out.println ("b = " + b);
      System.out.println ("s = " + s);
      System.out.println ("i = " + i);
      System.out.println ("l = " + l);
      System.out.println ("f = " + f );
      System.out.println ("d = " + d);
   }
}
```

Trecho de Código 1.2 A classe Base mostrando o uso dos tipos básicos.

Comentários

Observe o uso de comentários neste e nos outros exemplos. Os comentários são anotações para uso de humanos e não são processados pelo compilador Java. Java permite dois tipos de comentários – comentários de bloco e comentários de linha – usados para definir o texto a ser ignorado pelo compilador. Em Java, usa-se um /* para começar um bloco de comentário e um */ para fechá-lo. Deve-se destacar os comentários iniciados por /**, pois tais comentários têm um formato especial, que permite que um programa chamado de Javadoc os leia e automaticamente gere documentação para programas Java. A sintaxe e interpretação dos comentários Javadoc será discutida na Seção 1.9.3

Além de comentários de bloco, Java usa o // para começar comentários de linha e ignorar tudo mais naquela linha. Por exemplo:

```
/*
 * Este é um bloco de comentário
 */
// Este é um comentário de linha
```

Saída da classe Base

A saída resultante da execução da classe Base (método main) é mostrada na Figura 1.2.

```
flag = true
ch = A
b = 12
s = 24
i = 257
l = 890
f = 3.1415
d = 2.1828
```

Figura 1.2 Saída da classe Base.

Mesmo não se referindo a objetos, variáveis dos tipos básicos são úteis no contexto de objetos, porque são usadas para definir variáveis de instâncias (ou campos) dentro de um objeto. Por exemplo, a classe Counter (Trecho de Código 1.1) possui uma única variável de instância do tipo **int**. Outra característica adicional de Java é o fato de que variáveis de instância sempre recebem um valor inicial quando o objeto que as contém é criado (seja zero, falso ou um caracter nulo, dependendo do tipo).

1.1.2 Objetos

Em Java, um objeto novo é criado a partir de uma classe usando-se o operador **new**. O operador **new** cria um novo objeto a partir de uma classe especificada e retorna uma ***referência*** para este objeto. Para criar um objeto de um tipo específico, deve-se seguir o uso do operador **new** por uma chamada a um construtor daquele tipo de objeto. Pode-se usar qualquer construtor que faça parte da definição da classe, incluindo o construtor default (que não recebe argumentos entre os parênteses). Na Figura 1.3, apresentam-se vários exemplos de uso do operador **new** que criam novos objetos e atribuem uma referência para estes a uma variável.

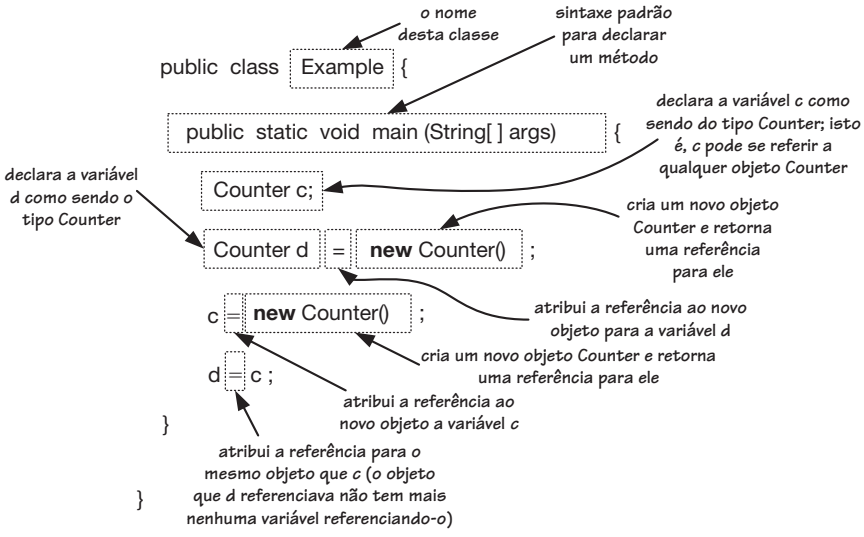


Figura 1.3 Exemplos de uso do operador **new**.

A chamada do operador **new** sobre um tipo de classe faz com que ocorram três eventos:

- Um novo objeto é dinamicamente alocado na memória, e todas as variáveis de instância são inicializadas com seus valores padrão. Os valores padrão são **null** para variáveis objeto e 0 para todos os tipos base, exceto as variáveis **boolean** (que são **false** por default).
- O construtor para o novo objeto é chamado com os parâmetros especificados. O construtor atribui valores significativos para as variáveis de instância e executa as computações adicionais que devam ser feitas para criar este objeto.
- Depois de o construtor retornar, o operador **new** retorna uma referência (isto é, um endereço de memória) para o novo objeto recém-criado. Se a expressão está na forma de uma atribuição, então este endereço é armazenado na variável objeto, e a variável objeto passa a ***referir*** o objeto recém-criado.

Objetos numéricos

Às vezes, quer-se armazenar números como objetos, mas os tipos básicos não são objetos, como já se observou. Para contornar esse problema, Java define uma classe especial para cada tipo básico numérico. Essas classes são chamadas de ***classes numéricas***. Na Tabela 1.2, estão os tipos básicos numéricos e as classes numéricas correspondentes, juntamente com exemplos de como se criam e se acessam os objetos numéricos. Desde o Java SE 5 a operação de criação é executada automaticamente sempre que se passa um número básico para um método que esteja esperando o objeto correspondente. Da mesma forma, o método de acesso correspondente é executado automaticamente sempre que se deseja atribuir o valor do objeto Número correspondente a um tipo numérico básico.

Tipo base	*Nome da classe*	*Exemplo de criação*	*Exemplo de acesso*
byte	Byte	n = new Byte((byte)34);	n.byteValue( )
short	Short	n = new Short((short)100);	n.shortValue( )
int	Integer	n = new Integer(1045);	n.intValue( )
long	Long	n = new Long(10849L);	n.longValue( )
float	Float	n = new Float(3.934F);	n.floatValue( )
double	Double	n = new Double(3.934);	n.doubleValue( )

Tabela 1.2 Classes numéricas de Java. Para cada classe é fornecido o tipo básico correspondente e expressões exemplificadoras de criação e acesso a esses objetos. Em cada linha, admite-se que a variável n é declarada com o nome de classe correspondente.

Objetos string

Uma string é uma sequência de caracteres que provêm de algum ***alfabeto*** (conjunto de todos os ***caracteres*** possíveis). Cada caracter *c* que compõe uma string *s* pode ser referenciado por seu índice na string, a qual é igual ao número de caracteres que vem antes de *c* em *s* (dessa forma, o primeiro caractere tem índice 0). Em Java, o alfabeto usado para definir strings é o conjunto internacional de caracteres Unicode, um padrão de codificação de caracteres de 16 bits que cobre as línguas escritas mais usadas. Outras linguagens de programação tendem a usar o conjunto de caracteres ASCII, que é menor (corresponde a um subconjunto do alfabeto Unicode baseado em um padrão de codificação de 7 bits). Além disso, Java define uma classe especial embutida de objetos chamados de objetos String.

Por exemplo, uma string *P* pode ser

```
"hogs and dogs"
```

que tem comprimento 13 e pode ter vindo da página Web de alguém. Nesse caso, o caractere de índice 2 é '`g`' e o caractere de índice 5 é '`a`'. Por outro lado, *P* poderia ser a string `"CGTAATAGTTAATCCG"`, que tem comprimento 16 e pode ser proveniente de uma aplicação científica de sequenciamento de DNA, onde o alfabeto é `{G,C,A,T}`.

Concatenação

O processamento de strings implica em lidar com strings. A operação básica para combinar strings chama-se ***concatenação,*** a qual toma uma string *P* e uma string *Q* e as combina em uma nova string denotada *P* + *Q*, que consiste em todos os caracteres de *P* seguidos por todos os caracteres de *Q*. Em Java, o operador "+" age exatamente dessa maneira quando aplicado sobre duas strings. Sendo assim, em Java é válido (e muito útil) escrever uma declaração de atribuição do tipo:

```
String s = "kilo" + "meters";
```

Essa declaração define uma variável `s` que referencia objetos da classe `String` e lhe atribui a string `"kilometers"`. (Mais adiante, neste capítulo, serão discutidos mais detalhadamente comandos de atribuição e expressões como a apresentada.) Pressupõe-se ainda que todo objeto Java tem um método predefinido chamado de `toString()` que retorna a string associada ao objeto. Esta descrição da classe `String` deve ser suficiente para a maioria dos usos. Analisaremos a classe `String` e sua "parente", a classe `StringBuffer`, na Seção 12.1.

Referências para objetos

Como mencionado acima, a criação de um objeto novo envolve o uso do operador **new** para alocar espaço em memória para o objeto e usar o construtor do objeto para inicializar esse espaço. A localização ou ***endereço*** deste espaço normalmente é atribuída para uma variável ***referência***. Consequentemente, uma variável referência pode ser entendida como sendo um "ponteiro" para um objeto. Isso é como se a variável fosse o suporte de um controle remoto que pudesse ser usado para controlar o objeto recém-criado (o dispositivo). Ou seja, a variável tem uma maneira de apontar para o objeto e solicitar que ele faça coisas ou acessar seus dados. Esse conceito pode ser visto na Figura 1.4.

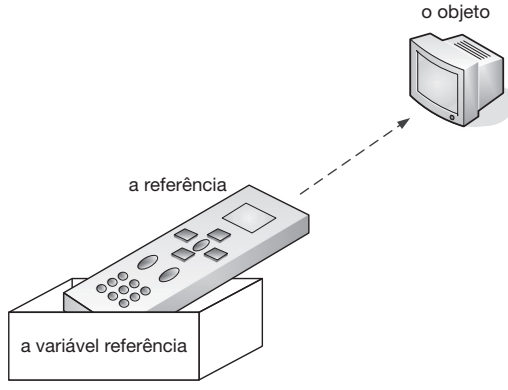


Figura 1.4 Demonstrando o relacionamento entre objetos e variáveis referência. Quando se atribui uma referência para um objeto (isto é, um endereço de memória) para uma variável referência, é como se fosse armazenado um controle remoto do objeto naquela variável.

O operador ponto

Toda a variável referência para objeto deve referir-se a algum objeto, a menos que seja **null**, caso em que não aponta para nada. Seguindo com a analogia do controle remoto, uma referência **null** é um suporte de controle remoto vazio. Inicialmente, a menos que se faça a variável referência apontar para alguma coisa por meio de uma atribuição, ela é **null**.

Pode haver, na verdade, várias referências para um mesmo objeto, e cada referência para um objeto específico pode ser usada para chamar métodos daquele objeto. Essa situação é equivalente a existirem vários controles remotos capazes de atuar sobre o mesmo dispositivo. Qualquer um dos controles pode ser usado para fazer alterações no dispositivo (como alterar o canal da televisão). Observe que, se um controle remoto é usado para alterar o dispositivo, então o (único) objeto apontado por todos os controles se altera. Da mesma forma, se uma variável referência for usada para alterar o estado do objeto, então seu estado muda para todas as suas referências. Esse comportamento vem do fato de que são muitas referências, mas todas apontando para o mesmo objeto.

Um dos principais usos de uma variável referência é acessar os membros da classe à qual pertence o objeto, a instância da classe. Ou seja, uma variável referência é útil para acessar os métodos e as variáveis de instância associadas com um objeto. Esse acesso é feito pelo operador ponto (“.”). Chama-se um método associado com um objeto usando o nome da variável referência seguido do operador ponto, e então o nome do método e seus parâmetros.

Isso ativa o método com o nome especificado associado ao objeto referenciado pela variável referência. De maneira opcional, podem ser passados vários parâmetros. Se existirem vários métodos com o mesmo nome definido para este objeto, então a máquina de execução do Java irá usar aquele cujo número de parâmetros e tipos melhor combinam. O nome de um método combinado com a quantidade e o tipo de seus parâmetros chama-se ***assinatura*** do método, uma vez que todas essas partes são usadas para determinar o método correto para executar uma certa chamada de método. Considerem-se os seguintes exemplos:

```
oven.cookDinner( );
oven.cookDinner(food);
oven.cookDinner(food, seasoning);
```

Cada uma dessas chamadas se refere, na verdade, a métodos diferentes, definidos com o mesmo nome na classe a qual pertencem. Observa-se, entretanto, que a assinatura de um método em Java não inclui o tipo de retorno do método, de maneira que Java não permite que dois métodos com a mesma assinatura retornem tipos diferentes.

Variáveis de instância

Classes Java podem definir ***variáveis de instância***, também chamadas de ***campos***. Essas variáveis representam os dados associados com os objetos de uma classe. As variáveis de instância devem ter um ***tipo,*** que pode tanto ser um ***tipo básico*** (como **int, float, double**) ou um ***tipo referência*** (como na analogia do controle remoto), isto é,

uma classe, como String; uma interface (ver Seção 2.4) ou um arranjo (ver Seção 1.5). Uma instância de variável de um tipo básico armazena um valor do tipo básico, enquanto variáveis de instância, declaradas usando-se um nome de classe, armazenam uma ***referência*** para um objeto daquela classe.

Continuando com a analogia entre variáveis referência e controles remotos, variáveis de instância são como parâmetros do dispositivo que podem tanto ser lidos, como alterados usando-se o controle remoto (tais como os controles de volume e canal do controle remoto de uma televisão). Dada uma variável referência *v*, que aponta para um objeto *o*, pode-se acessar qualquer uma das variáveis de instância de *o* que as regras de acesso permitirem. Por exemplo, variáveis de instância **públicas** podem ser acessadas por qualquer pessoa. Usando o operador ponto, pode-se ***obter*** o valor de qualquer variável de instância, *i*, usando-se *v.i* em uma expressão aritmética. Da mesma forma, pode-se ***alterar*** o valor de qualquer variável de instância *i*, escrevendo *v.i* no lado esquerdo do operador de atribuição ("="). (Ver Figura 1.5.) Por exemplo, se gnome se refere a um objeto Gnome que tem as variáveis de instância públicas name e age, então os seguintes comandos são possíveis:

```
gnome.name = "Professor Smythe";
gnome.age = 132;
```

Entretanto, uma referência para objeto não tem de ser apenas uma variável referência; pode ser qualquer expressão que retorna uma referência para objeto.

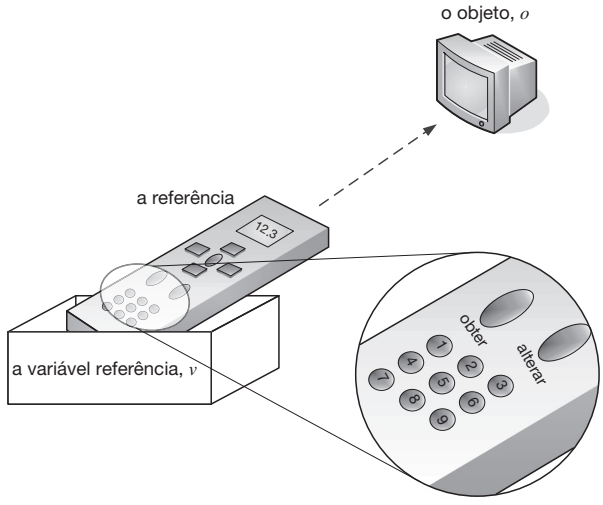


Figura 1.5 Demonstrando a maneira como uma referência para objeto pode ser usada para obter ou alterar variáveis de instância em um objeto (assumindo que se tem acesso a essas variáveis).

Modificadores de variáveis

Em alguns casos, o acesso direto a uma variável de instância de um objeto pode não estar habilitado. Por exemplo, uma variável de instância declarada como **privada** em alguma classe só pode ser acessada pelos métodos definidos dentro da classe. Tais variáveis de instância são parecidas com parâmetros de dispositivo que não podem ser acessados diretamente pelo controle remoto. Por exemplo, alguns dispositivos têm parâmetros internos que só podem ser lidos ou alterados por técnicos da fábrica (e o usuário não está autorizado a alterá-los sem violar a garantia do dispositivo).

Quando se declara uma variável de instância, pode-se, opcionalmente, definir um modificador de variável, seguido pelo tipo e identificador daquela variável. Além disso, também é opcional atribuir um valor inicial para a variável (usando o operador de atribuição "="). As regras para o nome da variável são as mesmas de qualquer outro identificador Java. O tipo da variável pode ser tanto um tipo básico, indicando que a variável armazena valores daquele tipo, ou um nome de classe, indicando que a variável é uma ***referência*** para um objeto desta classe. Por fim, o valor inicial opcional que se pode atribuir a uma variável de instância deve combinar com o tipo da variável. Como exemplo, definiu-se a classe Gnome, que contém várias definições de variáveis de instância, apresentada no Trecho de Código 1.3.

O ***escopo*** (ou visibilidade) de uma variável de instância pode ser controlado pelo uso dos seguintes ***modificadores de variáveis***:

- **public:** qualquer um pode acessar variáveis de instância públicas.
- **protected:** apenas métodos do mesmo pacote ou subclasse podem acessar variáveis de instância protegidas.
- **private:** apenas métodos da mesma classe (excluindo métodos de uma subclasse) podem acessar variáveis de instâncias privadas.
- Se nenhum dos modificadores acima for usado, então a variável de instância é considerada amigável. Variáveis de instância amigáveis podem ser acessadas por qualquer classe no mesmo pacote. Os pacotes são discutidos detalhadamente na Seção 1.8.

Além dos modificadores de escopo de variável, existem também os seguintes modificadores de uso:

- **static:** a palavra reservada **static** é usada para declarar uma variável que é associada com a classe, não com instâncias individuais daquela classe. Variáveis static são usadas para armazenar informações globais sobre uma classe (por exemplo, uma variável static pode ser usada para armazenar a quantidade total de objetos Gnome criados). Variáveis static existem mesmo se nenhuma instância de sua classe for criada.
- **final:** uma variável de instância final é um tipo de variável para o qual se ***deve*** atribuir um valor inicial, e para a qual, a partir de então, não é possível atribuir um novo valor. Se for de um tipo básico, então é uma constante (como a constante MAX_HEIGHT na classe Gnome). Se uma variável objeto é **final**, então irá sempre se referir ao mesmo objeto (mesmo se o objeto alterar seu estado interno).

```
public class Gnome {
   // Variáveis de instância:
   public String name;
   public int age;
   public Gnome gnomeBuddy;
   private boolean magical = false;
   protected double height = 2.6;
   public static final int MAX_HEIGHT = 3; // altura máxima
   // Construtores:
   Gnome(String nm, int ag, Gnome bud, double hgt) { // totalmente parametrizado
     name = nm;
     age = ag;
     gnomeBuddy = bud;
     height = hgt;
   }
   Gnome( ) { // Construtor default
     name = "Rumple";
     age = 204;
     gnomeBuddy = null;
     height = 2.1;
   }
   // Métodos:
   public static void makeKing (Gnome h) {
     h.name = "King" + h.getRealName( );
     h.magical = true; // Apenas a classe Gnome pode referenciar este campo.
   }
   public void makeMeKing ( ) {
     name = "King" + getRealName( );
     magical = true;
   }
   public boolean isMagical( ) { return magical; }
   public void setHeight(int newHeight) { height = newHeight; }
   public String getName( ) { return "I won't tell!"; }
   public String getRealName( ) { return name; }
   public void renameGnome(String s) { name = s; }
}
```

Trecho de código 1.3 A classe Gnome.

Observa-se o uso das variáveis de instância no exemplo da classe Gnome. As variáveis age, magical e height são de tipos básicos, a variável name é uma referência para uma instância da classe predefinida String, e a variável gnomeBuddy é uma referência para um objeto da classe que está sendo definida. A declaração da variável de instância MAX_HEIGHT está tirando proveito desses dois modificadores para definir uma "variável" que tem um valor constante fixo. Na verdade, valores constantes associados a uma classe sempre devem ser declarados **static** e **final**.

1.1.3 Tipos enumerados

Desde a versão SE 5, Java suporta tipos enumerados chamados de ***enums***. Esses tipos são permitidos apenas para que se possa obter valores provenientes de conjuntos específicos de nomes. Eles são declarados dentro de uma classe como segue:

modificador **enum** *nome* {$nome\_valor_0$, $nome\_valor_1$, ..., $nome\_valor_{n-1}$}

onde o *modificador* pode ser vazio, **public**, **protected** ou **private**. O nome desta enumeração, *nome,* pode ser qualquer identificador Java. Cada um dos identificadores de valor, $nome\_valor_i$, é o nome de um possível valor que variáveis desse tipo podem assumir. Cada um desses nomes de valor pode ser qualquer identificador Java legal, mas, por convenção, normalmente eles começam por letra maiúscula. Por exemplo, a seguinte definição de tipo enumerado pode ser útil em um programa que deve lidar com datas:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
```

Uma vez definido, um tipo enumerado pode ser usado na definição de outras variáveis da mesma forma que um nome de classe. Entretanto, como o Java conhece todos os nomes dos valores possíveis para um tipo enumerado, se um tipo enumerado for usado em uma expressão string, o Java irá usar o nome do valor automaticamente. Tipos enumerados também possuem alguns métodos predefinidos, incluindo o método valueOf, que retorna o valor enumerado que é igual a uma determinada string. Um exemplo de uso de tipo enumerado pode ser visto no Trecho de Código 1.4.

```
public class DayTripper {
  public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
  public static void main(String[ ] args) {
    Day d = Day.MON;
    System.out.println("Initially d is " + d);
    d = Day.WED;
    System.out.println("Then it is " + d);
    Day t = Day.valueOf("WED");
    System.out.println("I say d and t are the same: " + (d == t));
  }
}
```

A saída deste programa é:

```
Initially d is MON
Then it is WED
I say d and t are the same: true
```

Trecho de Código 1.4 Um exemplo de uso de tipo enumerado.

1.2 Métodos

Os métodos em Java são conceitualmente similares a procedimentos e funções em outras linguagens de alto nível. Normalmente correspondem a "trechos" de código que podem ser chamados em um objeto específico (de alguma classe). Os métodos podem admitir parâmetros como argumentos, e seu comportamento depende do objeto ao qual pertencem e dos valores passados por qualquer parâmetro. Todo método em Java é especificado no corpo de uma classe. A definição de um método compreende duas partes: a ***assinatura***, que define o nome e os parâmetros do método, e o ***corpo***, que define o que o método realmente faz.

Um método permite ao programador enviar uma mensagem para um objeto. A assinatura do método especifica como uma mensagem deve parecer, e o corpo do método especifica o que o objeto irá fazer quando receber tal mensagem.

Declarando métodos

A sintaxe da definição de um método é como segue:

modificadores tipo nome($tipo_0$ $parâmetro_0$, ..., $tipo_{n-1}$ $parâmetro_{n-1}$*)* {
    // corpo do método . . .
}

Cada uma das partes dessa declaração é importante e será descrita em detalhes nesta seção. A seção de *modificadores* usa os mesmos tipos de modificadores de escopo que podem ser usados para variáveis, como **public**, **protected** e **static**, com significados parecidos. A seção *tipo* define o tipo de retorno do método. O *nome* é o nome do método, e pode ser qualquer identificador Java válido. A lista de parâmetros e seus tipos declara as variáveis locais que correspondem aos valores que são passados como argumentos para o método. Cada declaração de tipo, $tipo_i$, pode ser qualquer nome tipo Java e cada $parâmetro_i$ pode ser qualquer identificador Java. Esta lista de identificadores e seus tipos pode ser vazia, o que significa que não existem valores para serem passados para esse método quando for acionado. As variáveis parâmetro, assim como as variáveis de instância da classe, podem ser usadas dentro do corpo do método. Da mesma forma, os outros métodos dessa classe podem ser chamados de dentro do corpo de um método.

Quando um método de uma classe é acionado, é chamado para uma instância específica da classe, e pode alterar o estado daquele objeto (exceto o método **static**, que é associado com a classe propriamente dita). Por exemplo, invocando-se o método que segue em um gnome particular, altera-se seu nome.

```
public void renameGnome (String s) {
  name = s; // Alterando a variável de instância nome desse gnome.
}
```

Modificadores de métodos

Como as variáveis de instância, modificadores de métodos também podem restringir o escopo de um método:

- **public:** qualquer um pode chamar métodos públicos.
- **protected:** apenas métodos do mesmo pacote ou subclasse podem chamar um método protegido.
- **private:** apenas métodos da mesma classe (excluindo os métodos de subclasses) podem chamar um método privado.
- Se nenhum dos modificadores citados for usado, então o método é considerado amigável. Métodos amigáveis só podem ser chamados por objetos de classes do mesmo pacote.

Os modificadores de método acima podem ser precedidos por modificadores adicionais:

- **abstract:** um método declarado como **abstract** não terá código. A lista de parâmetros desse método é seguida por um ponto e vírgula, sem o corpo do método. Por exemplo:

```
public abstract void setHeight (double newHeight);
```

  Métodos abstratos só podem ocorrer em classes abstratas. A utilidade desta construção será analisada na Seção 2.4.

- **final:** este é um método que não pode ser sobrescrito por uma subclasse.
- **static:** este é um método associado com a classe propriamente dita e não com uma instância em particular. Métodos static também podem ser usados para alterar o estado de variáveis static associadas com a classe (desde que estas variáveis não tenham sido declaradas como sendo **final**).

Tipos de retorno

Uma definição de método deve especificar o tipo do valor que o método irá retornar. Se o método não retorna um valor, então a palavra reservada **void** deve ser usada. Se o tipo de retorno é **void**, o método é chamado de ***procedimento***; caso contrário, é chamado de ***função***. Para retornar um valor em Java, um método deve usar a palavra reservada **return** (e o tipo retornado deve combinar com o tipo de retorno do método). Na sequência, um exemplo de método (interno à classe Gnome) que é uma função:

```
public boolean isMagical ( ) {
    return magical;
}
```

Assim que um **return** é executado em uma função Java, a execução do método termina.

Funções Java podem retornar apenas um valor. Para retornar múltiplos valores em Java, deve-se combiná-los em um ***objeto composto*** cujas variáveis de instância incluam todos os valores desejados e, então, retornar uma referência para este objeto composto. Além disso, pode-se alterar o estado interno de um objeto que é passado para um método como outra forma de "retornar" vários resultados.

Parâmetros

Os parâmetros de um método são definidos entre parênteses, após o nome do método, separados por vírgulas. Um parâmetro consiste em duas partes: seu tipo e o seu nome. Se um método não tem parâmetros, então apenas um par de parênteses vazio é usado.

Todos os parâmetros em Java são passados ***por valor***, ou seja, sempre que se passa um parâmetro para um método, uma cópia do parâmetro é feita para uso no contexto do corpo do método. Ao se passar uma variável **int** para um método, o valor daquela variável é copiado. O método pode alterar a cópia, mas não o original. Quando se passa uma referência do objeto como parâmetro para um método, então essa referência é copiada da mesma forma. É preciso lembrar que podem existir muitas variáveis diferentes referenciando o mesmo objeto. A alteração da referência recebida dentro de um método não irá alterar a referência que foi passada para ele. Por exemplo, ao passar uma referência g da classe Gnome para um método que chama este parâmetro de h, então o método pode alterar a referência h de maneira que ela aponte para outro objeto, porém g continuará a referenciar o mesmo objeto anterior. O método, contudo, pode usar a referência h para mudar o estado interno do objeto, alterando assim o estado do objeto apontado por g (desde que g e h referenciem o mesmo objeto).

Construtores

Um ***construtor*** é um tipo especial de método que é usado para inicializar objetos novos quando de sua criação. Java tem uma maneira especial de declarar um construtor e uma forma especial de invocá-lo. Primeiro será analisada a sintaxe de declaração de um construtor:

modificadores nome($tipo_0$ *parâmetro*$_0$, ..., $tipo_{n-1}$ *parâmetro*$_{n-1}$) {
    // corpo do construtor . . .
}

Vê-se que a sintaxe é igual à de qualquer outro método, mas existem algumas diferenças essenciais. O nome do construtor, *nome*, deve ser o mesmo nome da classe que constrói. Se a classe se chama Fish, então o construtor deve se chamar Fish da mesma forma. Além disso, um construtor não possui parâmetro de retorno – seu tipo de retorno é de forma implícita, o mesmo que seu nome (que é também o nome da classe). Os modificadores de construtor, indicados acima como *modificadores*, seguem as mesmas regras que os métodos normais, exceto pelo fato de que construtores **abstract**, **static** ou **final** não são permitidos.

Por exemplo:

```
public Fish (int w, String n) {
  weight = w;
  name = n;
}
```

Definição e invocação de um construtor

O corpo de um construtor é igual ao corpo de um método normal, com duas pequenas exceções. A primeira diferença diz respeito ao conceito conhecido como cadeia de construtores, tópico discutido na Seção 2.2.3 e que não é importante agora.

A segunda diferença entre o corpo de um construtor e o corpo de um método comum é que o comando **return** não é permitido no corpo de um construtor. A finalidade deste corpo é ser usado para a inicialização dos dados associados com os objetos da classe correspondente, de forma que eles fiquem em um estado inicial estável quando criados.

Métodos construtores são ativados de uma única forma: ***devem*** ser chamados pelo operador **new**. Assim, a partir da ativação, uma nova instância da classe é automaticamente criada, e seu construtor é então chamado para inicializar as variáveis de instância e executar outros procedimentos de configuração. Por exemplo, considere-se a seguinte ativação de construtor (que corresponde também a uma declaração da variável myFish):

```
Fish myFish = new Fish (7, "Wally");
```

Uma classe pode ter vários construtores, mas cada um deve ter uma ***assinatura*** diferente, ou seja, eles devem ser distinguíveis pelo tipo e número de parâmetros que recebem.

O método main

Certas classes Java têm como objetivo a utilização por outras classes, e outras têm como finalidade definir programas executáveis†. Classes que definem programas executáveis devem conter outro tipo especial de método para uma classe – o método main. Quando se deseja executar um programa executável Java, referencia-se o nome da classe que define este programa, por exemplo, disparando o seguinte comando (em um shell Windows, Linux ou UNIX):

```
java Aquarium
```

Nesse caso, o sistema de execução de Java procura por uma versão compilada da classe Aquarium, e então ativa o método especial main dessa classe. Esse método deve ser declarado como segue:

† N. de T.: Utiliza-se a expressão "programa executável" (*stand-alone program*) neste contexto para indicar um programa que é executado sem a necessidade de um navegador, não se referindo a um arquivo binário executável.

```
public static void main(String[ ] args) {
    // corpo do método main . . .
}
```

Os argumentos passados para o método main pelo parâmetro args são os argumentos de linha de comando fornecidos quando o programa é chamado. A variável args é um arranjo de objetos String; ou seja, uma coleção de strings indexadas, com a primeira string sendo args[0], a segunda args[1] e assim por diante. (Mais será discutido sobre arranjos na Seção 1.5.)

Chamando um programa Java a partir da linha de comando

Programas Java podem ser chamados a partir da linha de comando usando o comando Java seguido do nome da classe Java que contém o método main que se deseja executar, mais qualquer argumento opcional. Por exemplo, o programa Aquarium poderia ter sido definido para receber um parâmetro opcional que especificasse o número de peixes no aquário. O programa poderia ser ativado digitando-se o seguinte em uma janela shell:

```
java Aquarium 45
```

para especificar que se quer um aquário com 45 peixes dentro dele. Nesse caso, args[0] se refere à string `"45"`. Uma característica interessante do método main é que ele permite a cada classe definir um programa executável, e um dos usos deste método é testar os outros métodos da classe. Dessa forma, o uso completo do método main é uma ferramenta eficaz para a depuração de coleções de classes Java.

Blocos de comandos e variáveis locais

O corpo de um método é um ***bloco de comandos***, ou seja, uma sequência de declarações e comandos executáveis definidos entre chaves "{" e "}". O corpo de um método e outros blocos de comandos podem conter também blocos de comandos aninhados. Além de comandos que executam uma ação, como ativar um método de algum objeto, os blocos de comandos podem conter declarações de ***variáveis locais***. Essas variáveis são declaradas no corpo do comando, em geral no início (mas entre as chaves "{" e "}"). As variáveis locais são similares a variáveis de instância, mas existem apenas enquanto o bloco de comandos está sendo executado. Tão logo o fluxo de controle saia do bloco, todas as variáveis locais internas deste não podem mais ser referenciadas. Uma variável local pode ser tanto um ***tipo base*** (como **int**, **float**, **double**) como uma ***referência*** para uma instância de alguma classe. Comandos e declarações simples em Java sempre encerram com ponto e vírgula, ou seja um ";".

Existem duas formas de declarar variáveis locais:

tipo nome;
tipo nome = valor_inicial;

A primeira declaração simplesmente define que o identificador, *nome*, é de um tipo específico.

A segunda declaração define o identificador, seu tipo e também inicializa a variável com um valor específico. Seguem alguns exemplos de inicialização de variáveis locais:

```
{
  double r;
  Point p1 = new Point (3, 4);
  Point p2 = new Point (8, 2);
  int i = 512;
  double e = 2.71828;
}
```

1.3 Expressões

Variáveis e constantes são usadas em ***expressões*** para definir novos valores e para modificar variáveis. Nesta seção, discute-se com mais detalhes como as expressões Java funcionam. Elas envolvem o uso de ***literais***, ***variáveis*** e ***operadores***. Como as variáveis já foram examinadas, serão focados rapidamente os literais e analisados os operadores com mais detalhe.

1.3.1 Literais

Um ***literal*** é qualquer valor "constante" que pode ser usado em uma atribuição ou outro tipo de expressão. Java admite os seguintes tipos de literais:

- A referência para objeto **null** (este é o único literal que é um objeto e pode ser qualquer tipo de referência)
- Booleano: **true** e **false**.
- Inteiro: o default para um inteiro como 176 ou -52 é ser do tipo **int**, que corresponde a um inteiro de 32 bits. Um literal representando um inteiro longo deve terminar por um "L" ou "l", por exemplo, 176L ou -52l, e corresponde a um inteiro de 64 bits.
- Ponto flutuante: o default para números de ponto flutuante, como 3.1415 e 135.23, é ser do tipo **double**. Para especificar um literal **float**, ele deve terminar por um "F"ou um "f". Literais de ponto flutuante em notação exponencial também são aceitos, por exemplo, 3.14E2 ou 0.19e10; a base assumida é 10.
- Caracteres: assume-se que constantes de caracteres em Java pertencem ao alfabeto Unicode. Normalmente, um caractere é definido como um símbolo individual entre aspas simples. Por exemplo, 'a' e '?' são constantes caracteres. Além desses, Java define as seguintes constantes especiais de caracteres:

'\n'	(nova linha)	'\t'	(tabulação)
'\b'	(retorna um espaço)	'\r'	(retorno)
'\f'	(alimenta formulário)	'\\'	(barra invertida)
'\''	(aspas simples)	'\"'	(aspas duplas)

- String literal: uma string é uma sequência de caracteres entre aspas duplas, como

```
"dogs cannot climb trees"
```

1.3.2 Operadores

As expressões em Java implicam em concatenar literais e variáveis usando operadores. Os operadores de Java serão analisados nesta seção.

O operador de atribuição

O operador padrão de atribuição em Java é "=". É usado na atribuição de valores para variáveis de instância ou variáveis locais. Sua sintaxe é:

variável = expressão

onde *variável* se refere a uma variável que pode ser referenciada no bloco de comandos que contém esta expressão. O valor de uma operação de atribuição é o valor da expressão que é atribuída. Sendo assim, se i e j são declaradas do tipo **int**, é correto ter um comando de atribuição como o seguinte:

```
i = j = 25; // funciona porque o operador '=' é avaliado da direita para a esquerda
```

Operadores aritméticos

Os operadores que seguem são os operadores binários aritméticos de Java:

+	adição
−	subtração
*	multiplicação
/	divisão
%	operador módulo

O operador módulo também é conhecido como o operador de "resto", na medida em que fornece o resto de uma divisão de números inteiros. Com frequência, usamos "mod" para indicar o operador de módulo, e o definimos formalmente como:

$$n \bmod m = r$$

de maneira que

$$n = mq + r,$$

para um inteiro q e $0 \leq r < n$.

Java também fornece o operador unário menos $(-)$, que pode ser colocado na frente de qualquer expressão aritmética para inverter seu sinal. É possível utilizar pa-

rênteses em qualquer expressão para definir a ordem de avaliação. Java utiliza ainda uma regra de precedência de operadores bastante intuitiva para determinar a ordem de avaliação quando não são usados parênteses. Ao contrário de C++, Java não permite a sobrecarga de operadores.

Operadores de incremento e decremento

Da mesma forma que C e C++, Java oferece operadores de incremento e decremento. De forma mais específica, oferece os operadores incremento de um (++) e decremento de um (−−). Se tais operadores são usados na frente de um nome de variável, então 1 é somado ou subtraído à variável, e seu valor é empregado na expressão. Se for utilizado depois do nome da variável, então primeiro o valor é usado, e depois a variável é incrementada ou decrementada de 1. Assim, por exemplo, o trecho de código

```
int i = 8;
int j = i++;
int k = ++i;
int m = i--;
int n = 9 + i++;
```

atribui 8 para *j*, 10 para *k*, 10 para *m*, 18 para *n* e deixa *i* com o valor 10.

Operadores lógicos

Java oferece operadores padrão para comparações entre números:

<	menor que
<=	menor que ou igual a
==	igual a
!=	diferente de
>=	maior que ou igual a
>	maior que

Os operadores == e != também podem ser usados como referências para objetos. O tipo resultante de uma comparação é **boolean**.

Os operadores que trabalham com valores **boolean** são os seguintes:

!	negação (prefixado)
&&	e condicional
||	ou condicional

Os operadores booleanos && e || não avaliarão o segundo operando em suas expressões (para a direita) se isso não for necessário para determinar o valor da expressão. Esse recurso é útil, por exemplo, para construir expressões booleanas em que primeiro é testado se uma determinada condição se aplica (como uma referência não ser **null**) e, então, testa-se uma condição que geraria uma condição de erro, se o primeiro teste falhasse.

Operadores sobre bits

Java fornece também os seguintes operadores sobre bits para inteiros e booleanos:

~	complemento sobre bits (operador prefixado unário)
&	e sobre bits
|	ou sobre bits
^	ou exclusivo sobre bits
<<	deslocamento de bits para a esquerda, preenchendo com zeros
>>	deslocamento de bits para a direita, preenchendo com bits de sinal
>>>	deslocamento de bits para a direita, preenchendo com zeros

Operadores operacionais de atribuição

Além do operador de atribuição padrão (=), Java também oferece um conjunto de outros operadores de atribuição que têm efeitos colaterais operacionais. Esses outros tipos de operadores são da forma:

variável op = expressão

onde *op* é um operador binário. Essa expressão é equivalente a

variável = variável op expressão

excetuando-se que, se *variável* contém uma expressão (por exemplo, um índice de arranjo), a expressão é avaliada apenas uma vez. Assim, o trecho de código

```
a[5] = 10;
i = 5;
a[i++] += 2;
```

deixa a[5] com o valor 12 e i com o valor 6.

Concatenação de strings

As strings podem ser compostas usando o operador de ***concatenação*** (+), de forma que o código

```
String rug = "carpet";
String dog = "spot";
String mess = rug + dog;
String answer = mess + "will cost me" + 5 + " hours!";
```

terá o efeito de fazer answer apontar para a string

```
"carpetspot will cost me 5 hours!"
```

Esse exemplo também mostra como Java converte constantes que não são string em strings, quando estas estão envolvidas em uma operação de concatenação de strings.

Precedência de operadores

Os operadores em Java têm uma dada preferência, ou precedência, que determina a ordem na qual as operações são executadas quando a ausência de parênteses ocasiona ambiguidades na avaliação. Por exemplo, é necessário que exista uma forma de decidir se a expressão "`5+2*3`" tem valor 21 ou 11 (em Java, o valor é 11).

A Tabela 1.3 apresenta a precedência dos operadores em Java (que, coincidentemente, é a mesma de C).

Precedência de operadores		
	Tipo	**Símbolos**
1	operadores pós-fixados operadores pré-fixados *cast* (coerção)	*exp* ++ *exp* −− ++*exp* −− *exp* +*exp* −*exp* ∼*exp* !*exp* (*type*) *exp*
2	mult./div.	* / %
3	soma/subt.	+ −
4	deslocamento	<< >> >>>
5	comparação	< <= > >= **instanceof**
6	igualdade	== !=
7	"e" bit a bit	&
8	"xor" bit a bit	^
9	"ou" bit a bit	|
10	"e"	&&
11	"ou"	||
12	condicional	*expressão_booleana*? *valor_se_true* : *valor_se_false*
13	atribuição	= += −= *= /= %= >>= <<= >>>=&= ^=|=

Tabela 1.3 As regras de precedência de Java. Os operadores em Java são avaliados de acordo com a ordem acima se não forem utilizados parênteses para determinar a ordem de avaliação. Os operadores na mesma linha são avaliados da esquerda para a direita (exceto atribuições e operações pré-fixadas, que são avaliadas da direita para a esquerda), sujeitos à regra de avaliação condicional para as operações booleanas **e** e **ou**. As operações são listadas da precedência mais alta para a mais baixa (usamos *exp* para indicar uma expressão atômica ou entre parênteses). Sem parênteses, os operadores de maior precedência são executados depois de operadores de menor precedência.

Discutiu-se até agora quase todos os operadores listados na Tabela 1.3. Uma exceção importante é o operador condicional, o que implica avaliar uma expressão booleana e então tomar o valor apropriado, dependendo de a expressão booleana ser verdadeira ou falsa. (O uso do operador **instanceof** será analisado no próximo capítulo.)

1.3.3 Conversores e autoboxing/unboxing em expressões

A conversão é uma operação que nos permite alterar o tipo de uma variável. Em essência, pode-se ***converter*** uma variável de um tipo em uma variável equivalente de outro tipo. Os conversores podem ser úteis para fazer certas operações numéricas e de entrada e saída. A sintaxe para converter uma variável para um tipo desejado é a seguinte:

(*tipo*) *exp*

onde *tipo* é o tipo que se deseja que a expressão *exp* assuma. Existem dois tipos fundamentais de conversores que podem ser aplicados em Java. Pode-se tanto converter tipos de base numérica como tipos relacionados com objetos. Agora será discutida a conversão de tipos numéricos e strings, e a conversão de objetos será analisada na Seção 2.5.1. Por exemplo, pode ser útil converter um **int** em um **double** de maneira a executar operações como uma divisão.

Conversores usuais

Quando se converte um **double** em um **int**, pode-se perder a precisão. Isso significa que o valor double resultante será arredondado para baixo. Mas pode-se converter um **int** em um **double** sem essa preocupação. Por exemplo, considere o seguinte:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int)d1;              // i1 tem valor 3
int i2 = (int)d2;              // i2 tem valor 3
double d3 = (double)i2;    // d3 tem valor 3.0
```

Convertendo operadores

Alguns operadores binários, como o de divisão, terão resultados diferentes dependendo dos tipos de variáveis envolvidas. Devemos ter cuidado para garantir que tais operações executem seus cálculos em valores do tipo desejado. Quando usada com inteiros, por exemplo, a divisão não mantém a parte fracionária. No caso de uso com double, a divisão conserva esta parte, como ilustra o exemplo a seguir:

```
int i1 = 3;
int i2 = 6;
dresult = (double)i1 / (double)i2;    // dresult tem valor 0.5
dresult = i1 / i2;                    // dresult tem valor 0.0
```

Observe que a divisão normal para números reais foi executada quando i1 e i2 foram convertidos em double. Quando i1 e i2 não foram convertidos, o operador " / " executou uma divisão inteira e o resultado de i1 / i2 foi o **int** 0. Java executou uma ***conversão implícita*** para atribuir um valor **int** ao resultado **double**. A conversão implícita será estudada a seguir.

Conversores implícitos e autoboxing/unboxing

Existem casos em que o Java irá executar uma ***conversão implícita***, de acordo com o tipo da variável atribuída, desde que não haja perda de precisão. Por exemplo:

```
int iresult, i = 3;
double dresult, d = 3.2;
dresult = i / d;          // dresult tem valor 0.9375. i foi convertido para double
iresult = i / d;          // perda de precisão -> isso é um erro de compilação;
iresult = (int) i / d;    // iresult é 0, uma vez que a parte fracionária será perdida.
```

Considerando que Java não executará conversões implícitas onde houver perda de precisão, a conversão explícita da última linha do exemplo é necessária.

A partir do Java SE 5, existe um novo tipo de conversão implícita entre objetos numéricos, como Integer e Float, e seus tipos básicos relacionados, como **int** e **float**. Sempre que um objeto numérico for esperado como parâmetro para um método, o tipo básico correspondente pode ser informado. Nesse caso, o Java irá proceder uma conversão implícita chamada de ***autoboxing***, que irá converter o tipo base para o objeto numérico correspondente. Da mesma forma, sempre que um tipo base for esperado em uma expressão envolvendo um objeto numérico, o objeto numérico será convertido no tipo base correspondente em uma operação chamada de ***unboxing***.

Existem, entretanto, alguns cuidados a serem tomados no uso de autoboxing e unboxing. O primeiro é que, se uma referência numérica for **null**, então qualquer tentativa de unboxing irá gerar um erro de **NullPointerException**. Em segundo, o operador "==" é usado tanto para testar a igualdade de dois valores numéricos como também se duas referências para objetos apontam para o mesmo objeto. Assim, quando se testa a igualdade, deve-se evitar a conversão implícita fornecida por autoboxing/unboxing. Por fim, a conversão implícita de qualquer tipo toma tempo, então devemos minimizar nossa confiança nela se performance for um requisito.

A propósito, existe uma situação em Java em que apenas a conversão implícita é permitida: na concatenação de strings. Sempre que uma string é concatenada com qualquer objeto ou tipo base, o objeto ou tipo base é automaticamente convertido em uma string. Entretanto, a conversão explícita de um objeto ou tipo base para uma string não é permitida. Portanto, as seguintes atribuições são incorretas:

```
String s = (String) 4.5;                 // Isso está errado!
String t = "Value = " + (String) 13;     // Isso está errado!
String u = 22;                           // Isso está errado!
```

Para executar conversões para string, deve-se, em vez disso, usar o método toString apropriado ou executar uma conversão implícita via operação de concatenação. Assim, os seguintes comandos estão corretos:

```
String s = "" + 4.5;                 // correto, porém mau estilo de programação
String t = "Value = " + 13;          // correto
String u = Integer.toString(22);     // correto
```

1.4 Controle de fluxo

O controle de fluxo em Java é similar ao oferecido em outras linguagens de alto nível. Nesta seção, revisa-se a estrutura básica e a sintaxe do controle de fluxo em Java, incluindo retorno de métodos, comando **if**(**condicional**), comandos **switch** (de **seleção múltipla**), laços e formas restritas de "desvios" (os comandos **break** e **continue**).

1.4.1 Os comandos if e switch

Em Java, comandos condicionais funcionam da mesma forma que em outras linguagens. Eles fornecem a maneira de tomar uma decisão e então executar um ou mais blocos de comandos diferentes baseados no resultado da decisão.

O comando if

A sintaxe básica do comando **if** é a que segue:

```
if (expr_booleana)
    comando_se_verdade
else
    comando_se_falso
```

onde *expr_booleana* é uma expressão booleana e *comando_se_verdade* e *comando_se_falso* podem ser um comando simples ou um bloco de comandos entre chaves ("{" e "}"). Observa-se que, diferentemente de outras linguagens de programação, os valores testados por um comando **if** devem ser uma expressão booleana. Particularmente, não são uma expressão inteira. Por outro lado, como em outras linguagens similares, a cláusula **else** (e seus comandos associados) são opcionais. Existe também uma forma de agrupar um conjunto de testes booleanos como segue:

```
if (primeira_expressão_booleana)
    comando_se_verdade
else if (segunda_expressão_booleana)
    segundo_comando_se_verdade
else
    comando_se_falso
```

Se a primeira expressão booleana for falsa, então a segunda expressão booleana será testada e assim por diante. Um comando **if** pode ter qualquer quantidade de cláusulas **else.** Por segurança, quando se define um comando **if** complicado, usam-se chaves para agrupar o corpo do comando.

Por exemplo, a estrutura a seguir está correta:

```
if (snowLevel < 2) {
  goToClass( );
  comeHome( );
```

```
  }
  else if (snowLevel < 5) {
    goSledding( );
    haveSnowballFight( );
  }
  else
    stayAtHome( );
```

Comando switch

Java oferece o comando **switch** para controle de fluxo multivalorado, o que é especialmente útil com tipos enumerados. O exemplo a seguir é indicativo (baseado na variável d do tipo Day da Seção 1.1.3).

```
switch (d) {
  case MON:
    System.out.println("This is tough.");
    break;
  case TUE:
    System.out.println("This is getting better.");
    break;
  case WED:
    System.out.println("Half way there.");
    break;
  case THU:
    System.out.println("I can see the light.");
    break;
  case FRI:
    System.out.println("Now we are talking.");
    break;
  default:
    System.out.println("Day off!");
    break;
}
```

O comando **switch** avalia uma expressão inteira ou enumeração e faz com que o fluxo de controle desvie para o ponto marcado com o valor dessa expressão. Se não existir um ponto com tal marca, então o fluxo é desviado para o ponto marcado com "**default**". Entretanto, este é o único desvio explícito que o comando **switch** executa, e, a seguir, o controle "cai" através das cláusulas case se o código dessas cláusulas não for terminado por uma instrução **break** (que faz o fluxo de controle desviar para a próxima linha depois do comando **switch**).

1.4.2 Laços

Outro mecanismo de controle de fluxo importante em uma linguagem de programação é o laço. Java possui três tipos de laços.

Laços while

O tipo mais simples de laço em Java é o laço **while**. Este tipo de laço testa se uma certa condição é satisfeita e executa o corpo do laço enquanto essa condição for **true**. A sintaxe para testar uma condição antes de o corpo do laço ser executado é a seguinte:

while (*expressão_booleana*)
    corpo_do_laço

No início de cada iteração, o laço testa a expressão booleana*, boolean_exp*, e então, se esta resultar **true***,* executa o corpo do laço, *loop_statement.* O corpo do laço pode ser um bloco de comandos.

Considere-se, por exemplo, um gnomo tentando regar todas as cenouras de seu canteiro de cenouras, o que faz até seu regador ficar vazio. Se o regador estiver vazio logo no início, escreve-se o código para executar esta tarefa como segue:

```
public void waterCarrots( ) {
    Carrot current = garden.findNextCarrot ( );

    while (!waterCan.isEmpty ( )) {
        water (current, waterCan);
        current = garden.findNextCarrot ( );
    }
}
```

Lembre-se de que "!" em Java é o operador "not".

Laços for

Outro tipo de laço é o laço **for**. Na sua forma mais simples, os laços **for** oferecem uma repetição codificada baseada em um índice inteiro. Em Java, entretanto, pode-se fazer muito mais. A funcionalidade de um laço **for** é significativamente mais flexível. Sua estrutura se divide em quatro seções: inicialização, condição, incremento e corpo.

Definindo um laço for

Esta é a sintaxe de um laço **for** em Java

```
for (inicialização; condição; incremento)
    corpo_do_laço
```

onde cada uma das seções de *inicialização*, *condição* e *incremento* podem estar vazias.

Na seção *inicialização*, pode-se declarar uma variável índice que será válida apenas no escopo do laço **for**. Por exemplo, quando se deseja um laço indexado por um contador, e não há necessidade desse contador fora do contexto do laço **for**, então declara-se algo como o que segue:

```
for (int counter = 0; condição; incremento)
    corpo_do_laço
```

que declara uma variável counter cujo escopo é limitado apenas ao corpo do laço.

Na seção *condição*, especifica-se a condição de repetição (enquanto) do laço. Esta deve ser uma expressão booleana. O corpo do laço **for** será executado toda vez que a *condição* resultar **true**, quando avaliada no início de uma iteração potencial. Assim que a *condição* resultar **false**, então o corpo do laço não será executado e, em seu lugar, o programa executa o próximo comando depois do laço **for**.

Na seção de *incremento*, declara-se o comando de incremento do laço. O comando de incremento pode ser qualquer comando válido, o que permite uma flexibilidade significativa para a programação. Assim, a sintaxe do laço **for** é equivalente ao que segue:

```
inicialização;
while (condição) {
    comandos_do_laço
    incremento;
}
```

exceto pelo fato de que um laço **while** não pode ter uma condição booleana vazia, enquanto um laço **for** pode. O exemplo a seguir apresenta um exemplo simples de laço **for** em Java:

```
publi void eatApples (Apples apples) {
    numApples = apples.getNumApples ( );
    for (int × = 0; × < numApples; ×++) {
        eatApple (apples.getApple (×));
        spitOutCore ( );
    }
}
```