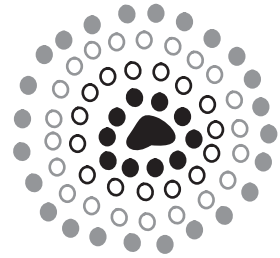
Capítulo

1 Conceitos Básicos de Programação Java



Conteúdo

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 tal 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 provê 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, tais como threads e sockets. O leitor que desejar aprender mais sobre Java deve observar as notas do final deste capítulo. Iniciamos com um programa que imprime "Hello Universe!" na tela, que é mostrado e dissecado na Figura 1.1.

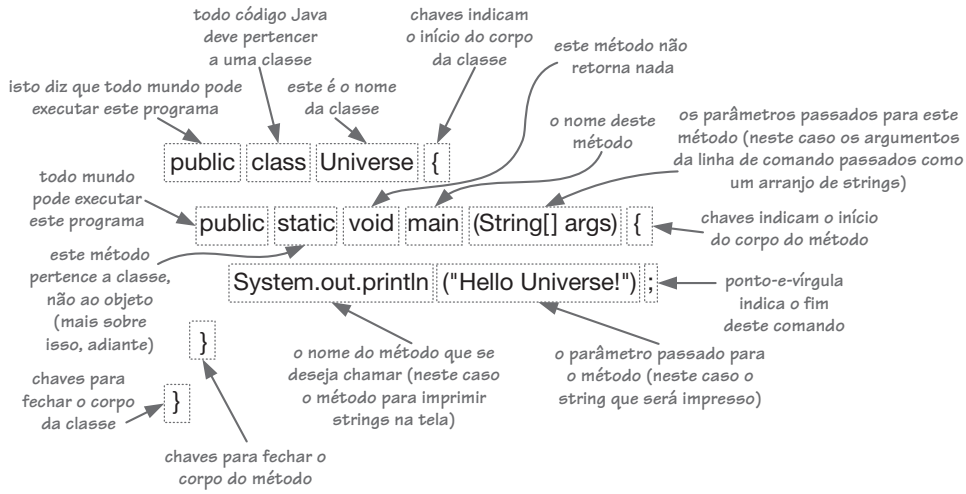


Figura 1.1 O programa "Hello Universe!"

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 consistem de 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 essa razão, objetos são também 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.

Neste 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 count. Esta variável é inicializada com zero no método construtor, Counter, que é chamado quando se deseja criar um novo objeto Counter (este método sempre tem o mesmo nome que a classe a qual pertence). Esta classe também tem um método de acesso, getCount, que retorna o valor corrente do contador. Finalmente, esta 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 aborrecida, 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 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). Listam-se as exceções a esta regra geral para identificadores Java 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. Estas 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 que seguem:

- 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 melhor 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 deste 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

Observar o uso de comentários neste e nos outros exemplos. Os comentários são anotações para uso de humanos, e não são processadas 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 tem um formato especial que permite que um programa chamado 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, na medida em que 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**. Uma 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 os mesmos a uma variável.

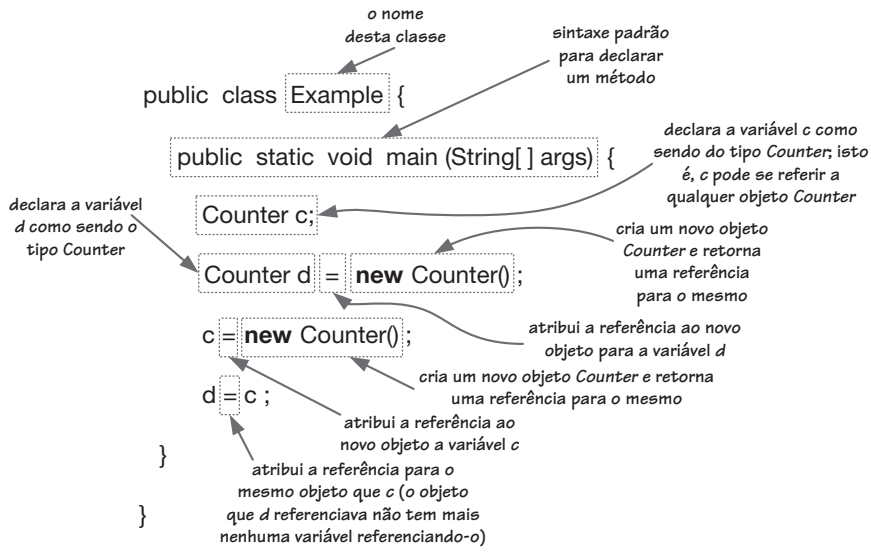


Figura 1.3 Exemplos de uso do operador **new**.

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 do 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 então 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 bá-

sico 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 5.0, 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, se admite que a variável n é declarada com o nome de classe correspondente.

Objetos string

Uma string é uma seqüência 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* (desta 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 objetos String.

Por exemplo, *P* pode ser

```
"hogs and dogs"
```

que tem comprimento 13 e pode ter vindo da página Web de alguém. Neste 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 seqüenciamento 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 de todos os caracteres de *P* seguidos por todos os caracteres de *Q*. Em Java, o operador "+" age exatamente desta 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 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***. Conseqüentemente, 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 o mesmo faça coisas ou acessar seus dados. Este conceito pode ser visto na Figura 1.4.

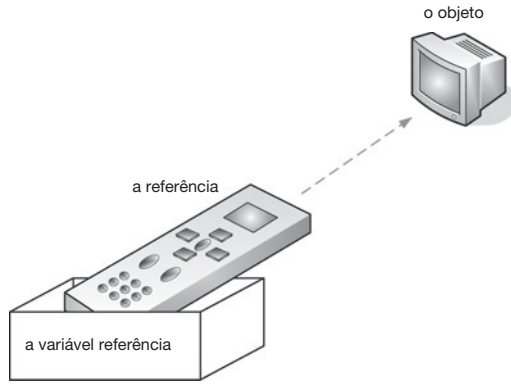


Figura 1.4 Demonstrando o relacionamento entre objetos e variáveis referencia. Quando se atribui uma referência para um objeto (isto é, um endereço de memória) para uma variável referência, é como se fosse armazenado um controle remoto do objeto naquela variável.

O operador ponto

Toda a variável referência para objeto deve referir 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 através 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. Esta situação corresponde 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. Este 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 a 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. Este acesso é feito através do 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. Opcionalmente, 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 combinem. O nome de um método combinado com a quantidade e o tipo de seus parâmetros chama-se de ***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 que 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 = "ProfessorSmythe";
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.

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 tem 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 opcio-

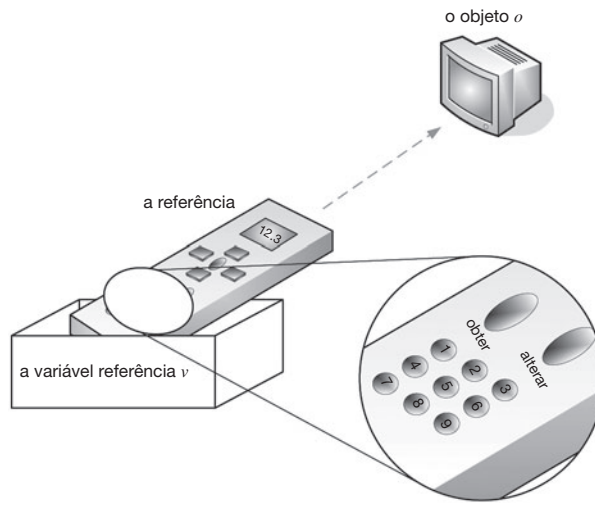


Figura 1.5 Demonstrando a maneira pela qual uma referencia para objeto pode ser usada para obter ou alterar variáveis de instância em um objeto (assumindo que se tem acesso a estas variáveis).

nal 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 através do 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

* N. de T. Gnomo.

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( ) { // Constructor 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 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**.

* N. de T. "Idade", "mágica" e "altura", respectivamente.
** N. de T. Companheiro do duende.
*** N. de T. Largura máxima.

1.1.3 Tipos enumerados

Desde a versão 5.0, Java suporta tipos enumerados chamados **enums**. Esses tipos são permitidos apenas para que se possa obter valores provenientes de conjuntos específicos de valores. Eles são declarados dentro de uma classe como segue:

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

onde *modificador* pode ser vazio, **public**, **protected** ou **private**. O nome desta enumeração, *nome,* pode ser qualquer identificador Java. Cada um dos identificadores de valor, $nome\_value_i$, 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 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 o mesmo que 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 desta 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, tais 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 declaram 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 este 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 desta 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 deste gnome.
}
```

Modificadores de métodos

À semelhança das variáveis de instância, modificadores de métodos 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 acima 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 de um método abstrato 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 que é 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 seqüência, um exemplo de método (interno à classe Gnome) que tem a forma de 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 mesmo, 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 se podem ter 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 o mesmo. 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).

Métodos 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 tipo 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 *name*, deve ser o mesmo nome da classe que constrói. Se a classe se chama Fish ("peixe"), então o construtor deve se chamar Fish da mesma forma. Além disso, um construtor não possui parâmetro de retorno – seu tipo de retorno é o mesmo que seu nome implicitamente (que é também o nome da classe). Os modificadores de construtor, indicados acima como *modifier*s, 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 um par de 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 a esta altura.

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 os mesmos fiquem em um estado inicial estável quando criados.

Métodos construtores são ativados de uma única forma: ***devem*** ser chamados através do 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, devem ser distinguíveis pelo tipo e número de parâmetros que recebem.

O método main

Certas classes Java destinam-se a ser utilizadas por outras classes, e outras têm como finalidade definir programas executáveis**. Classes que definem programas executáveis devem conter um 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
```

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

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

---

* N. de T. Meu peixe.

** 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.

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 sendo args[1] e assim por diante. (Falaremos mais 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. Neste caso, args[0] se refere à string `"45"`. Uma característica interessante do método main é que permite a cada classe definir um programa executável, e um dos usos deste método é testar os outros métodos da classe. Desta 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 seqüência 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, tal 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 do mesmo não podem mais ser referenciadas. Uma variável local pode ser tanto um ***tipo base*** (tal como **int**, **float**, **double**), como uma ***referência*** para uma instância de alguma classe. Comandos e declarações simples em Java sempre se 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 pertence à classe genérica Object por definição).
- 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, tais como 3.1415 e 10035.23, 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, como 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 caractere. Além desses, Java define as seguintes constantes especiais de caracteres:

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

- Strings: uma string é uma seqüência de caracteres entre aspas duplas, por exemplo, o que segue é um string literal

```
"cachorros não sobem em árvores"
```

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 freqüência, 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 parênteses 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 com 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. Este recurso é útil, por exemplo, para construir expressões booleanas onde primeiro se testa se uma determinada condição se aplica (tal como uma referência não ser null) e, então, se testa 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 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. Esta 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 fragmento 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 + "dollars!";
```

terá o efeito de fazer answer apontar para a string

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

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 ambigüidades 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*
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 prefixadas, que são avaliadas da direita para 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 notável 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 esta 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**. Vamos estudar a conversão implícita a seguir.

Conversores implícitos e autoboxing/unboxing

Existem casos onde o Java irá executar uma ***conversão implícita***, de acordo com o tipo da variável sendo 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 em 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 5.0, existe um novo tipo de conversão implícita entre objetos numéricos, tais como Integer e Float, e seus tipos básicos relacionados, tais como **int** e **float**. Sempre que um ob-

jeto numérico for esperado como parâmetro para um método, o tipo básico correspondente pode ser informado. Neste caso, o Java irá proceder uma conversão implícita chamada ***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 boxing 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 se duas referências para objetos apontam para o mesmo objeto. Sendo assim, quando se testa a igualdade, deve-se evitar a conversão implícita provida por autoboxing/unboxing. Por fim, a conversão implícita de qualquer tipo toma tempo, logo devemos minimizar nossa confiança nela se performance for um requisito.

Cinrcunstancialmente, existe uma situação em Java em que apenas a conversão implícita é permitida, que é 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, ao invés 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 **condicional**, comandos 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
    commando_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 if**.

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 esta 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*. Da mesma forma que o laço **for**, o corpo do laço também 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 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 a 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 que 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 ( );
    }
}
```

Neste exemplo, a variável de laço × foi declarada como **int** × = 0. Antes de cada iteração, o laço testa a condição "× < numApples" e executa o corpo do laço apenas se isso for verdadeiro. Por último, ao final de cada iteração, o laço usa a expressão ×++ para incrementar a variável × do laço antes de testar a condição novamente.

Desde a versão 5.0, Java inclui o laço *for-each*, que será discutido na Seção 6.3.2.

Laços do-while

Java tem ainda outro tipo de laço além do laço **for** e do laço **while** padrão – o laço **do-while**. Enquanto que os primeiros testam a condição antes de executar a primeira iteração com o corpo do laço, o laço **do-while** testa a condição após o corpo do laço. A sintaxe de um laço **do-while** é mostrada a seguir:

do
    corpo_do_laço
while (*condição*)

Mais uma vez, o *corpo do laço* pode ser um comando ou um bloco de comandos, e a *condição* será uma expressão booleana. Em um laço **do-while**, repete-se o corpo do laço enquanto a expressão resultar verdadeira a cada avaliação.

Suponha-se, por exemplo, que se deseja solicitar uma entrada ao usuário e posteriormente fazer algo útil com essa entrada. (Entradas e saídas em Java serão examinadas com mais detalhes

na Seção 1.6.) Uma condição possível para sair do laço, neste caso, é quando o usuário entra uma string vazia. Entretanto, mesmo neste caso, pode-se querer manter a entrada e informar ao usuário que ela saiu. O exemplo a seguir ilustra o caso:

```
public void getUserInput( ) {
    String input;
    do {
        input = getInputString( );
        handleInput(input);
    } while (input.length( )>0);
}
```

Observe-se a condição de saída do exemplo. Mais especificamente, está escrita para ser consistente com a regra de Java que diz que laços **do-while** se encerram quando a condição ***não*** é verdadeira (ao contrário da construção repeat-until usada em outras linguagens).

1.4.3 Expressões explícitas de controle de fluxo

Java também oferece comandos que permitem alterações explícitas no fluxo de controle de um programa.

Retornando de um método

Se um método Java é declarado com o tipo de retorno **void**, então o fluxo de controle retorna quando encontra a última linha de código do método ou quando encontra um comando **return** sem argumentos. Entretanto, se um método é declarado com um tipo de retorno, ele é uma função e deverá terminar retornando o valor da função como um argumento do comando **return**. O exemplo seguinte (correto) ilustra o retorno de uma função:

```
// Verifica um aniversário específico
public boolean checkBDay (int date) {
  if (date == Birthday.MIKES_BDAY){
    return true;
  }
  return false;
}
```

Conclui-se que o comando **return** ***deve*** ser o último comando executado em uma função, já que o resto do código nunca será alcançado.

Existe uma diferença significativa entre um comando ser a última linha de código a ser ***executada*** em um método ou ser a última linha de código do método propriamente dita. No exemplo anterior, a linha **return true;** claramente não é a última linha do código escrito para a função, mas pode ser a última linha executada (se a condição envolvendo date for **true**). Esse comando interrompe de forma explícita o fluxo de controle do método. Existem dois outros comandos explícitos de controle de fluxo que são usados em conjunto com laços e com o comando switch.

O comando break

O uso típico do comando **break** tem a seguinte sintaxe simples:

```
break;
```

É usado para "sair" do bloco internamente mais aninhado dos comandos **switch**, **for**, **while** ou **do-while**. Quando executado, um comando **break** faz com que o fluxo de controle seja desviado para a próxima linha depois do laço ou **switch** que contém o **break**.

O comando **break** também pode ser usado de forma rotulada para desviar para o laço ou comando **switch** de aninhamento mais externo. Neste caso, ele tem a sintaxe:

break *label*;

onde *label* é um identificador em Java usado para rotular um laço ou um comando **switch**. Este tipo de rótulo só pode aparecer no início da declaração de um laço; não existem outras formas de comandos "go to" em Java.

O uso de um rótulo com comando **break** é ilustrado no seguinte exemplo:

```
public static boolean hasZeroEntry (int[ ][ ] a) {
    boolean foundFlag = false;

  zeroSearch:
    for (int i=0; i<a.length; i++) {
        for (int j=0; j<a[i].length; j++) {
            if (a[i][j] == 0) {
              foundFlag = true;
              break zeroSearch;
            }
        }
    }
    return foundFlag;
}
```

O exemplo acima usa arranjos que serão abordados na Seção 3.1

O comando *continue*

O outro comando que altera explicitamente o fluxo de controle em um programa Java é o comando **continue**, que tem a seguinte sintaxe:

continue *label*;

onde *label* é um identificador em Java usado para rotular o laço. Como já foi mencionado anteriormente, não existem comandos "go to" explícitos em Java. Da mesma forma, o comando **continue** só pode ser usado dentro de laços (**for**, **while** e **do-while**). O comando **continue** faz com que a execução pule os passos restantes do laço na iteração atual (mas continue o laço se a condição for satisfeita).

1.5 Arranjos

Uma tarefa comum em programação é a manutenção de um conjunto numerado de objetos relacionados. Por exemplo, deseja-se que um jogo de videogame mantenha a relação das dez melhores pontuações. Em vez de se usar dez variáveis diferentes para esta tarefa, prefere-se usar um único nome para o conjunto e usar índices numéricos para referenciar as pontuações mais altas dentro do conjunto. Da mesma forma, deseja-se que um sistema de informações médicas mantenha a relação de pacientes associados aos leitos de um certo hospital. Novamente, não é necessário inserir 200 variáveis no programa apenas porque o hospital tem 200 leitos.

Nestes casos, minimiza-se o esforço de programação pelo uso de **arranjos**, que são coleções numeradas de variáveis do mesmo tipo. Cada variável ou **célula** em um arranjo tem um **índice**, que referencia o valor armazenado na célula de forma única. As células de um arranjo *a* são numeradas 0, 1, 2 e assim por diante. A Figura 1.6 apresenta o desenho de um arranjo contendo as melhores pontuações do videogame.

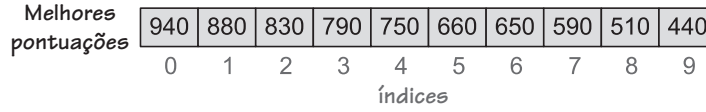


Figura 1.6 Desenho de um arranjo com as dez (**int**) melhores pontuações de um videogame.

Essa forma de organização é extremamente útil, na medida em que permite computações interessantes. Por exemplo, o método a seguir soma todos os valores armazenados em um arranjo de inteiros:

```
/** Soma todos os valores de um arranjo de inteiros */
public static int sum(in [ ] a) {
int total = 0;
for (int i=0; i<a.length; i++) // observe-se o uso da variável length
  total += a[i];
return total;
}
```

Este exemplo tira vantagem de um recurso interessante de Java que permite determinar a quantidade de células mantidas por um arranjo, ou seja, seu ***tamanho****. Em Java um arranjo *a* é um tipo especial de objeto, e o tamanho de *a* está armazenado na variável de instância length. Isto é, jamais será necessário ter de adivinhar o tamanho de um arranjo em Java, visto que o tamanho de um arranjo pode ser acessado como segue:

nome_do_arranjo.length

onde *nome_do_arranjo* é o nome do arranjo. Assim, as células de um arranjo *a* são numeradas 0, 1, 2, e assim por diante até *a*.length – 1.

Elementos e capacidade de um arranjo

Cada objeto armazenado em um arranjo é chamado de ***elemento*** do arranjo. O elemento número 0 é *a*[0], o elemento número 1 é *a*[1], o elemento número 2 é *a*[2], e assim por diante. Uma vez que o comprimento de um arranjo determina o número máximo de coisas que podem ser armazenadas no arranjo, também pode-se referir ao comprimento de um arranjo como sendo sua ***capacidade***. O trecho de código que segue apresenta outro exemplo simples de uso de arranjos, que conta o número de vezes que um certo número aparece em um arranjo.

```
/** Conta o número de vezes que um inteiro aparece em um arranjo */
public static int findCount(int[ ] a, int k) {
  int count = 0;
  for (int e: a) {      // observe-se o uso do laço "foreach"
    if (e == k)         //deve-se  verificar se o elemento corrente é igua a k
      count++;
  }
  return count;
}
```

Erros de limites

É um erro perigoso tentar indexar um arranjo *a* usando um número fora do intervalo de 0 a *a*.length – 1. Tal referência é dita estar ***fora de faixa***. Referências fora de faixa tem sido fre-

* N. de T. Apesar da expressão em inglês *length* indicar comprimento, é mais comum entre programadores Java o uso da expressão *tamanho* do arranjo.

qüentemente exploradas por hackers usando um método chamado ***ataque do estouro do buffer****
comprometendo a segurança de sistemas de computação escritos em outras linguagens em vez de Java. Por questões de segurança, os índices de arranjo são sempre verificados em Java para constatar se não estão fora de faixa. Se um índice de arranjo está fora de faixa, o ambiente de execução de Java sinaliza uma condição de erro. O nome desta condição é ArrayIndexOutOfBoundsExceptions. Esta verificação auxilia para que Java evite uma série de problemas (incluindo problemas de ataque de estouro de buffer) com os quais outras linguagens tem de lutar.

Pode-se evitar erros de índice fora de faixa tendo certeza de que as indexações sempre serão feitas dentro de um arranjo *a*, usando valores inteiros entre 0 e *a*.length. Uma forma simples de fazer isso é usando com cuidado o recurso das operações booleanas de Java já apresentado. Por exemplo, um comando como o que segue nunca irá gerar um erro de índice fora de faixa:

```
if ((i >= 0) && (i < a.length) && (a[i] > 2) )
  x = a[i];
```

pois a comparação "a[i] > 5" só será executada se as duas primeiras comparações forem bem-sucedidas.

1.5.1 Declarando arranjos

Uma forma de declarar e inicializar um arranjo é a seguinte:

tipo_do_elemento[] *nome_do_arranjo* = { *val_inic_0*, *val_inic_1*, ..., *val_inic_N-1*};

O *tipo_do_elemento* pode ser qualquer tipo base de Java ou um nome de classe e *nome_do_arranjo* pode ser qualquer identificador Java válido. Os valores de inicialização devem ser do mesmo tipo que o arranjo. Por exemplo, considere-se a seguinte declaração de um arranjo que é inicializado para conter os primeiros dez números primos:

```
int[ ] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Além de declarar um arranjo e inicializar todos os seus valores na declaração, pode-se declarar um arranjo sem inicializá-lo. A forma desta declaração é a que segue:

tipo_do_elemento[] *nome_do_arranjo*;

Um arranjo criado desta forma é inicializado com zeros se o tipo do arranjo for um tipo numérico. Arranjos de objetos são inicializados com referências **null**. Uma vez criado um arranjo desta forma, pode-se criar o conjunto de células mais tarde usando a sintaxe a seguir:

new *tipo_do_elemento*[*comprimento*]

onde *comprimento* é um inteiro positivo que denota o comprimento do arranjo criado. Normalmente, esta expressão aparece em comandos de atribuição com o nome do arranjo do lado esquerdo do operador de atribuição. Então, por exemplo, o seguinte comando define uma variável arranjo chamada *a* e, mais tarde, atribuem-se à mesma um arranjo de dez células, cada uma do tipo **double**, que então é inicializado:

```
double[ ] a;

// ... vários passos ...

a = new double[10];
for (int k=0; k < a.length; k++) {
  a[k] = 1.0;
}
```

* N. de T. Em inglês, *buffer overflow attack*.

As células do novo arranjo "a" são indexadas usando o conjunto inteiro {0,1,2, ..., 9} (lembre-se que os arranjos em Java sempre iniciam a indexação em 0), e, da mesma forma que qualquer arranjo Java, todas as células deste arranjo são do mesmo tipo – **double**.

1.5.2 Arranjos são objetos

Arranjos em Java são tipos especiais de objetos. Na verdade, esta é a razão pela qual pode-se usar o operador **new** para criar uma nova instância de arranjo. Um arranjo pode ser usado da mesma forma que qualquer outro objeto de Java, mas existe uma sintaxe especial (usando colchetes, "[" e "]") para se referenciar a seus membros. Um arranjo Java pode fazer tudo que um objeto genérico pode fazer. Como se trata de um objeto, o nome de um arranjo em Java é, na verdade, uma referência para o lugar na memória onde o arranjo está armazenado. Assim, não existe nada de tão especial em se usar o operador ponto e a variável de instância length, para se referir ao comprimento de um arranjo como no exemplo "a.length". O nome *a*, neste caso, é apenas uma referência ou ponteiro para o arranjo subjacente.

O fato de que arranjos em Java são objetos tem uma implicação importante quando se usa nomes de arranjos em expressões de atribuição. Quando se escreve algo tal como

```
b = a;
```

em um programa Java, na verdade significa que agora tanto b como a se referem ao mesmo arranjo. Então, ao se escrever algo como

```
b[3] = 5;
```

se está alterando a[3] para 5. Este ponto crucial é demonstrado na Figura 1.7.

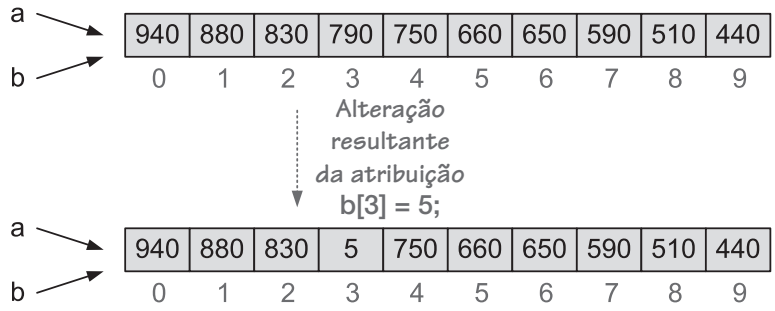


Figura 1.7 Desenho da atribuição de um arranjo de objetos. Apresenta-se o resultado da atribuição de "b[3] = 5;" depois de previamente ter executado "b = a".

Clonando um arranjo

Se, por outro lado, for necessário criar uma cópia exata do arranjo a e atribuir esse arranjo para a variável arranjo b, pode-se escrever:

```
b = a.clone( );
```

que copia todas as células em um novo arranjo e o atribui a b, de maneira que este aponta para o novo arranjo. Na verdade, o método clone é um método predefinido de todo objeto Java, e cria uma cópia exata de um objeto. Neste caso, se então escrever-se

```
b[3] = 5;
```

o novo arranjo (copiado) terá o valor 5 atribuído para a célula de índice 3, mas a[3] irá permanecer inalterado. Demonstra-se esta situação na Figura 1.8.

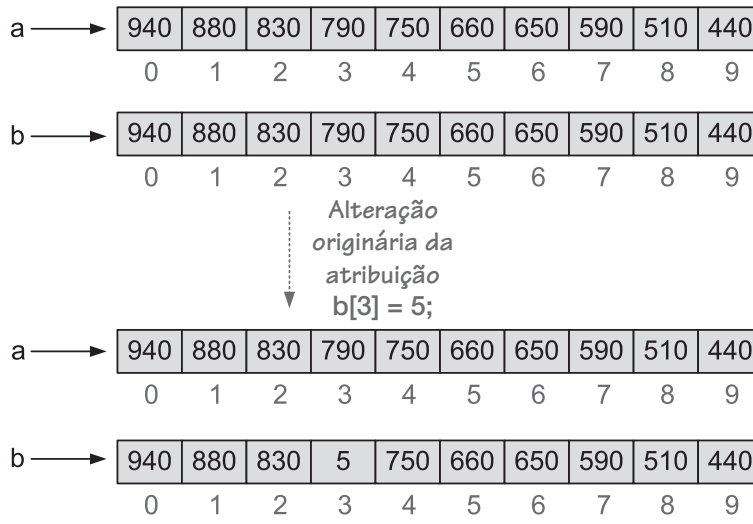


Figura 1.8 Demonstração da clonagem de um arranjo de objetos. Demonstra-se o resultado da atribuição b[3] = 5 após a atribuição "b = a.clone( );".

Detalhando, pode-se afirmar que as células de um arranjo são copiadas quando o mesmo é clonado. Se as células são de um tipo base, como **int**, os valores são copiados. Mas se as células são referências para objetos, então essas referências são copiadas. Isso significa que existem duas maneiras de referenciar tais objetos. As conseqüências deste fato são exploradas no Exercício R-1.1.

1.6 Entrada e saída simples

Java oferece um conjunto rico de classes e métodos para executar entrada e saída. Existem classes para executar projetos de interfaces gráficas com o usuário, incluindo diálogos e menus suspensos, assim como métodos para a exibição e a entrada de texto e números. Java também oferece métodos para lidar com objetos gráficos, imagens, sons, páginas Web e eventos de mouse (tais como cliques, deslocamentos do mouse e arrasto). Além do mais, muitos desses métodos de entrada e saída podem ser usados tanto em programas executáveis como em applets. Infelizmente, detalhar como cada um desses métodos funciona para construir interfaces gráficas sofisticadas com o usuário está além do escopo deste livro. Entretanto, em nome de uma maior abrangência, descreve-se nesta seção como se pode fazer entrada e saída simples em Java.

Em Java, a entrada e saída simples é feita através da janela console de Java. Dependendo do ambiente Java que se está empregando, esta janela pode ser a janela especial usada para exibição e entrada de texto, ou é a janela que se utiliza para passar comandos para nosso sistema operacional (tais janelas costumam ser chamadas de janelas de console, janelas DOS ou janelas de terminal).

Métodos de saída

Java oferece um objeto static embutido chamado System.out, que envia a saída para o dispositivo de saída-padrão. Alguns sistemas operacionais permitem aos usuários redirecionar a saída-padrão para arquivos ou até mesmo como entrada para outros programas, embora a saída-padrão seja a janela de console de Java. O objeto System.out é uma instância da classe java.io.PrintStream. Essa classe define métodos para um fluxo buferizado de saída, o que significa que os caracteres são colocados em uma localização temporária, chamada ***buffer***, que é esvaziada quando a janela de console estiver pronta para imprimir os caracteres.

Mais especificamente, a classe java.io.PrintStream fornece os seguintes métodos para executar saídas simples (usamos *base_type* para indicar qualquer um dos possíveis tipos básicos):

print(Object *o*): imprime o objeto *o* usando seu método toString;
print(String *s*): imprime a string *s*;
print(*base_type b*): imprime o valor de *b* conforme seu tipo básico;
println(String *s*): imprime a string *s*, seguida pelo caractere de nova linha.

Um exemplo de saída

Considere, por exemplo, o seguinte trecho de código:

```
System.out.print("Java values: ");
System.out.print(3.1415);
System.out.print(',');
System.out.print(15);
System.out.println(" (double,char,int).");
```

Quando executado, este trecho de código produz a seguinte saída na janela console de Java:

```
Java values: 3.1415,15 (double,char,int).
```

Entrada simples usando a classe java.util.Scanner

Assim como existe um objeto especial para enviar a saída para a janela de console de Java, existe também um objeto especial, chamado System.in, para executar a entrada de dados a partir da janela de console de Java. Tecnicamente, a entrada vem, na verdade, do "dispositivo de entrada-padrão", o qual, por default, é o teclado do computador ecoando os caracteres na janela de console de Java. O objeto System.in é um objeto associado com o dispositivo de entrada padrão. Uma maneira simples de ler a entrada usando este objeto é utilizá-lo para criar um objeto Scanner, através da expressão:

```
new Scanner(System.in)
```

A classe Scanner inclui uma série de métodos convenientes para ler do fluxo de entrada. Por exemplo, o programa que segue usa um objeto Scanner para processar a entrada:

```
import java.io.*;
import java.util.Scanner;
public class InputExample {
  public static void main(Stringargs[ ]) throws IOException {
    Scanner s = new Scanner(System.in);
    System.out.print("Enteryourheightincentimeters: ");
    float height = s.nextFloat( );
    System.out.print("Enteryourweightinkilograms: ");
    float weight = s.nextFloat( );
    float bmi = weight/(height*height)*10000;
    System.out.println("Yourbodymassindexis" + bmi + ".");
  }
}
```

Quando executado, este programa gera o seguinte no console de Java:

```
Enter your height in centimeters: 180
Enter your weight in kilograms: 80.5
Your body mass index is 24.84568.
```

Métodos de java.util.Scanner

A classe Scanner lê o fluxo de entrada e divide o mesmo em ***tokens***, que são strings de caracteres contíguos separados por ***delimitadores***, que correspondem a caracteres separadores. O delimitador padrão é o espaço em branco, ou seja, tokens são separados por strings de espaços, tabulações ou nova-linha, por default. Tokens tanto podem ser lidos imediatamente como strings ou um objeto Scanner pode converter um token para um tipo base, se o token estiver sintaticamente correto. Para tanto, a classe Scanner inclui os seguintes métodos para lidar com tokens:

hasNext( ): retorna **true** se e somente se existe mais um token no strings de entrada.

next( ): retorna o próximo token do fluxo de entrada; gera um erro se não existem mais tokens.

hasNext*Type*(*Tipo*): retorna **true** se e somente se existe mais um token no fluxo de entrada e se pode ser interpretado como sendo do tipo base correspondente, *Tipo*, onde *Tipo* pode ser Boolean, Byte, Double, Float, Int, Long ou Short.

next*Type*(*Tipo*): retorna o próximo token do fluxo de entrada, retornando-o com o tipo base correspondente a *Tipo*; gera uma erro se não existem mais tokens ou se o próximo token não pode ser interpretado como sendo do tipo base correspondente a *Tipo*.

Além disso, objetos Scanner podem processar a entrada linha por linha, ignorando os delimitadores e mesmo podem procurar por padrões em linhas. Os métodos para processar a entrada desta forma incluem os seguintes:

hasNextLine( ): retorna **true** se e somente se o fluxo de entrada tem outra linha de texto.

nextLine( ): avança a entrada até o final da linha corrente e retorna toda a entrada que foi deixada para trás.

findInLine(String s): procura encontrar um string que combine com o padrão (expressão regular) s na linha corrente. Se o padrão for encontrado, ele é retornado e o scanner avança para o primeiro caractere depois do padrão. Se o padrão não for encontrado, o scanner retorna **null** e não avança.

Esses métodos podem ser usados com os anteriores, como no exemplo que segue:

```
Scanner input = new Scanner(System.in);
System.out.print("Please enter an integer: ");
while (!input.hasNextInt( )) {
  input.nextLine( );
  System.out.print("That's not an integer; please enter an integer: ");
}
int i = input.nextInt( );
```

1.7 Um programa de exemplo

Nesta seção, será descrito um exemplo simples de programa Java que ilustra muitas das construções definidas anteriormente. O exemplo consiste em duas classes: CreditCard, que define objetos que representam cartões de crédito; e Test, que testa as funcionalidades da classe CreditCard. Os objetos que representam cartões de crédito, definidos pela classe CreditCard, são versões simplificadas dos cartões de crédito tradicionais. Eles têm um número de identificação, informações de identificação do proprietário e do banco que os emitiram e informações sobre o saldo corrente e o limite de crédito. Não debitam juros ou pagamentos atrasados, mas restringem pagamentos que possam fazer com que o saldo vá além do limite de gastos.

A classe CreditCard

A classe CreditCard é apresentada no Trecho de código 1.5. Ela define cinco variáveis de instância, todas exclusivas, e possui um construtor simples que inicializa essas variáveis.

Ela também define cinco ***métodos de acesso*** que permitem acessar o valor corrente dessas variáveis de instância. Evidentemente, as variáveis de instância poderiam ter sido definidas como públicas, o que faria com que os métodos de acesso fossem duvidosos. A desvantagem dessa abordagem direta, porém, é que permite ao usuário modificar as variáveis de instância do objeto diretamente, enquanto que, em muitos casos como este, é preferível restringir a alteração de variáveis de instância a métodos especiais chamados de ***métodos de atualização***. No Trecho de código 1.5, inclui-se dois métodos de atualização, chargeIt e makePayment.

Além disso, é conveniente incluir ***métodos de ação***, que com freqüência definem as ações específicas do comportamento do objeto. Para demonstrar isso, define-se um método de ação, o printCard, como um método estático, que também está incluído no Trecho de código 1.5.

A classe test

A classe CreditCard é testada na classe Test. Observa-se aqui o uso de um arranjo de objetos CreditCard, wallet, e como se usam iterações para fazer débitos e pagamentos. Apresenta-se o código completo da classe Test no Trecho de código 1.6. Para simplificar, a classe Test não produz nenhum gráfico elaborado, simplesmente envia a saída para o console de Java. Apresenta-se esta saída no Trecho de código 1.7. Observa-se a diferença na maneira em que se utilizam os métodos não-estáticos chargeIt e makePayment e o método estático printCard.

```
public class CreditCard {
    // Variáveis de instância:
  private String number;
  private String name;
  private String bank;
  private double balance;
  private int limit;
    // Construtor:
  CreditCard(String no, String nm, String bk, double bal, int lim) {
    number = no;
    name = nm;
    bank = bk;
    balance = bal;
    limit = lim;
  }
  // Métodos de acesso:
  public String getNumber( ) { return number; }
  public String getName( ) { return name; }
  public String getBank( ) { return bank; }
  public double getBalance( ) { return balance; }
  public int getLimit( ) { return limit; }
  // Métodos de ação:
  public boolean chargeIt(double price) { // Debita
    if (price + balance > (double) limit)
      return false; // Não há dinheiro suficiente para debitar
    balance += price;
    return true; // Neste caso o débito foi efetivado
  }
```

```
  public void makePayment(double payment) { // Faz um pagamento
    balance –= payment;
  }
  public static void printCard(CreditCard c) { // Imprime informações sobre o cartão
    System.out.println("Number = " + c.getNumber( ));
    System.out.println("Name = " + c.getName( ));
    System.out.println("Bank = " + c.getBank( ));
    System.out.println("Balance = " + c.getBalance( )); // conversão implícita
    System.out.println("Limit = " + c.getLimit( )); // conversão implícita
  }
}
```

Trecho de código 1.5 A classe CreditCard.

```
public class Test {
  public static void main(String[ ] args) {
    CreditCard wallet[ ] = new CreditCard[10];
    wallet[0] = new CreditCard("5391 0375 9387 5309",
                    "John Bowman", "California Savings", 0.0, 2500);
    wallet[1] = new CreditCard("3485 0399 3395 1954",
                    "John Bowman", "California Federal", 0.0, 3500);
    wallet[2] = new CreditCard("6011 4902 3294 2994",
                    "John Bowman", "California Finance", 0.0, 5000);
    for (int i=1; i<=16; i++) {
      wallet[0].chargeIt((double)i);
      wallet[1].chargeIt(2.0*i); // conversão implícita
      wallet[2].chargeIt((double)3*i); // conversão explícita
    }
    for (int i=0; i<3; i++) {
      CreditCard.printCard(wallet[i]);
      while (wallet[i].getBalance( ) > 100.0) {
        wallet[i].makePayment(100.0);
        System.out.println("New balance = " + wallet[i].getBalance( ));
      }
    }
  }
}
```

Trecho de código 1.6 A classe Test.

```
Number = 5391 0375 9387 5309
Name = John Bowman
Bank = California Savings
Balance = 136.0
Limit = 2500
New balance = 36.0
Number = 3485 0399 3395 1954
Name = John Bowman
Bank = California Federal
Balance = 272.0
Limit = 3500
New balance = 172.0
New balance = 72.0
```

```
Number = 6011 4902 3294 2994
Name = John Bowman
Bank = California Finance
Balance = 408.0
Limit = 5000
New balance = 308.0
New balance = 208.0
New balance = 108.0
New balance = 8.0
```

Trecho de código 1.7 Saída da classe Test.

1.8 Classes aninhadas e pacotes

A linguagem Java usa uma abordagem prática e genérica para organizar as classes de um programa. Toda a classe pública definida em Java deve ser fornecida em um arquivo separado. O nome do arquivo é o nome da classe com uma terminação ***.java***. Desta forma, a classe **public class** SmartBoard, é definida em um arquivo chamado ***SmartBoard.java***. Nesta seção, são apresentadas duas maneiras interessantes pelas quais Java permite que várias classes sejam organizadas.

Classes aninhadas

Java permite que definições de classes sejam feitas dentro, isto é, ***aninhadas*** dentro das definições de outras classes. Este é um tipo de construção útil que será explorada diversas vezes neste livro na implementação de estruturas de dados. O uso principal de classes aninhadas é para definir uma classe fortemente conectada com outra. Por exemplo, a classe de um editor de textos pode definir uma classe cursor relacionada. Definindo a classe cursor como classe aninhada dentro da definição da classe editor, mantém-se a definição destas duas classes altamente relacionadas juntas no mesmo arquivo. Além disso, permite que ambas acessem os métodos públicos uma da outra. Um aspecto técnico relacionado a classes aninhadas é que classes aninhadas podem ser declaradas como **static**. Esta declaração implica que a classe aninhada está associada com a classe mais externa, mas não com uma instância da classe mais externa, isto é, um objeto específico.

Pacotes

Um conjunto de classes relacionadas, todas pertencentes ao mesmo subdiretório, pode ser um **package** (pacote) Java. Cada arquivo em um pacote se inicia com a linha:

package *nome_do_pacote*;

O subdiretório que contém o pacote deve ter o mesmo nome que o pacote. É possível, também, definir um pacote em um único arquivo que contenha diversas definições de classe, mas quando for compilado, todas as classes o serão em arquivos separados no mesmo subdiretório.

Em Java, pode-se usar classes que estão definidas em outros pacotes prefixando os nomes das classes com pontos (isto é, usando o caractere ".") que corresponde à estrutura de diretório dos outros pacotes.

```
public boolean Temperature(TA.Measures.Thermometer thermometer,
                           int temperature) {
  // ...
  }
```

A função Temperature recebe a classe Thermometer como parâmetro. Thermometer é definida no pacote TA, em um subpacote chamado Measures. Os pontos em TA.Measures.Thermometer têm correspondência direta com a estrutura de diretório do pacote TA.

Toda a digitação necessária para fazer referência a uma classe fora do pacote corrente pode-se tornar cansativa. Em Java, é possível usar a palavra reservada **import** para incluir classes externas ou pacotes inteiros no arquivo corrente. Para importar uma classe individual de um pacote específico, digita-se no início do arquivo o seguinte:

import *nome_do_pacote.nome_da_classe*;

Por exemplo, pode-se digitar

```
package Project;
import TA.Measures.Thermometer;
import TA.Measures.Scale;
```

no início do pacote Project para indicar que se está importando as classes TA.Measures.Thermometer e TA.Measures.Scale. O ambiente de execução de Java irá procurar essas classes para verificar os identificadores com as classes, métodos e variáveis de instância que se usa no programa.

Também se pode importar um pacote inteiro utilizando a seguinte sintaxe:

import ⟨packageName⟩.*;

Por exemplo:

```
package student;
import TA.Measures.*;
public boolean Temperature(Thermometer thermometer, int temperature) {
    // ...
    }
```

Nos casos em que dois pacotes têm classes com o mesmo nome, ***deve-se*** referenciar especificamente o pacote que contém a classe. Por exemplo, supondo que ambos os pacotes Gnomes e Cooking tenham uma classe chamada Mushroom (“cogumelo”). Se for determinado um comando **import** para cada pacote, deve-se especificar que classe se quer designar:

```
Gnomes.Mushroom shroom = new Gnomes.Mushroom ("purple");
Cooking.Mushroom topping = new Cooking.Mushroom ( );
```

Se não for especificado o pacote (ou seja, no exemplo anterior apenas foi empregada uma variável do tipo Mushroom), o compilador irá sinalizar um erro de “classe ambígua”.

Resumindo a estrutura de um programa Java, pode-se ter variáveis de instância e métodos dentro de uma classe, além de classes dentro de um pacote.

1.9 Escrevendo um programa em Java

O processo de escrever um programa em Java envolve três etapas fundamentais:

1. projeto,
2. codificação,
3. teste e depuração.

Cada uma será resumida nesta seção.

1.9.1 Projeto

A etapa de projeto é talvez o passo mais importante no processo de escrever um programa. É na fase de projeto que se decide como dividir as tarefas do programa em classes, como essas classes irão interagir com os dados que irão armazenar e que ações cada uma irá executar. Um dos maiores desafios com o qual os programadores iniciantes se deparam em Java é determinar que classes definir para executar as tarefas do seu programa. Ainda que seja difícil obter prescrições genéricas, existem algumas regras práticas que podem ser aplicadas quando se está procurando definir as classes:

- ***Responsabilidades*:** dividir o trabalho entre diferentes ***atores***, cada um com responsabilidades diferentes. Procurar descrever as responsabilidades usando verbos de ação. Os atores irão formar as classes do programa.
- ***Independência*:** definir, se possível, o trabalho de cada classe de forma independente das outras classes. Subdividir as responsabilidades entre as classes de maneira que cada uma tenha autonomia sobre algum aspecto do programa. Fornecer os dados (como variáveis de instância) para as classes que têm competência sobre as ações que requerem acesso a esses dados.
- ***Comportamento*:** as conseqüências de cada ação executada por uma classe terão de ser bem compreendidas pelas classes que interagem com ela. Portanto, é preciso definir o comportamento de cada uma com cuidado e precisão. Esses comportamentos irão definir os métodos que a classe executa. O conjunto de comportamentos de uma classe é algumas vezes chamado de ***protocolo***, porque espera-se que os comportamentos de uma classe sejam agrupados como uma unidade coesa.

A definição das classes juntamente com seus métodos e variáveis de instância determina o projeto de um programa Java. Um bom programador, com o tempo, vai desenvolver naturalmente grande habilidade em executar essas tarefas à medida que a experiência lhe ensinar a observar padrões nos requisitos de um programa que se parecem com padrões já vistos.

1.9.2 Pseudocódigo

Freqüentemente, programadores são solicitados a descrever algoritmos de uma maneira que seja compreensível para olhos humanos, em vez de escrever um código real. Tais descrições são chamadas de ***pseudocódigo***. Pseudo-código não é um programa de computador, mas é mais estruturado que a prosa normal. Pseudo-código é uma mistura de língua-natural com estruturas de programação de alto-nível que descrevem as idéias principais que estão por trás da implementação de uma estrutura de dados ou algoritmo. Não existe, portanto, uma definição precisa de uma linguagem de ***pseudocódigo***, em razão de sua dependência da língua natural. Ao mesmo tempo, para auxiliar na clareza, o pseudo-código mistura língua natural com construções de padrão de linguagens de programação. As construções que foram escolhidas são consistentes com as modernas linguagens de alto nível, tais como C, C++ e Java.

Estas construções incluem:

- ***Expressões*:** usam-se símbolos matemáticos padrão para expressões numéricas e booleanas. Usa-se a seta para esquerda (←) como operador de atribuição em comandos de atribuição (equivalente ao operador = de Java) e o sinal de igual (=) para o relacional de igualdade em expressões booleanas (equivalente ao relacional "==" em Java).
- ***Declarações de métodos*: Algoritmo** nome(*param*1, *param*2, ...) declara um nome de método novo e seus parâmetros.

- ***Estruturas de decisão*: se** condição, **então** ações caso verdade [**senão**, ações caso falso]. Usa-se identação para indicar quais ações devem ser incluídas nas ações caso verdade e nas ações caso falso.
- ***Laços enquanto*: enquanto** condição, **faça** ações. Usa-se indentação para indicar quais ações devem ser incluídas no laço.
- ***Laços repete*: repete** ações **até** condição. Usa-se indentação para indicar que ações devem ser incluídas no laço.
- ***Laços for*: for** definição de variável de incremento, **faça** ações. Usa-se identação para indicar que ações devem ser incluídas no laço.
- ***Indexação de arranjos*:** A[*i*] representa a *i*-ésima célula do arranjo *A*. As células de um arranjo *A* com *n* células são indexadas de *A*[0] até *A*[*n* – 1] (de forma consistente com Java).
- ***Chamadas de métodos*:** objeto.método(argumentos) (objeto é opcional se for subentendido).
- ***Retorno de métodos*: retorn** valor. Esta operação retorna o valor especificado para o método que chamou o método corrente.
- ***Comentários*:** { os comentários vão aqui }. Os comentários são colocados entre chaves.

Quando se escreve pseudo-código, deve-se ter em mente que se está escrevendo para um leitor humano, não para um computador. Assim, o esforço deve ser no sentido de comunicar idéias de alto-nível, e não detalhes de implementação. Ao mesmo tempo, não se deve omitir passos importantes. Como em muitas outras formas de comunicação humana, encontrar o balanceamento correto é uma habilidade importante, que é refinada pela prática.

1.9.3 Codificação

Como mencionado anteriormente, um dos passos-chave na codificação de um programa orientado a objetos é codificar a partir de descrições de classes e seus respectivos métodos. Para acelerar o desenvolvimento dessa habilidade, serão estudados, em diferentes momentos ao longo deste texto, vários ***padrões de projeto*** para os projetos de programas orientados a objeto (ver Seção 2.1.3). Esses padrões fornecem moldes para a definição de classes e interações entre essas classes.

Muitos programadores não fazem seus projetos iniciais em um computador, e sim usando ***cartões CRC***. *Componente-responsabilidade-colaborador*, ou CRC, são simples cartões indexados que subdividem as tarefas especificadas para um programa. A idéia principal por trás desta ferramenta é que cada cartão represente um componente, o qual, no final, se transformará em uma classe de nosso programa. Escreve-se o nome do componente no alto do cartão. No lado esquerdo do mesmo, anotam-se as responsabilidades desse componente. No lado direito, listam-se os colaboradores desse componente, isto é, os outros componentes com os quais o primeiro terá de interagir de maneira a cumprir suas finalidades. O processo de projeto é iterativo através de um ciclo ação/ator, onde primeiro se identifica uma ação (ou seja, uma responsabilidade) e, então, se determina o ator (ou seja, um componente) mais adequado para executar tal ação. O processo de projeto está completo quando se tiver associado atores para todas as ações.

A propósito, ao utilizar cartões indexados para executar nosso projeto, assume-se que cada componente terá um pequeno conjunto de responsabilidades e colaboradores. Esta premissa não é acidental, uma vez que auxilia a manter os programas gerenciáveis.

Uma alternativa ao uso dos cartões CRC é o uso de diagramas UML (Linguagem de Modelagem Unificada*) para expressar a organização de um programa e pseudo-código para expressar algoritmos. Diagramas UML são uma notação visual padrão para expressar projetos orientados a

* N. de T. Unified Modeling Language.

objetos. Existem muitas ferramentas auxiliadas por computador capazes de construir diagramas UML. A descrição de algoritmos em pseudo-código, por outro lado, é uma técnica que será utilizada ao longo deste livro.

Tendo decidido sobre as classes de nosso programa, juntamente com suas responsabilidades, pode-se começar a codificação. Cria-se o código propriamente dito das classes do programa usando tanto um editor de textos independente (por exemplo emacs, WordPad ou vi) como um editor embutido em um ***ambiente integrado de desenvolvimento*** (IDE*), tal como o Eclipse ou o Borland JBuilder.

Após completar a codificação de uma classe (ou pacote), se compila o arquivo para código executável usando um compilador. Quando não se está usando um IDE, então se compila o programa chamando um programa tal como `javac` sobre o arquivo. Estando em uso um IDE, então se compila o programa clicando o botão apropriado. Felizmente se o programa não tiver erros de sintaxe, então o processo de compilação irá criar arquivos com a extensão "`.class`".

Se o programa contiver erros de sintaxe, estes serão identificados, e se terá de voltar ao editor de textos para consertar as linhas de código com problema. Eliminados todos os erros de sintaxe e criado o código compilado correspondente, pode-se executar o programa tanto chamando um comando, tal como "`java`" (fora de um IDE), ou clicando o botão de execução apropriado (dentro de um IDE). Quando um programa Java estiver executando dessa forma, o ambiente de execução localiza os diretórios contendo as classes criadas e quaisquer outras referenciadas a partir destas, usando uma variável especial de ambiente do sistema operacional. Essa variável é chamada de "CLASSPATH", e a ordem dos diretórios a serem pesquisados é fornecida como uma lista de diretórios, separados por vírgulas se em Unix/Linux ou por ponto-e-vírgulas se em DOS/Windows. Um exemplo de atribuição para a variável CLASSPATH no sistema operacional DOS/Windows pode ser o seguinte:

```
SET CLASSPATH=.;C:\java;C:\Program Files\Java\
```

Um exemplo de atribuição para CLASSPATH no sistema operacional Unix/Linux pode ser:

```
setenv CLASSPATH ".:/usr/local/java/lib:/usr/netscape/classes"
```

Em ambos os casos, o ponto (".") se refere ao diretório atual a partir do qual o ambiente de execução foi chamado.

Javadoc

Para incentivar o bom uso de comentários em bloco e a produção automática de documentação, o ambiente de programação Java vem com um programa para a geração de documentação chamado ***javadoc***. Esse programa examina uma coleção de arquivos fontes Java que tenham sido comentados usando-se certas palavras reservadas, chamadas de ***tags***, e produz uma série de documentos HTML que descrevem as classes, métodos, variáveis e constantes contidas nestes arquivos. Por razões de espaço, não se usa o estilo de comentários do javadocs em todos os programas exemplificadores contidos neste livro, mas se incluiu um exemplo de javadoc no Trecho de código 1.8, bem como em outros disponíveis no site da Web que acompanha este livro.

Cada comentário javadoc é um comentário em bloco que se inicia com "/**", termina com "*/" e tem cada linha entre estas duas iniciada por um único asterisco, "*" que é ignorado. Pressupõe-se que o bloco de comentário deve começar com uma frase descritiva seguida por uma linha em branco, e posteriormente por linhas especiais que começam por tags javadoc. Um comentário em bloco que venha imediatamente antes de uma definição de classe, uma declaração de variável ou uma definição de método é processado pelo javadoc em um comentário que se refere à classe, à variável ou ao método.

* N. de T. A abreviatura, já consagrada, corresponde à expressão em inglês *integrated development environment*.

```
/**
 * Esta classe define um ponto (x,y) não alterável no plano
 *
 * @author Michael Goodrich
 */
public class XYPoint {
  private double x,y; // variáveis de instância privada para as coordenadas

  /**
   * Constrói um ponto (x,y) em uma localização específica
   *
   * @param xCoor A abscissa do ponto
   * @param yCoor A ordenada do ponto
   */

  public XYPoint(double xCoor, double yCoor) {
    x = xCoor;
    y = yCoor;
  }

  /**
   * Retorna o valor da abscissa
   *
   * @return abscissa
   */
  public double getX( ) { return x; }

  /**
   * Retorna o valor da ordenada
   *
   * @return ordenada
   */
  public double getY( ) { return x; }
}
```

Trecho de código 1.8 Um exemplo de definição de classe usando o estilo javadoc de comentário. Observa-se que esta classe inclui apenas duas variáveis de instância, um construtor e dois métodos de acesso.

As tags javadoc mais importantes são as seguintes:

- `@author` *texto*: identifica os autores (um por linha) de uma classe;
- `@exception` *descrição do nome de uma exceção*: identifica uma condição de erro sinalizada por este método (ver Seção 2.3);
- `@param` *descrição de um nome de parâmetro*: identifica um parâmetro aceito por este método;
- `@return` *descrição*: descreve o tipo de retorno de um método e seu intervalo de valores.

Existem outras tags como estas; o leitor interessado deve consultar a documentação online do javadoc para um estudo mais aprofundado.

Clareza e estilo

É possível fazer programas fáceis de ler e entender. Bons programadores devem, portanto, ser cuidadosos com seu estilo de programação, desenvolvendo-o de forma a comunicar os aspectos importantes do projeto de um programa tanto para os usuários como para os computadores.

Alguns dos princípios mais importantes sobre bons estilos de programação são os seguintes:

- *Usar nomes significativos para identificadores*. Devem-se escolher nomes que possam ser lidos em voz alta, que reflitam a ação, a responsabilidade ou os dados que o identificador está nomeando. A tradição na maioria dos círculos de Java é usar maiúsculas na primeira letra de cada palavra que compõem um identificador, excetuando-se a primeira palavra de identificadores de variáveis ou métodos. Então, segundo esta tradição, "Date", "Vector", "DeviceManage" identificam classes e "isFull( )", "insertItem( )", "studentName" e "studentHeigth" referem-se, respectivamente, a métodos e a variáveis.
- *Usar constantes ou tipos enumerados em vez de valores*. A clareza, a robustez e a manutenção serão melhoradas se forem incluídos uma série de valores constantes em uma definição de classe. Estes poderão então ser usados nesta e em outras classes para fazer referência a valores especiais desta classe. A tradição Java é usar apenas maiúsculas em tais constantes, como mostrado abaixo:

```
public class Student {
  public static final int MIN_CREDITS = 12; // créditos mínimos por período
  public static final int MAX_CREDITS = 24; // créditos máximos por período
  public static final int FRESHMAN = 1; // código de calouro
  public static final int SOPHOMORE = 2; // código de aluno do primeiro ano
  public static final int JUNIOR = 3; // código para júnior
  public static final int SENIOR = 4; // código para sênior

  // definições de variáveis de instância, construtores e métodos seguem aqui...
}
```

- *Indentar os blocos de comandos*. Os programadores normalmente indentam cada bloco de comandos com quatro espaços; neste livro, entretanto, usam normalmente dois espaços, para evitar que o código extravase as margens do livro.
- *Organizar as classes conforme a seguinte ordem*:

  1. constantes,
  2. variáveis de instância,
  3. construtores,
  4. métodos.

  Alguns programadores Java preferem colocar as declarações de variáveis de instância por último. Aqui, opta-se por colocá-las antes, de forma que se possa ler cada classe seqüencialmente, compreendendo os dados com que cada método está lidando.
- *Usar comentários para acrescentar significado ao programa e explicar construções ambíguas ou confusas*. Comentários de linha são úteis para explicações rápidas e não precisam ser frases completas. Comentários em bloco são úteis para explicar os propósitos de um método ou as seções de código complicadas.

1.9.4 Teste e depuração

Teste é o processo de verificar a correção de um programa; depuração é o processo de seguir a execução de um programa para descobrir seus erros. Teste e depuração são, em geral, as atividades que mais consomem tempo durante o desenvolvimento de um programa.

Teste

Um plano de testes cuidadoso é parte essencial da escrita de um programa. Apesar de a verificação da correção de um programa para todas as entradas possíveis ser normalmente impraticável, pode-se privilegiar a execução do programa a partir de subconjuntos representativos das entradas. Na pior das hipóteses, deve-se ter certeza de que cada método do programa tenha sido testado pelo menos uma vez (cobertura de método). Melhor ainda, cada linha de código do programa deve ser executada pelo menos uma vez (cobertura de comandos).

Em geral, as entradas dos programas falham em ***casos especiais***. Tais casos precisam ser cuidadosamente identificados e testados. Por exemplo, quando se testa um método que ordena (isto é, coloca em ordem) um arranjo de inteiros, deve-se considerar as seguintes entradas:

- se o arranjo tiver tamanho zero (nenhum elemento);
- se o arranjo tiver um elemento;
- se todos os elementos do arranjo forem iguais;
- se o arranjo já estiver ordenado;
- se o arranjo estiver ordenado na ordem inversa.

Além das entradas especiais para o programa, deve-se também analisar condições especiais para as estruturas usadas pelo programa. Por exemplo, usando-se um arranjo para armazenar dados, é preciso ter certeza de que os casos-limite, tais como a inserção/remoção no início ou no fim do arranjo que armazena os dados, estão sendo convenientemente tratados.

Se é essencial usar conjuntos de testes definidos manualmente, também é fundamental executar o programa a partir de grandes conjuntos de dados gerados randomicamente. A classe `Random` do pacote `java.util` oferece vários métodos para a geração de números randômicos.

Existe uma hierarquia entre as classes e métodos de um programa, induzida pelas relações de "ativador-ativado". Isto é, um método *A* está acima de um método *B* na hierarquia, se *A* chamar *B*. Existem duas estratégias de teste principais, ***top-down*** e ***bottom-up***, que diferem na ordem em que os métodos são testados.

O teste bottom-up é executado desde métodos de mais baixo nível até os de mais alto nível. Ou seja, métodos de mais baixo nível que não ativam outros métodos são testados primeiro, seguidos pelos métodos que chamam apenas um método de baixo nível, e assim por diante. Esta estratégia garante que os erros encontrados em um método nunca são causados por um método de nível mais baixo aninhado no mesmo.

O teste bottom-up é executado do topo para a base da hierarquia de métodos. Normalmente é usado em conjunto com ***terminadores***, uma técnica de rotina de inicialização que substitui métodos de mais baixo nível por um ***tampão****, um substituto para o método que simula a saída do método original. Por exemplo, se o método *A* chama o método *B* para pegar a primeira linha de um arquivo, quando se testa *A* pode-se substituir *B* por um tampão que retorna uma string fixa.

Depuração

A técnica mais simples de depuração consiste em usar ***comandos de impressão*** (usando o método `System.out.println(`*string*`)`) para rastrear os valores das variáveis durante a execução do programa. Um problema desta abordagem é que os comandos de impressão, por vezes, necessitam ser removidos ou comentados antes de o programa poder ser executado.

Uma melhor abordagem é executar o programa com um ***depurador***, que é um ambiente especializado para controlar e monitorar a execução de um programa. A funcionalidade básica

* N. de T. Em inglês, *stub*.

oferecida por um depurador é a inserção de ***pontos de parada**** no código. Quando um programa é executado com um depurador, ele interrompe a cada ponto de parada. Enquanto o programa está parado, o valor corrente das variáveis pode ser verificado. Além de pontos de parada fixos, depuradores mais avançados permitem a especificação de ***pontos de parada condicionais***, que são disparados apenas se uma determinada condição for satisfeita.

As ferramentas-padrão de Java incluem um depurador básico chamado jdb, controlado por linhas de comando. Os IDEs para programação em Java oferecem ambientes de depuração avançados com interface gráfica com o usuário.

1.10 Exercícios

Para obter ajuda e o código fonte dos exercícios, visite **java.datastructures.net**.

Reforço

R-1.1 Suponha que seja criado um arranjo A de objetos GameEntry, que possui um campo inteiro scores, e que A seja clonado e o resultado seja armazenado em um arranjo B. Se o valor de $A[4]$.score for imediatamente alterado para 550, qual o valor do campo score do objeto GameEntry referenciado por $B[4]$?

R-1.2 Modifique a classe CreditCard do Trecho de código 1.5 de maneira a debitar juros em cada pagamento.

R-1.3 Modifique a classe CreditCard do Trecho de código 1.5 de maneira a debitar uma taxa por atraso para qualquer pagamento feito após a data de vencimento.

R-1.4 Modifique a classe CreditCard do Trecho de código 1.5 para incluir ***métodos modificadores*** que permitam ao usuário modificar variáveis internas da classe CreditCard de forma controlada.

R-1.5 Modifique a declaração do primeiro laço **for** da classe Test do Trecho de código 1.6 de maneira que os débitos possam, mais cedo ou mais tarde, fazer com que um dos três cartões ultrapasse seu limite de crédito. Qual é esse cartão?

R-1.6 Escreva uma pequena função em Java, inputAllBaseTypes que recebe diferentes valores de cada um dos tipos base na entrada padrão e o imprime de volta no dispositivo de saída padrão.

R-1.7 Escreva uma classe Java, Flower, que tenha três variáveis de instância dos tipos **String, int** e **float** representando, respectivamente, o nome da flor, seu número de pétalas e o preço. A classe pode incluir um construtor que inicialize cada variável adequadamente, além de métodos para alterar o valor de cada tipo e recuperar o valor de cada tipo.

R-1.8 Escreva uma pequena função em Java, isMultiple, que recebe dois valores **long**, n e m, e retorna **true** se e somente se n é múltiplo de m, isto é, $n = mi$ para algum inteiro i.

R-1.9 Escreva uma pequena função Java isOdd, que recebe um **int** i e retorna **true** se e somente se i é par. Entretanto, esta função não pode usar operadores de multiplicação, módulo ou divisão.

* N. de T. Em inglês, *breakpoints*.

R-1.10 Escreva uma pequena função em Java que receba um inteiro n e retorne a soma de todos os inteiros menores que n.

R-1.11 Escreva uma pequena função em Java que receba um inteiro n e retorne a soma de todos os inteiros pares menores que n.

Criatividade

C-1.1 Escreva uma pequena função Java que recebe um arranjo de valores **int** e determina se existe um par de números no arranjo cujo produto seja par.

C-1.2 Escreva um método Java que recebe um arranjo de valores **int** e determina se todos os números são diferentes entre si (isto é, se são valores distintos).

C-1.3 Escreva um método em Java que receba um arranjo contendo o conjunto de todos os inteiros no intervalo de 1 a 52 e embaralhe os mesmos de forma aleatória. O método deve exibir as possíveis seqüências com igual probabilidade.

C-1.4 Escreva um pequeno programa em Java que exiba todas as strings possíveis de serem formadas usando os caracteres '`c`', '`a`', '`r`', '`b`', '`o`' e '`n`' apenas uma vez.

C-1.5 Escreva um pequeno programa em Java que receba linhas de entrada pelo dispositivo de entrada padrão, e escreva as mesmas no dispositivo de saída padrão na ordem contrária. Isto é, cada linha é exibida na ordem correta, mas a ordem das linhas é invertida.

C-1.6 Escreva um pequeno programa em Java que receba dois arranjos a e b de tamanho n que armazenam valores **int** e retorne o produto escalar de a por b. Isto é, retorna um arranjo c de tamanho n onde $c[i] = a[i] \cdot b[i]$, para $i = 0,...,n-1$.

Projetos

P-1.1 Uma punição comum para alunos de escola é escrever a mesma frase várias vezes. Escreva um programa executável em Java que escreva a mesma frase uma centena de vezes: "Eu não mandarei mais spam para meus amigos". Seu programa deve numerar as frases e "acidentalmente" fazer oito erros aleatórios diferentes de digitação.

P-1.2 (Para aqueles que conhecem os métodos de interface gráfica com o usuário em Java.) Defina uma classe GraphicalTest que teste a funcionalidade da classe CreditCard do Trecho de código 1.5, usando campos de entrada de texto e botões.

P-1.3 O ***paradoxo do aniversário*** diz que a probabilidade de duas pessoas em uma sala terem a mesma data de aniversário é maior que 50% desde que n, o número de pessoas na sala, seja maior que 23. Esta propriedade não é realmente um paradoxo, mas muitas pessoas se surpreendem. Projete um programa em Java que possa testar esse paradoxo por uma série de experimentos sobre aniversários gerados aleatoriamente, testando o paradoxo para $n = 5, 10, 15, 20, ..., 100$.

Observações sobre o capítulo

Para mais informações sobre a linguagem de programação Java, indicamos ao leitor alguns dos melhores livros de Java: Arnold e Gosling [7], Campione e Walrath [19], Cornell e Horstmann [26], Flanagan [34] e Horstmann [51], assim como a página de Java da Sun (http://www.java.sun.com).