



DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Programação Orientada a Objetos: Uma Abordagem com Java

Ivan Luiz Marques Ricarte

2001

Sumário

1	Fundamentos da programação orientada a objetos	3
1.1	Classes	3
1.2	Objetos	4
1.3	Herança	6
1.4	Polimorfismo	6
2	Princípios da programação na linguagem Java	8
2.1	Tipos primitivos	8
2.2	Identificadores	10
2.3	Expressões	11
2.3.1	Expressões retornando valores numéricos	11
2.3.2	Expressões retornando valores booleanos	12
2.3.3	Outros tipos de expressões	13
2.3.4	Controle do fluxo de execução	13
2.3.5	Comentários	15
2.4	Operações sobre objetos	16
2.4.1	Arranjos	18
2.4.2	<i>Strings</i>	19
2.5	Classes em Java	20
2.5.1	Pacotes	20
2.5.2	Definição de classes em Java	21
2.5.3	O método <code>main</code>	24
2.5.4	Visibilidade da classe e seus membros	25
2.5.5	Classes derivadas	25
2.5.6	Classes abstratas e finais	27
2.5.7	Interfaces	28
2.6	Exceções	29
2.6.1	Tratamento de exceções	30
2.6.2	Erros e exceções de <i>runtime</i>	31
2.6.3	Propagando exceções	32
2.6.4	Definindo e gerando exceções	32
2.7	O ambiente de Java	33
2.7.1	Ferramentas do Java SDK	34
2.7.2	Geração de código portátil	34

2.7.3	Desenvolvimento de aplicações	35
3	Uso das classes da API padrão de Java	37
3.1	Funcionalidades básicas	37
3.2	Entrada e saída	38
3.2.1	Transferência de texto	39
3.2.2	Transferência de bytes	40
3.2.3	Manipulação de arquivos	41
3.2.4	Serialização	42
3.3	<i>Framework</i> de coleções	43
3.4	Extensões padronizadas	45
4	Desenvolvimento de aplicações gráficas	46
4.1	Apresentação gráfica	46
4.2	Interfaces gráficas com usuários	47
4.2.1	Eventos da interface gráfica	47
4.2.2	Componentes gráficos	50
4.2.3	<i>Containers</i>	54
4.2.4	Janelas	55
4.2.5	Gerenciadores de <i>layout</i>	57
4.3	Desenvolvimento de applets	65
4.3.1	Criação de <i>applet</i>	66
4.3.2	Execução de <i>applets</i>	67
4.3.3	Passagem de parâmetros	68
4.3.4	Contexto de execução	69
5	Desenvolvimento de aplicações distribuídas	72
5.1	Programação cliente-servidor	72
5.1.1	Conceitos preliminares	73
5.1.2	Aplicações TCP/IP	75
5.1.3	Aplicações UDP	79
5.1.4	Aplicações HTTP	81
5.2	Acesso a bancos de dados	83
5.2.1	Bancos de dados relacionais	84
5.2.2	SQL	85
5.2.3	JDBC	87
5.3	<i>Servlets</i>	90
5.3.1	Ciclo de vida de um <i>servlet</i>	91
5.3.2	Fundamentos da API de <i>servlets</i>	92
5.4	Programação com objetos distribuídos	94
5.4.1	Arquiteturas de objetos distribuídos	94
5.4.2	Java RMI	96
5.4.3	Java IDL	113
A	Palavras chaves de Java	117

Capítulo 1

Fundamentos da programação orientada a objetos

Neste capítulo são apresentados os conceitos básicos que permeiam o uso das técnicas de orientação a objetos na programação, sempre utilizando a linguagem Java como motivador.

Objetos são instâncias de classes, que determinam qual informação um objeto contém e como ele pode manipulá-la.

Um dos grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação que também permitem a definição de estruturas e operações sobre essas estruturas está no conceito de herança, mecanismo através do qual definições existentes podem ser facilmente estendidas. Juntamente com a herança deve ser enfatizada a importância do polimorfismo, que permite selecionar funcionalidades que um programa irá utilizar de forma dinâmica, durante sua execução.

1.1 Classes

A definição de classes e seus inter-relacionamentos é o principal resultado da etapa de projeto de software. Em geral, esse resultado é expresso em termos de alguma linguagem de modelagem, tal como UML.

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades — ou atributos — o objeto terá.

Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe.

Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java.

Na *Unified Modeling Language* (UML), a representação para uma classe no diagrama de classes é tipicamente expressa na forma gráfica, como mostrado na Figura 1.1.

Como se observa nessa figura, a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos da classe e o conjunto de métodos da classe.

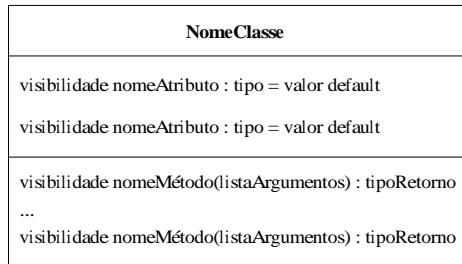


Figura 1.1: Uma classe em UML.

O **nome da classe** é um identificador para a classe, que permite referenciá-la posteriormente — por exemplo, no momento da criação de um objeto.

O conjunto de **atributos** descreve as propriedades da classe. Cada atributo é identificado por um **nome** e tem um **tipo** associado. Em uma linguagem de programação orientada a objetos pura, o tipo é o nome de uma classe. Na prática, a maior parte das linguagens de programação orientada a objetos oferecem um grupo de tipos primitivos, como inteiro, real e caráter, que podem ser usados na descrição de atributos. O atributo pode ainda ter um **valor_default** opcional, que especifica um valor inicial para o atributo.

Os **métodos** definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe. Cada método é especificado por uma **assinatura**, composta por um identificador para o método (o nome do método), o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome.

Através do mecanismo de **sobrecarga** (*overloading*), dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura (*early binding*) para o método correto.

O **modificador de visibilidade** pode estar presente tanto para atributos como para métodos. Em princípio, três categorias de visibilidade podem ser definidas:

público, denotado em UML pelo símbolo +: nesse caso, o atributo ou método de um objeto dessa classe pode ser acessado por qualquer outro objeto (visibilidade externa total);

privativo, denotado em UML pelo símbolo -: nesse caso, o atributo ou método de um objeto dessa classe não pode ser acessado por nenhum outro objeto (nenhuma visibilidade externa);

protegido, denotado em UML pelo símbolo #: nesse caso, o atributo ou método de um objeto dessa classe poderá ser acessado apenas por objetos de classes que sejam derivadas dessa através do mecanismo de herança (ver Seção 1.3).

1.2 Objetos

Objetos são instâncias de classes. É através deles que (praticamente) todo o processamento ocorre em sistemas implementados com linguagens de programação orientadas a objetos. O uso racional de

objetos, obedecendo aos princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é chave para o desenvolvimento de sistemas complexos e eficientes.

Um objeto é um elemento que representa, no domínio da solução, alguma entidade (abstrata ou concreta) do domínio de interesse do problema sob análise. Objetos similares são agrupados em classes.

No paradigma de orientação a objetos, tudo pode ser potencialmente representado como um objeto. Sob o ponto de vista da programação orientada a objetos, um objeto não é muito diferente de uma variável normal. Por exemplo, quando define-se uma variável do tipo `int` em uma linguagem de programação como C ou Java, essa variável tem:

- um espaço em memória para registrar o seu estado (valor);
- um conjunto de operações que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros.

Da mesma forma, quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).

Um programa orientado a objetos é composto por um conjunto de objetos que interagem através de “trocas de mensagens”. Na prática, essa troca de mensagem traduz-se na aplicação de métodos a objetos.

As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, de um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada.

Encapsulação é o princípio de projeto pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade (uma cápsula). Aliado ao conceito de ocultamento de informação, é um poderoso mecanismo da programação orientada a objetos.

Ocultamento da informação é o princípio pelo qual cada componente deve manter oculta sob sua guarda uma decisão de projeto única. Para a utilização desse componente, apenas o mínimo necessário para sua operação deve ser revelado (tornado público).

Na orientação a objetos, o uso da encapsulação e ocultamento da informação recomenda que a representação do estado de um objeto deve ser mantida oculta. Cada objeto deve ser manipulado exclusivamente através dos métodos públicos do objeto, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas dos métodos públicos da classe constitui sua interface operacional.


Dessa forma, detalhes internos sobre a operação do objeto não são conhecidos, permitindo que o usuário do objeto trabalhe em um nível mais alto de abstração, sem preocupação com os detalhes internos da classe. Essa facilidade permite simplificar a construção de programas com funcionalidades complexas, tais como interfaces gráficas ou aplicações distribuídas.


1.3 Herança


O conceito de encapsular estrutura e comportamento em um tipo não é exclusivo da orientação a objetos; particularmente, a programação por tipos abstratos de dados segue esse mesmo conceito. O que torna a orientação a objetos única é o conceito de herança.

Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela.

Há várias formas de relacionamentos em herança:

Extensão: a subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos). A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é normalmente referenciado como **herança estrita**. 

Especificação: a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Diz-se que apenas a interface (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse. 

Combinação de extensão e especificação: a subclasse herda a interface e uma implementação padrão de (pelo menos alguns de) métodos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado mas não implementado. Normalmente, este tipo de relacionamento é denominado herança polimórfica. 

A última forma é, sem dúvida, a que mais ocorre na programação orientada a objetos.

Algumas modelagens introduzem uma forma de herança conhecida como contração. **Contração** é uma variante de herança onde a subclasse elimina métodos da superclasse com o objetivo de criar uma “classe mais simples”. A eliminação pode ocorrer pela redefinição de métodos com corpo vazio. O problema com este mecanismo é que ele viola o **princípio da substituição**, segundo o qual uma subclasse deve poder ser utilizada em todos os pontos onde a superclasse poderia ser utilizada. Se a contração parece ser uma solução adequada em uma hierarquia de classes, provavelmente a hierarquia deve ser re-analisada para detecção de inconsistências (problema pássaros-pinguins). De modo geral, o mecanismo de contração deve ser evitado.

1.4 Polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. Esse mecanismo é fundamental na programação orientada a objetos, permitindo definir funcionalidades que operem genericamente com objetos, abstraindo-se de seus detalhes particulares quando esses não forem necessários.

Para que o polimorfismo possa ser utilizado, é necessário que os métodos que estejam sendo definidos nas classes derivadas tenham exatamente a mesma assinatura do método definido na superclasse; nesse caso, está sendo utilizado o mecanismo de **redefinição de métodos** (*overriding*). Esse mecanismo de redefinição é muito diferente do mecanismo de sobrecarga de métodos, onde as listas de argumentos são diferentes.

No caso do polimorfismo, o compilador não tem como decidir qual o método que será utilizado se o método foi redefinido em outras classes — afinal, pelo princípio da substituição um objeto de uma classe derivada pode estar sendo referenciado como sendo um objeto da superclasse. Se esse for o caso, o método que deve ser selecionado é o da classe derivada e não o da superclasse.

Dessa forma, a decisão sobre qual dos métodos método que deve ser selecionado, de acordo com o tipo do objeto, pode ser tomada apenas em tempo de execução, através do mecanismo de **ligação tardia**. O mecanismo de ligação tardia também é conhecido pelos termos em inglês *late binding*, *dynamic binding* ou ainda *run-time binding*.

Capítulo 2

Princípios da programação na linguagem Java

No restante deste texto, os princípios da programação orientada a objetos são exercitados através de exemplos na linguagem de programação Java. Como qualquer linguagem de programação, Java tem regras estritas indicando como programas devem ser escritos. É bom ter familiaridade com essa sintaxe da linguagem, aprendendo a reconhecer e definir os tipos primitivos, identificadores e expressões em Java.

Merece atenção à parte, para quem não tem familiaridade com o conceito de exceções, o estudo deste mecanismo oferecido por Java para detectar e tratar situações de erro segundo os princípios da orientação a objetos.

2.1 Tipos primitivos

Em Java, são oferecidos tipos literais primitivos (não objetos) para representar valores booleanos, caracteres, numéricos inteiros e numéricos em ponto flutuante.

Variáveis do tipo **boolean** podem assumir os valores `true` ou `false`. O valor *default* para um atributo booleano de uma classe, se não especificado, é `false`. Uma variável do tipo `boolean` ocupa 1 bit de armazenamento.

Variáveis booleanas e variáveis inteiras, ao contrário do que ocorre em C e C++, não são compatíveis em Java. Assim, não faz sentido atribuir uma variável booleana a uma variável inteira ou usar um valor inteiro no contexto de uma condição de teste.

Exemplo de declaração e uso:

```
boolean deuCerto;  
deuCerto = true;
```

Combinando definição e inicialização,

```
boolean deuCerto = true;
```

Uma variável do tipo **char** contém um carácter Unicode¹, ocupando 16 bits de armazenamento

¹Unicode é um padrão internacional para a representação unificada de caracteres de diversas linguagens; para maiores informações, <http://www.unicode.org/>.

em memória. O valor *default* de um atributo de classe do tipo `char`, se não especificado, é o caráter NUL (código hexadecimal 0000).

Um valor literal do tipo caráter é representado entre aspas simples (apóstrofes), como em:

```
char umCaracter = 'A';
```

Nesse caso, a variável ou atributo `umCaracter` recebe o caráter A, correspondente ao código hexadecimal 0041 ou valor decimal 65.

Valores literais de caracteres podem também ser representados por seqüências de escape, como em `'\n'` (o caráter *newline*). A lista de seqüências de escape reconhecidas em Java é apresentada na Tabela 2.1.

<code>\b</code>	backspace
<code>\t</code>	tabulação horizontal
<code>\n</code>	newline
<code>\f</code>	form feed
<code>\r</code>	carriage return
<code>\"</code>	aspas
<code>\'</code>	aspas simples
<code>\\</code>	contrabarra
<code>\xxx</code>	o caráter com código de valor octal xxx, que pode assumir valores entre 000 e 377 na representação octal
<code>\uxxxx</code>	o caráter Unicode com código de valor hexadecimal xxxx, onde xxxx pode assumir valores entre 0000 e ffff na representação hexadecimal.

Tabela 2.1: Caracteres representados por seqüências de escape.

Seqüências de escape Unicode (precedidas por `\u`) são processadas antes das anteriores, podendo aparecer não apenas em variáveis caracteres ou *strings* (como as outras seqüências) mas também em identificadores da linguagem Java.

Valores numéricos inteiros em Java podem ser representados por variáveis do tipo **byte**, **short**, **int** ou **long**. Todos os tipos contém valores inteiros com sinal, com representação interna em complemento de dois. O valor *default* para atributos desses tipos é 0.

Cada um desses tipos de dados tem seu espaço de armazenamento definido na especificação da linguagem, não sendo dependente de diferentes implementações. Variáveis do tipo `byte` ocupam 8 bits de armazenamento interno. Com esse número de bits, é possível representar valores na faixa de -128 a +127. Variáveis do tipo `short` ocupam 16 bits, podendo assumir valores na faixa de -32.768 a +32.767. Variáveis do tipo `int` ocupam 32 bits, podendo assumir valores na faixa de -2.147.483.648 a +2.147.483.647. Finalmente, variáveis do tipo `long` ocupam 64 bits, podendo assumir valores na faixa de -9.223.372.036.854.775.808 a +9.223.372.036.854.775.807.

Constantes literais do tipo `long` podem ser identificadas em código Java através do sufixo `l` ou `L`, como em

```
long valorQuePodeCrescer = 100L;
```

Ao contrário do que ocorre em C, não há valores inteiros sem sinal (unsigned) em Java. Adicionalmente, as combinações da forma `long int` ou `short int` são inválidas em Java.

Valores reais, com representação em ponto flutuante, podem ser representados por variáveis de tipo **float** ou **double**. Em qualquer situação, a representação interna desses valores segue o padrão de representação IEEE 754, sendo 0.0 o valor *default* para tais atributos.

Como para valores inteiros, o espaço de armazenamento e conseqüentemente a precisão de valores associados a esses tipos de dados são definidos na especificação da linguagem. Variáveis do tipo `float` ocupam 32 bits, podendo assumir valores na faixa de $\pm 1.40239846 \times 10^{-45}$ a $\pm 3.40282347 \times 10^{+38}$, com nove dígitos significativos de precisão. Variáveis do tipo `double` ocupam 64 bits, podendo assumir valores de $\pm 4.94065645841246544 \times 10^{-324}$ a $\pm 1.79769313486231570 \times 10^{+308}$, com 18 dígitos significativos de precisão.

Constantes literais do tipo `float` podem ser identificadas no código Java pelo sufixo `f` ou `F`; do tipo `double`, pelo sufixo `d` ou `D`.

2.2 Identificadores

Identificadores são seqüências de caracteres Unicode, que devem obedecer às seguintes regras:

- Um nome pode ser composto por letras, por dígitos e pelos símbolos `_` e `$`.
- Um nome não pode ser iniciado por um dígito (0 a 9).
- Letras maiúsculas são diferenciadas de letras minúsculas.
- Uma palavra-chave da linguagem Java (veja Apêndice A) não pode ser um identificador.

Java diferencia as letras minúsculas da maiúsculas. Assim, as duas variáveis

```
int socorro;  
int soCorro;
```

são variáveis diferentes.

Além dessas regras obrigatórias, o uso da convenção padrão para identificadores Java torna a codificação mais uniforme e pode facilitar o entendimento de um código. Embora não seja obrigatório, o conhecimento e uso da seguinte convenção padrão para atribuir nomes em Java pode facilitar bastante a manutenção de um programa:

- Nomes de classes são iniciados por letras maiúsculas.
- Nomes de métodos, atributos e variáveis são iniciados por letras minúsculas.
- Em nomes compostos, cada palavra do nome é iniciada por letra maiúscula — as palavras não são separadas por nenhum símbolo.

Detalhes sobre as convenções de codificação sugeridas pelos projetistas da linguagem Java podem ser encontrados no documento *Code Conventions for the Java™ Programming Language*².

²<http://www.dca.fee.unicamp.br/projects/sapiens/calm/References/Java/codeconv/CodeConventions.doc8.html>

2.3 Expressões

Expressões são sentenças da linguagem Java terminadas pelo símbolo ‘;’. Essas sentenças podem denotar expressões envolvendo uma operação aritmética, uma operação lógica inteira, uma operação lógica booleana, uma avaliação de condições, uma atribuição, um retorno de método, ou ainda operações sobre objetos.

2.3.1 Expressões retornando valores numéricos

Expressões aritméticas envolvem atributos, variáveis e/ou constantes numéricas (inteiras ou reais). Os operadores aritméticos definidos em Java incluem:

soma, operador binário infixado denotado pelo símbolo +;

subtração, operador binário infixado denotado pelo símbolo -. O mesmo símbolo pode ser utilizado como operador unário prefixado, denotando a complementação (subtração de 0) do valor;

multiplicação, operador binário infixado denotado pelo símbolo *;

divisão, operador binário infixado denotado pelo símbolo /;

resto da divisão, operador binário infixado denotado pelo símbolo % e que pode ser aplicado apenas para operandos inteiros;

incremento, operador unário denotado pelo símbolo ++ que é definido apenas para operandos inteiros, podendo ocorrer na forma prefixa (pré-incremento) ou posfixa (pós-incremento); e

decremento, operador unário denotado pelo símbolo -- que também é definido apenas para operandos inteiros, podendo ocorrer na forma prefixa (pré-decremento) ou posfixa (pós-decremento).

Operações lógicas sobre valores inteiros atuam sobre a representação binária do valor armazenado, operando internamente bit a bit. Operadores desse tipo são:

complemento, operador unário prefixado denotado pelo símbolo ~ que complementa cada bit na representação interna do valor;

OR bit-a-bit, operador binário infixado denotado pelo símbolo | que resulta no bit 1 se pelo menos um dos bits na posição correspondente na representação interna dos operandos era 1;

AND bit-a-bit, operador binário infixado denotado pelo símbolo & que resulta no bit 0 se pelo menos um dos bits na posição correspondente na representação interna dos operandos era 0;

XOR bit-a-bit, operador binário infixado denotado pelo símbolo ^ que resulta no bit 1 se os bits na posição correspondente na representação interna dos operandos eram diferentes;

deslocamento à esquerda, operador binário infixado denotado pelo símbolo << que desloca a representação interna do primeiro operando para a esquerda pelo número de posições indicado pelo segundo operando;

deslocamento à direita, operador binário infixado denotado pelo símbolo `>>` que desloca a representação interna do primeiro operando para a direita pelo número de posições indicado pelo segundo operando. Os bits inseridos à esquerda terão o mesmo valor do bit mais significativo da representação interna;

deslocamento à direita com extensão 0, operador binário infixado denotado pelo símbolo `>>>` que desloca a representação interna do primeiro operando para a direita pelo número de posições indicado pelo segundo operando. Os bits inseridos à esquerda terão o valor 0.

2.3.2 Expressões retornando valores booleanos

As operações lógicas booleanas operam sobre valores booleanos. Operadores booleanos incluem:

complemento lógico, operador unário prefixado denotado pelo símbolo `!` que retorna `true` se o argumento é `false` ou retorna `false` se o argumento é `true`;

OR lógico booleano, operador binário infixado denotado pelo símbolo `|` que retorna `true` se pelo menos um dos dois argumentos é `true`;

OR lógico condicional, operador binário infixado denotado pelo símbolo `||` que opera como o correspondente booleano; porém, se o primeiro argumento já é `true`, o segundo argumento não é nem avaliado;

AND lógico booleano, operador binário infixado denotado pelo símbolo `&` que retorna `false` se pelo menos um dos dois argumentos é `false`;

AND lógico condicional, operador binário infixado denotado pelo símbolo `&&` que opera como o correspondente booleano; porém, se o primeiro argumento já é `false`, o segundo argumento não é nem avaliado;

XOR booleano, operador binário infixado denotado pelo símbolo `^` que retorna `true` quando os dois argumentos têm valores lógicos distintos.

Condições permitem realizar testes baseados nas comparações entre valores numéricos. Operadores condicionais incluem:

maior, operador binário infixado denotado pelo símbolo `>` que retorna `true` se o primeiro valor for exclusivamente maior que o segundo;

maior ou igual, operador binário infixado denotado pelo símbolo `>=` que retorna `true` se o primeiro valor for maior que ou igual ao segundo;

menor, operador binário infixado denotado pelo símbolo `<` que retorna `true` se o primeiro valor for exclusivamente menor que o segundo;

menor ou igual, operador binário infixado denotado pelo símbolo `<=` que retorna `true` se o primeiro valor for menor que ou igual ao segundo;

igual, operador binário infixado denotado pelo símbolo `==` que retorna `true` se o primeiro valor for igual ao segundo;

diferente, operador binário infix denotado pelo símbolo `!=` que retorna `true` se o primeiro valor não for igual ao segundo.

2.3.3 Outros tipos de expressões

Assim como C e C++, Java oferece o operador condicional ternário denotado pelos símbolos `? : .` Na expressão `b ? s1 : s2`, `b` é uma expressão que resulta em um valor booleano (variável ou expressão). O resultado da expressão será o resultado da expressão (ou variável) `s1` se `b` for verdadeiro, ou o resultado da expressão (ou variável) `s2` se `b` for falso.

O operador binário `=` atribui o valor da expressão do lado direito à variável à esquerda do operador. Os tipos da expressão e da variável devem ser compatíveis. O operador de atribuição pode ser combinado com operadores aritméticos e lógicos, como em C e C++. Assim, a expressão

```
a += b
```

equivale a

```
a = a + b
```

Métodos em Java têm sua execução encerrada de duas maneiras possíveis. A primeira é válida quando um método não tem um valor de retorno (o tipo de retorno é `void`): a execução é encerrada quando o bloco do corpo do método chega ao final. A segunda alternativa encerra a execução do método através do comando `return`. Se o método tem tipo de retorno `void`, então o comando é usado sem argumentos:

```
return;
```

Caso o método tenha um valor de retorno especificado, então a expressão assume a forma

```
return expressão;
```

onde o valor de retorno é o resultado da expressão, que deve ser compatível com o tipo de retorno especificado para o método.

2.3.4 Controle do fluxo de execução

A ordem de execução normal de comandos em um método Java é sequencial. Comandos de fluxo de controle permitem modificar essa ordem natural de execução através dos mecanismos de escolha ou de iteração.

A estrutura **if** permite especificar um comando (ou bloco de comandos, uma sequência de expressões delimitada por chaves, `{ e }`) que deve apenas ser executado quando uma determinada condição for satisfeita:

```
if (condição) {  
    bloco_comandos  
}
```

Quando o bloco de comandos é composto por uma única expressão, as chaves que delimitam o corpo do bloco podem ser omitidas:

```
if (condição)
    expressão;
```

Embora a indentação do código não seja mandatória, é uma recomendação de boa prática de programação que o bloco subordinado a uma expressão de controle de fluxo seja representada com maior indentação que aquela usada no nível da própria expressão.

A forma **if...else** permite expressar duas alternativas de execução, uma para o caso da condição ser verdadeira e outra para o caso da condição ser falsa:

```
if (condição) {
    bloco_comandos_caso_verdade
}
else {
    bloco_comandos_caso_falso
}
```

O comando **switch...case** também expressa alternativas de execução, mas nesse caso as condições estão restritas à comparação de uma variável inteira com valores constantes:

```
switch (variável) {
case valor1:
    bloco_comandos
    break;
case valor2:
    bloco_comandos
    break;
...
case valorn:
    bloco_comandos
    break;
default:
    bloco_comandos
}
```

O comando **while** permite expressar iterações que devem ser executadas se e enquanto uma condição for verdadeira:

```
while (condição) {
    bloco_comandos
}
```

O comando **do...while** também permite expressar iterações, mas neste caso pelo menos uma execução do bloco de comandos é garantida:

```
do {
    bloco_comandos
} while (condição);
```

O comando **for** permite expressar iterações combinando uma expressão de inicialização, um teste de condição e uma expressão de incremento:

```
for (inicialização; condição; incremento) {  
    bloco_comandos  
}
```

Embora Java não suporte o operador `,` (vírgula) presente em C e C++, no comando `for` múltiplas expressões de inicialização e de incremento podem ser separadas por vírgulas.

Os comandos `continue` e `break` permitem expressar a quebra de um fluxo de execução. O uso de `break` já foi utilizado juntamente com `switch` para delimitar bloco de comandos de cada `case`. No corpo de uma iteração, a ocorrência do comando `break` interrompe a iteração, passando o controle para o próximo comando após o comando de iteração. O comando `continue` também interrompe a execução da iteração, mas apenas da iteração corrente. Como efeito da execução de `continue`, a condição de iteração é reavaliada e, se for verdadeira, o comando de iteração continua executando.

Em situações onde há diversos comandos de iteração aninhados, os comandos `break` e `continue` transferem o comando de execução para o ponto imediatamente após o bloco onde ocorrem. Se for necessário especificar transferência para o fim de outro bloco de iteração, os comandos `break` e `continue` rotulados podem ser utilizados:

```
label: {  
    for (...) {  
        ...  
        while (...) {  
            ...  
            if (...)  
                break label;  
            ...  
        }  
        ...  
    }  
    ...  
}
```

2.3.5 Comentários

Comentários em Java podem ser expressos em três formatos distintos. Na primeira forma, uma “barra dupla” `//` marca o início do comentário, que se estende até o fim da linha. A segunda forma é o comentário no estilo C tradicional, onde o par de símbolos `/*` marca o início de comentário, que pode se estender por diversas linhas até encontrar o par de símbolos `*/`, que delimita o fim do comentário.

O outro estilo de comentário é específico de Java e está associado à geração automática de documentação do *software* desenvolvido. Nesse caso, o início de comentário é delimitado pelas seqüências de início `/**` e de término `*/`. O texto entre essas seqüências será utilizado pela ferramenta `javadoc` (ver Seção 2.7.1) para gerar a documentação do código em formato hipertexto.

Para gerar esse tipo de documentação, os comentários devem estar localizados imediatamente antes da definição da classe ou membro (atributo ou método) documentado. Cada comentário pode conter uma descrição textual sobre a classe ou membro, possivelmente incluindo *tags* HTML, e diretrizes para o programa javadoc. As diretrizes para javadoc são sempre precedidas por @, como em

```
@see nomeClasseOuMembro
```

que gera na documentação um *link* para a classe ou membro especificado, precedido pela frase “See also”.

A documentação de uma classe pode incluir as diretrizes @author, que inclui o texto na sequência como informação sobre o autor do código na documentação; e @version, que inclui na documentação informação sobre a versão do código.

A documentação de um método pode incluir as diretrizes @param, que apresenta o nome e uma descrição de cada argumento do método; @return, que apresenta uma descrição sobre o valor de retorno; e @exception, que indica que exceções podem ser lançadas pelo método (ver Seção 2.6.1).

Para maiores informações, veja o texto *How to Write Doc Comments for Javadoc*³.

2.4 Operações sobre objetos

A criação de um objeto dá-se através da aplicação do operador **new**. Dada uma classe `Cls`, é possível (em princípio) criar um objeto dessa classe usando esse operador:

```
Cls obj = newCls();
```

O “método” à direita do operador `new` é um construtor da classe `Cls`. Um construtor é um (pseudo-)método especial, definido para cada classe. O corpo desse método determina as atividades associadas à inicialização de cada objeto criado. Assim, o construtor é apenas invocado no momento da criação do objeto através do operador `new`.

A assinatura de um construtor diferencia-se das assinaturas dos outros métodos por não ter nenhum tipo de retorno — nem mesmo `void`. Além disto, o nome do construtor deve ser o próprio nome da classe.

O construtor pode receber argumentos, como qualquer método. Usando o mecanismo de sobrecarga, mais de um construtor pode ser definido para uma classe.

Toda classe tem pelo menos um construtor sempre definido. Se nenhum construtor for explicitamente definido pelo programador da classe, um construtor *default*, que não recebe argumentos, é criado pelo compilador Java. No entanto, se o programador da classe criar pelo menos um método construtor, o construtor *default* não será criado automaticamente — se ele o desejar, deverá criar um construtor sem argumentos explicitamente.

No momento em que um construtor é invocado, a seguinte sequência de ações é executada para a criação de um objeto:

1. O espaço para o objeto é alocado e seu conteúdo é inicializado (bitwise) com zeros.
2. O construtor da classe base é invocado.

³[urlhttp://java.sun.com/products/jdk/javadoc/writingdoccomments.html](http://java.sun.com/products/jdk/javadoc/writingdoccomments.html)

3. Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.
4. O restante do corpo do construtor é executado.

Seguir essa sequência é uma necessidade de forma a garantir que, quando o corpo de um construtor esteja sendo executado, o objeto já terá à disposição as funcionalidades mínimas necessárias, quais sejam aquelas definidas por seus ancestrais. O primeiro passo garante que nenhum campo do objeto terá um valor arbitrário, que possa tornar erros de não inicialização difíceis de detectar. Esse passo é que determina os valores *default* para atributos de tipos primitivos, descritos na Seção 2.1.

Uma vez que um objeto tenha sido criado e haja uma referência válida para ele, é possível solicitar que ele execute métodos que foram definidos para objetos dessa classe. A aplicação de um método `meth(int)`, definido em uma classe `Cls`, a um objeto `obj`, construído a partir da especificação de `Cls`, se dá através da construção

```
obj.meth(varint);
```

onde `varint` é uma expressão ou variável inteira anteriormente definida.

Em uma linguagem tradicional, a forma correspondente seria invocar uma função ou procedimento passando como argumento a estrutura de dados sobre a qual as operações teriam efeito:

```
meth(obj, varint);
```

Esse “atributo implícito” — a referência para o próprio objeto que está ativando o método — pode ser utilizado no corpo dos métodos quando for necessário fazê-lo explicitamente. Para tanto, a palavra-chave `this` é utilizada.

Quando se declara uma variável cujo tipo é o nome de uma classe, como em

```
String nome;
```

não está se criando um objeto dessa classe, mas simplesmente uma referência para um objeto da classe `String`, a qual inicialmente não faz referência a nenhum objeto válido.

Quando um objeto dessa classe é criado, obtém-se uma referência válida, que é armazenada na variável cujo tipo é o nome da classe do objeto. Por exemplo, quando cria-se uma *string* como em

```
nome = new String("POO/Java");
```

`nome` é uma variável que armazena uma referência válida para um objeto específico da classe `String` — o objeto cujo conteúdo é a *string* `POO/Java`. É importante ressaltar que a variável `nome` mantém apenas a referência para o objeto e não o objeto em si. Assim, uma atribuição como

```
String outroNome = nome;
```

não cria outro objeto, mas simplesmente uma outra referência para o mesmo objeto.

A linguagem Java permite que o programador crie e manipule objetos explicitamente. Porém, a remoção de objetos em Java é manipulada automaticamente pelo sistema, usando um mecanismo de coleta de lixo (*garbage collector*). O coletor de lixo é um processo de baixa prioridade que permanece verificando quais objetos não têm nenhuma referência válida, retomando o espaço despendido para

cada um desses objetos. Dessa forma, o programador não precisa se preocupar com a remoção explícita de objetos.

Outra operação que pode envolver um objeto é sua verificação de tipo. Dado um objeto `obj` e uma classe `Cls`, é possível verificar dinamicamente (durante a execução do método) se o objeto pertence ou não à classe usando o operador `instanceof`. Este retorna `true` se o objeto à esquerda do operador é da classe especificada à direita do operador. Assim,

```
obj instanceof Cls
```

retornaria `true` se `obj` fosse da classe `Cls`.

2.4.1 Arranjos

Arranjos são agregados homogêneos de valores que podem agrupar literais ou objetos.

A declaração de um arranjo, como

```
int array1[];
```

apenas cria uma referência para um arranjo de inteiros — porém o arranjo não foi criado. Assim como objetos, arranjos são criados com o operador `new`:

```
array1 = new int[100];
```

Essa expressão cria espaço para armazenar 100 inteiros no arranjo `array1`.

As duas expressões poderiam ter sido combinadas em uma sentença incluindo a declaração e a criação de espaço:

```
int array1[] = new int[100];
```

Arranjos podem ser alternativamente criados com a especificação de algum conteúdo:

```
int array2[] = {2, 4, 5, 7, 9, 11, 13};
```

O acesso a elementos individuais de um arranjo é especificado através de um índice inteiro. O elemento inicial, assim como em C e C++, tem índice 0. Assim, do exemplo acima, a expressão

```
int x = array2[3];
```

faz com que a variável `x` receba o valor 7, o conteúdo da quarta posição.

O acesso a elementos do arranjo além do último índice permitido — por exemplo, a `array2[7]` — gera um erro em tempo de execução, ou uma exceção (ver Seção 2.6.1).

A dimensão de um arranjo pode ser obtida através da propriedade `length` presente em todos os arranjos. Assim, a expressão

```
int y = array2.length;
```

faz com que `y` receba o valor 7, o número de elementos no arranjo `array2`.

2.4.2 Strings

Ao contrário do que ocorre em C e C++, *strings* em Java não são tratadas como seqüências de caracteres terminadas por NUL. São objetos, instâncias da classe `java.lang.String`.

Uma *string* pode ser criada como em:

```
String s = "abc";
```

O operador `+` pode ser utilizado concatenar strings:

```
System.out.println("String s: " + s);
```

Se, em uma mesma expressão, o operador `+` combinar valores numéricos e *strings*, os valores numéricos serão convertidos para *strings* e então concatenados.

A classe `String` define um conjunto de funcionalidades para manipular *strings* através de seus métodos.

Para obter o número de caracteres em uma *string*, o método `int length()` é usado. Assim, a expressão

```
s.length();
```

retornaria o valor 3.

Para criar uma nova *string* a partir da concatenar duas *strings*, o método `String concat(String outro)` pode ser usado. Por exemplo,

```
String t = s.concat(".java");
```

faria com que a *string* `t` tivesse o conteúdo “abc.java”.

Para comparar o conteúdo de duas *strings* há três formas básicas. O método `equals(String outro)` compara as duas *strings*, retornando verdadeiro se seus conteúdos forem exatamente iguais. Assim,

```
t.equals(s);
```

retornaria falso, mas

```
s.equals("abc");
```

retornaria verdadeiro. O método `equalsIgnoreCase(String outro)` tem a mesma funcionalidade mas ignora se as letras são maiúsculas ou minúsculas. Assim,

```
s.equalsIgnoreCase("ABC");
```

também retornaria verdadeiro. A terceira forma é através do método `compareTo(String outro)`, que retorna um valor inteiro igual a zero se as duas *strings* forem iguais, negativo se a *string* à qual o método for aplicado preceder lexicograficamente a *string* do argumento (baseado nos valores Unicode dos caracteres) ou positivo, caso contrário. Há também um método `compareToIgnoreCase(String str)`.

Para extrair caracteres individuais de uma *string*, pode-se utilizar o método `charAt(int pos)`, que retorna o caráter na posição especificada no argumento (0 é a primeira posição). Para obter *substrings* de uma *string*, o método `substring()` pode ser utilizado. Esse método tem duas assinaturas:

```
String substring(int pos_inicial);  
String substring(int pos_inicial, int pos_final);
```

A localização de uma *substring* dentro de uma *string* pode ocorrer de diversas formas. O método `indexOf()`, com quatro assinaturas distintas, retorna a primeira posição na *string* onde o caráter ou *substring* especificada no argumento ocorre. É possível também especificar uma posição diferente da inicial a partir de onde a busca deve ocorrer. Similarmente, `lastIndexOf()` busca pela última ocorrência do caráter ou *substring* a partir do final ou da posição especificada.

É possível também verificar se uma *string* começa com um determinado prefixo ou termina com um sufixo através respectivamente dos métodos `startsWith()` e `endsWith()`. Ambos recebem como argumento uma *string* e retornam um valor booleano.

2.5 Classes em Java

Como descrito na Seção 1.1, a definição de classes é a base da programação orientada a objetos. Em Java, classes são definidas em arquivos fonte (extensão `.java`) e compiladas para arquivos de classes (extensão `.class`). Classes são organizadas em pacotes.

2.5.1 Pacotes

No desenvolvimento de pequenas atividades ou aplicações, é viável manter o código da aplicação e suas classes associadas em um mesmo diretório de trabalho — em geral, o diretório corrente. No entanto, para grandes aplicações é preciso organizar as classes de maneira a evitar problemas com nomes duplicados de classes e permitir a localização do código da classe de forma eficiente.

Em Java, a solução para esse problema está na organização de classes e interfaces em pacotes. Um pacote é uma unidade de organização de código que congrega classes, interfaces e exceções relacionadas. O código-base de Java está todo estruturado em pacotes e as aplicações desenvolvidas em Java também devem ser assim organizadas.

Essencialmente, uma classe `Xyz` que pertence a um pacote `nome.do.pacote` tem um “nome completo” que é `nome.do.pacote.Xyz`. Assim, se outra aplicação tiver uma classe de mesmo nome não haverá conflitos de resolução, pois classes em pacotes diferentes têm nomes completos distintos.

A organização das classes em pacotes também serve como indicação para o compilador Java para encontrar o arquivo que contém o código da classe. O ambiente Java normalmente utiliza a especificação de uma variável de ambiente `CLASSPATH`, a qual define uma lista de diretórios que contém os arquivos de classes Java. No entanto, para não ter listas demasiadamente longas, os nomes dos pacotes definem subdiretórios de busca a partir dos diretórios em `CLASSPATH`.

No mesmo exemplo, ao encontrar no código uma referência para a classe `Xyz`, o compilador deverá procurar o arquivo com o nome `Xyz.class`; como essa classe faz parte do pacote `nome.do.pacote`, ele irá procurar em algum subdiretório `nome/do/pacote`. Se o arquivo `Xyz.class` estiver no diretório `/home/java/nome/do/pacote`, então o diretório `/home/java` deve estar incluído no caminho de busca de classes definido por `CLASSPATH`.

Para indicar que as definições de um arquivo fonte Java fazem parte de um determinado pacote, a primeira linha de código deve ser a declaração de pacote:

```
package nome.do.pacote;
```

Caso tal declaração não esteja presente, as classes farão parte do “pacote *default*”, que está mapeado para o diretório corrente.

Para referenciar uma classe de um pacote no código fonte, é possível sempre usar o “nome completo” da classe; no entanto, é possível também usar a declaração `import`. Por exemplo, se no início do código estiver presente a declaração

```
import nome.do.pacote.Xyz;
```

então a classe `Xyz` pode ser referenciada sem o prefixo `nome.do.pacote` no restante do código. Alternativamente, a declaração

```
import nome.do.pacote.*;
```

indica que quaisquer classes do pacote especificado podem ser referenciadas apenas pelo nome no restante do código fonte.

A única exceção para essa regra refere-se às classes do pacote `java.lang` — essas classes são consideradas essenciais para a interpretação de qualquer programa Java e, por este motivo, o correspondente `import` é implícito na definição de qualquer classe Java.

2.5.2 Definição de classes em Java

Em Java, classes são definidas através do uso da palavra-chave `class`. Para definir uma classe, utiliza-se a construção:

```
[modif] class NomeDaClasse {  
    // corpo da classe...  
}
```

A primeira linha é um comando que inicia a declaração da classe. Após a palavra-chave **class**, segue-se o nome da classe, que deve ser um identificador válido para a linguagem (ver Seção 2.2). O modificador *modif* é opcional; se presente, pode ser uma combinação de **public** e **abstract** ou **final**.

A definição da classe propriamente dita está entre as chaves { e }, que delimitam blocos na linguagem Java. Este corpo da classe usualmente obedece à seguinte seqüência de definição:

1. As variáveis de classe (definidas como `static`), ordenadas segundo sua visibilidade: iniciando pelas **public**, seguidos pelas **protected**, pelas com visibilidade padrão (sem modificador) e finalmente pelas **private**.
2. Os atributos (ou variáveis de instância) dos objetos dessa classe, seguindo a mesma ordenação segundo a visibilidade definida para as variáveis de classe.
3. Os construtores de objetos dessa classe.
4. Os métodos da classe, geralmente agrupados por funcionalidade.

A definição de atributos de uma classe Java reflete de forma quase direta a informação que estaria contida na representação da classe em um diagrama UML. Para tanto, a sintaxe utilizada para definir um atributo de um objeto é:

```
[modificador] tipo nome [ = default];
```

onde

- o modificador é opcional, especificando a visibilidade diferente da padrão (`public`, `protected` ou `private`), alteração da modificabilidade (`final`) e se o atributo está associado à classe (`static`).
- o tipo deve ser um dos tipos primitivos da linguagem Java ou o nome de uma classe;
- o nome deve ser um identificador válido da linguagem Java;
- o valor *default* é opcional; se presente, especifica um valor inicial para a variável.

Métodos são essencialmente procedimentos que podem manipular atributos de objetos para os quais o método foi definido. Além dos atributos de objetos, métodos podem definir e manipular variáveis locais; também podem receber parâmetros por valor através da lista de argumentos. A forma genérica para a definição de um método em uma classe é

```
[modificador] tipo nome(argumentos) {  
    corpo do método  
}
```

onde

- o modificador (opcional) é uma combinação de: `public`, `protected` ou `private`; `abstract` ou `final`; e `static`.
- o tipo é um indicador do valor de retorno, sendo `void` se o método não tiver um valor de retorno;
- o nome do método deve ser um identificador válido na linguagem Java;
- os argumentos são representados por uma lista de parâmetros separados por vírgulas, onde para cada parâmetro é indicado primeiro o tipo e depois (separado por espaço) o nome.

Uma boa prática de programação é manter a funcionalidade de um método simples, desempenhando uma única tarefa. O nome do método deve refletir de modo adequado a tarefa realizada. Se a funcionalidade do método for simples, será fácil encontrar um nome adequado para o método.

Como ocorre para a definição de atributos, a definição de métodos reflete de forma quase direta a informação que estaria presente em um diagrama de classes UML, a não ser por uma diferença vital: o corpo do método.

Métodos de mesmo nome podem co-existir em uma mesma classe desde que a lista de argumentos seja distinta, usando o mecanismo de sobrecarga.

O exemplo a seguir ilustra a definição de uma classe de nome `Ponto2D`:

```
public class Ponto2D {  
    private int x;  
    private int y;
```

```
public Ponto2D(int x, int y) {
    this.x = x;
    this.y = y;
}

public Ponto2D( ) {
    this(0,0);
}

public double distancia(Ponto2D p) {
    double distX = p.x - x;
    double distY = p.y - y;

    return Math.sqrt(distX*distX + distY*distY);
}
}
```

Um objeto dessa classe tem dois atributos privativos que definem as coordenadas do ponto bidimensional, *x* e *y* — nesse caso, as coordenadas são valores inteiros.

A classe tem dois construtores. O primeiro deles recebe dois argumentos, também de nome *x* e *y*, que definem as coordenadas do ponto criado. Para diferenciar no corpo do construtor os parâmetros dos atributos, estes têm seu nome prefixado pela palavra-chave *this*.

O segundo construtor ilustra outro uso da palavra-chave *this*. Esse construtor não recebe argumentos e assume portanto o ponto tem como coordenadas a origem, ou seja, o ponto (0,0). O corpo desse construtor poderia ser simplesmente

```
x = 0;
y = 0;
```

A alternativa apresentada usa a forma

```
this(0,0);
```

que equivale a “use o construtor desta mesma classe que recebe dois argumentos inteiros, passando os valores aqui especificados”.

Finalmente, a classe define um método público para calcular a distância entre o ponto que invocou o método (*this*) e outro ponto especificado como o argumento *p*. Nesse caso, como não há risco de confusão, não é preciso usar a palavra-chave *this* antes dos atributos da coordenada do objeto. Assim, as expressões que calculam *distX* e *distY* equivalem a

```
double distX = p.x - this.x;
double distY = p.y - this.y;
```

A expressão de retorno ilustra ainda a utilização de um método estático da classe *Math* do pacote *java.lang* para o cálculo da raiz quadrada. Usualmente, métodos definidos em uma classe são aplicados a objetos daquela classe. Há no entanto situações nas quais um método pode fazer uso dos recursos de uma classe para realizar sua tarefa sem necessariamente ter de estar associado a um objeto individualmente.

Para lidar com tais situações, Java define os métodos da classe, cuja declaração deve conter o modificador `static`. Um método estático pode ser aplicado à classe e não necessariamente a um objeto.

Exemplos de métodos estáticos em Java incluem os métodos para manipulação de tipos primitivos definidos nas classes `Character`, `Integer` e `Double`, todas elas do pacote `java.lang`, assim como todos os métodos definidos para a classe `Math`.

2.5.3 O método `main`

Toda classe pode também ter um método **main** associado, que será utilizado pelo interpretador Java para dar início à execução de uma aplicação. Ao contrário do que acontece em C e C++, onde apenas uma função `main` deve ser definida para a aplicação como um todo, toda e qualquer classe Java pode ter um método `main` definido. Apenas no momento da interpretação o `main` a ser executado é definido através do primeiro argumento (o nome da classe) para o programa interpretador.

O método `main` é um método associado à classe e não a um objeto específico da classe — assim, ele é definido como um método estático. Adicionalmente, deve ser um método público para permitir sua execução a partir da máquina virtual Java. Não tem valor de retorno, mas recebe como argumento um arranjo de strings que corresponde aos parâmetros que podem ser passados para a aplicação a partir da linha de comando. Essas características determinam a assinatura do método:

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

ou

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

Mesmo que não seja utilizado, o argumento de `main` deve ser especificado. Por exemplo, a definição da classe `Ponto2D` poderia ser complementada com a inclusão de um método para testar sua funcionalidade:

```
public class Ponto2D {  
    ...  
    public static void main(String[] args) {  
        Ponto2D ref2 = new Ponto2D();  
        Ponto2D p2 = new Ponto2D(1,1);  
        System.out.println("Distancia: " + p2.distancia(ref2));  
    }  
}
```

O nome do parâmetro (`args`) obviamente poderia ser diferente, mas os demais termos da assinatura devem obedecer ao formato especificado. Esse argumento é um parâmetro do tipo arranjo de objetos da classe `String`. Cada elemento desse arranjo corresponde a um argumento passado para o interpretador Java na linha de comando que o invocou. Por exemplo, se a linha de comando for

```
java Xyz abc 123 def
```

o método `main(String[] args)` da classe `Xyz` vai receber, nessa execução, um arranjo de três elementos na variável `args` com os seguintes conteúdos:

- em `args[0]`, o objeto `String` com conteúdo "abc";
- em `args[1]`, o objeto `String` com conteúdo "123";
- em `args[2]`, o objeto `String` com conteúdo "def".

Como o método `main` é do tipo `void`, ele não tem valor de retorno. No entanto, assim como programas desenvolvidos em outras linguagens, é preciso algumas vezes obter uma indicação se o programa executou com sucesso ou não. Isto é principalmente útil quando o programa é invocado no contexto de um script do sistema operacional.

Em Java, o mecanismo para fornecer essa indicação é o método `System.exit(int)`. A invocação desse método provoca o fim imediato da execução do interpretador Java. Tipicamente, o argumento de `exit()` obedece à convenção de que '0' indica execução com sucesso, enquanto um valor diferente de 0 indica a ocorrência de algum problema.

2.5.4 Visibilidade da classe e seus membros

Em Java, a visibilidade padrão de classes, atributos e métodos está restrita a todos os membros que fazem parte de um mesmo pacote. A palavra-chave **public** modifica essa visibilidade de forma a ampliá-la, deixando-a sem restrições.

Uma classe definida como pública pode ser utilizada por qualquer objeto de qualquer pacote. Em Java, uma unidade de compilação (um arquivo fonte com extensão `.java`) pode ter no máximo uma classe pública, cujo nome deve ser o mesmo do arquivo (sem a extensão). As demais classes na unidade de compilação, não públicas, são consideradas classes de suporte para a classe pública e têm a visibilidade padrão.

Um atributo público de uma classe pode ser diretamente acessado e manipulado por objetos de outras classes.

Um método público de uma classe pode ser aplicado a um objeto dessa classe a partir de qualquer outro objeto de outra classe. O conjunto de métodos públicos de uma classe determina o que pode ser feito com objetos da classe, ou seja, determina o seu comportamento.

A palavra-chave `protected` restringe a visibilidade do membro modificado, atributo ou método, apenas à própria classe e àquelas derivada desta.

A palavra-chave `private` restringe a visibilidade do membro modificado, método ou atributo, exclusivamente a objetos da própria classe que contém sua definição.

2.5.5 Classes derivadas

Sendo uma linguagem de programação orientada a objetos, Java oferece mecanismos para definir classes derivadas a partir de classes existentes. É fundamental que se tenha uma boa compreensão sobre como objetos de classes derivadas são criados e manipulados, assim como das restrições de acesso que podem se aplicar a membros de classes derivadas. Também importante para uma completa compreensão da utilização desse mecanismo em Java é entender como relacionam-se interfaces e herança (Seção 2.5.7).

A forma básica de herança em Java é a extensão simples entre uma superclasse e sua classe derivada. Para tanto, utiliza-se na definição da classe derivada a palavra-chave **extends** seguida pelo nome da superclasse.

A hierarquia de classes de Java tem como raiz uma classe básica, `Object`. Quando não for especificada uma superclasse na definição de uma classe, o compilador assume que a superclasse é `Object`. Assim, definir a classe `Ponto2D` como em

```
class Ponto2D {  
    // ...  
}
```

é equivalente a defini-la como

```
class Ponto2D extends Object {  
    // ...  
}
```

É por esse motivo que todos os objetos podem invocar os métodos da classe `Object`, tais como `equals()` e `toString()`. O método `equals` permite comparar objetos por seus conteúdos. A implementação padrão desse método realiza uma comparação de conteúdo bit a bit; se um comportamento distinto for desejado para uma classe definida pelo programador, o método deve ser redefinido. O método `toString()` permite converter uma representação interna do objeto em uma *string* que pode ser apresentada ao usuário.

Outros métodos da classe `Object` incluem `clone()`, um método protegido que permite criar uma duplicata de um objeto, e `getClass()`, que retorna um objeto que representa a classe à qual o objeto pertence. Esse objeto será da classe `Class`; para obter o nome da classe, pode-se usar o seu método estático `getName()`, que retorna uma *string*.

Para criar uma classe `Ponto3D` a partir da definição da classe que representa um ponto em duas dimensões, a nova classe deve incluir um atributo adicional para representar a terceira coordenada:

```
public class Ponto3D extends Ponto2D {  
    private int z;  
  
    public Ponto3D(int x, int y, int z) {  
        super(x, y);  
        this.z = z;  
    }  
  
    public Ponto3D( ) {  
        z = 0;  
    }  
  
    public static void main(String[] args) {  
        Ponto2D ref2 = new Ponto2D();  
        Ponto3D p3 = new Ponto3D(1,2,3);  
        System.out.println("Distancia: " + p3.distancia(ref2));  
    }  
}
```

Nesse exemplo, o método `distancia` que é utilizado em `main` é aquele que foi definido para a classe `Ponto2D` que, por herança, é um método da classe `Ponto3D`.

Esse exemplo ilustra, na definição do primeiro construtor, o uso da palavra-chave `super` para invocar um construtor específico da superclasse. Se presente, essa expressão deve ser a primeira do construtor, pois o início da construção do objeto é a construção da parte da superclasse. Caso não esteja presente, está implícita uma invocação para o construtor padrão, sem argumentos, da superclasse, equivalente à forma `super()`.

2.5.6 Classes abstratas e finais

Uma **classe abstrata** não pode ser instanciada, ou seja, não há objetos que possam ser construídos diretamente de sua definição. Por exemplo, a compilação do seguinte trecho de código

```
abstract class AbsClass {
    public static void main(String[] args) {
        AbsClass obj = new AbsClass();
    }
}
```

geraria a seguinte mensagem de erro:

```
AbsClass.java:3: class AbsClass is an abstract class.
               It can't be instantiated.
               AbsClass obj = new AbsClass();
                           ^
1 error
```

Em geral, classes abstratas definem um conjunto de funcionalidades das quais pelo menos uma está especificada mas não está definida — ou seja, contém pelo menos um método abstrato, como em

```
abstract class AbsClass {
    public abstract int umMetodo();
}
```

Um método abstrato não cria uma definição, mas apenas uma declaração de um método que deverá ser implementado em uma classe derivada. Se esse método não for implementado na classe derivada, esta permanece como uma classe abstrata mesmo que não tenha sido assim declarada explicitamente.

Assim, para que uma classe derivada de uma classe abstrata possa gerar objetos, os métodos abstratos devem ser definidos em classes derivadas:

```
class ConcClass extends AbsClass {
    public int umMetodo() {
        return 0;
    }
}
```

Uma **classe final**, por outro lado, indica uma classe que não pode ser estendida. Assim, a compilação do arquivo `Reeleicao.java` com o seguinte conteúdo:

```
final class Mandato {  
}  
  
public class Reeleicao extends Mandato {  
}
```

ocasionaria um erro de compilação:

```
caolho:Exemplos[39] javac Reeleicao.java  
Reeleicao.java:4: Can't subclass final classes: class Mandato  
public class Reeleicao extends Mandato {  
                        ^  
1 error
```

A palavra-chave **final** pode também ser aplicada a métodos e a atributos de uma classe. Um método final não pode ser redefinido em classes derivadas.

Um atributo final não pode ter seu valor modificado, ou seja, define valores constantes. Apenas valores de tipos primitivos podem ser utilizados para definir constantes. O valor do atributo deve ser definido no momento da declaração, pois não é permitida nenhuma atribuição em outro momento.

A utilização de final para uma referência a objetos é permitida. Como no caso de constantes, a definição do valor (ou seja, a criação do objeto) também deve ser especificada no momento da declaração. No entanto, é preciso ressaltar que o conteúdo do objeto em geral pode ser modificado — apenas a referência é fixa. O mesmo é válido para arranjos.

A partir de Java 1.1, é possível ter atributos de uma classe que sejam final mas não recebem valor na declaração, mas sim nos construtores da classe. (A inicialização deve obrigatoriamente ocorrer em uma das duas formas.) São os chamados *blank finals*, que introduzem um maior grau de flexibilidade na definição de constantes para objetos de uma classe, uma vez que essas podem depender de parâmetros passados para o construtor.

Argumentos de um método que não devem ser modificados podem ser declarados como final, também, na própria lista de parâmetros.

2.5.7 Interfaces

Java também oferece outra estrutura, denominada interface, com sintaxe similar à de classes mas contendo apenas a especificação da funcionalidade que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada. Uma interface Java é uma classe abstrata para a qual todos os métodos são implicitamente `abstract` e `public`, e todos os atributos são implicitamente `static` e `final`. Em outros termos, uma interface Java implementa uma “classe abstrata pura”.

A sintaxe para a declaração de uma interface é similar àquela para a definição de classes, porém seu corpo define apenas assinaturas de métodos e constantes. Por exemplo, para definir uma interface `Interfacel` que declara um método `met1` sem argumentos e sem valor de retorno, a sintaxe é:

```
interface Interfacel {  
    void met1();  
}
```

A diferença entre uma classe abstrata e uma interface Java é que a interface obrigatoriamente não tem um “corpo” associado. Para que uma classe seja abstrata basta que ela seja assim declarada, mas a classe pode incluir atributos de objetos e definição de métodos, públicos ou não. Na interface, apenas métodos públicos podem ser declarados — mas não definidos. Da mesma forma, não é possível definir atributos — apenas constantes públicas.

Enquanto uma classe abstrata é “estendida” (palavra chave `extends`) por classes derivadas, uma interface Java é “implementada” (palavra chave `implements`) por outras classes.

Uma interface estabelece uma espécie de contrato que é obedecido por uma classe. Quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

Outro uso de interfaces Java é para a definição de constantes que devem ser compartilhadas por diversas classes. Neste caso, a recomendação é implementar interfaces sem métodos, pois as classes que implementarem tais interfaces não precisam tipicamente redefinir nenhum método:

```
interface Coins {
    int
    PENNY = 1,
    NICKEL = 5,
    DIME = 10,
    QUARTER = 25,
    DOLAR = 100;
}

class SodaMachine implements Coins {
    int price = 3*QUARTER;
    // ...
}
```

2.6 Exceções

Uma exceção é um sinal que indica que algum tipo de condição excepcional ocorreu durante a execução do programa. Assim, exceções estão associadas a condições de erro que não tinham como ser verificadas durante a compilação do programa.

As duas atividades associadas à manipulação de uma exceção são:

geração: a sinalização de que uma condição excepcional (por exemplo, um erro) ocorreu, e

captura: a manipulação (tratamento) da situação excepcional, onde as ações necessárias para a recuperação da situação de erro são definidas.

Para cada exceção que pode ocorrer durante a execução do código, um bloco de ações de tratamento (um *exception handler*) deve ser especificado. O compilador Java verifica e enforça que toda exceção “não-trivial” tenha um bloco de tratamento associado.

O mecanismo de manipulação de exceções em Java, embora apresente suas particularidades, teve seu projeto inspirado no mecanismo equivalente de C++, que por sua vez foi inspirado em Ada. É

um mecanismo adequado à manipulação de erros síncronos, para situações onde a recuperação do erro é possível.

A sinalização da exceção é propagada a partir do bloco de código onde ela ocorreu através de toda a pilha de invocações de métodos até que a exceção seja capturada por um bloco manipulador de exceção. Eventualmente, se tal bloco não existir em nenhum ponto da pilha de invocações de métodos, a sinalização da exceção atinge o método `main()`, fazendo com que o interpretador Java apresente uma mensagem de erro e aborte sua execução.

2.6.1 Tratamento de exceções

A captura e o tratamento de exceções em Java se dá através da especificação de blocos `try`, `catch` e `finally`, definidos através destas mesmas palavras reservadas da linguagem.

A estruturação desses blocos obedece à seguinte sintaxe:

```
try {  
    // código que inclui comandos/invocações de métodos  
    // que podem gerar uma situação de exceção.  
}  
catch (XException ex) {  
    // bloco de tratamento associado à condição de  
    // exceção XException ou a qualquer uma de suas  
    // subclasses, identificada aqui pelo objeto  
    // com referência ex  
}  
catch (YException ey) {  
    // bloco de tratamento para a situação de exceção  
    // YException ou a qualquer uma de suas subclasses  
}  
finally {  
    // bloco de código que sempre será executado após  
    // o bloco try, independentemente de sua conclusão  
    // ter ocorrido normalmente ou ter sido interrompida  
}
```

onde `XException` e `YException` deveriam ser substituídos pelo nome do tipo de exceção. Os blocos não podem ser separados por outros comandos — um erro de sintaxe seria detectado pelo compilador Java neste caso. Cada bloco `try` pode ser seguido por zero ou mais blocos `catch`, onde cada bloco `catch` refere-se a uma única exceção.

O bloco `finally`, quando presente, é sempre executado. Em geral, ele inclui comandos que liberam recursos que eventualmente possam ter sido alocados durante o processamento do bloco `try` e que podem ser liberados, independentemente de a execução ter encerrado com sucesso ou ter sido interrompida por uma condição de exceção. A presença desse bloco é opcional.

Alguns exemplos de exceções já definidas no pacote `java.lang` incluem:

ArithmeticException: indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0. A divisão de um valor real por 0 não gera uma exceção (o resultado é o valor infinito);

NumberFormatException: indica que tentou-se a conversão de uma string para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de `IllegalArgumentException`;

IndexOutOfBoundsException: indica a tentativa de acesso a um elemento de um agregado aquém ou além dos limites válidos. É a superclasse de `ArrayIndexOutOfBoundsException`, para arranjos, e de `StringIndexOutOfBoundsException`, para *strings*;

NullPointerException: indica que a aplicação tentou usar uma referência a um objeto que não foi ainda definida;

ClassNotFoundException: indica que a máquina virtual Java tentou carregar uma classe mas não foi possível encontrá-la durante a execução da aplicação.

Além disso, outros pacotes especificam suas exceções, referentes às suas funcionalidades. Por exemplo, no pacote `java.io` define-se `IOException`, que indica a ocorrência de algum tipo de erro em operações de entrada e saída. É a superclasse para condições de exceção mais específicas desse pacote, tais como `EOFException` (fim de arquivo ou stream), `FileNotFoundException` (arquivo especificado não foi encontrado) e `InterruptedIOException` (operação de entrada ou saída foi interrompida).

Uma exceção contém pelo menos uma *string* que a descreve, que pode ser obtida pela aplicação do método `getMessage()`, mas pode eventualmente conter outras informações. Por exemplo, `InterruptedIOException` inclui um atributo público do tipo inteiro, `bytesTransferred`, para indicar quantos bytes foram transferidos antes da interrupção da operação ocorrer. Outra informação que pode sempre ser obtida de uma exceção é a sequência de métodos no momento da exceção, obtível a partir do método `printStackTrace()`.

Como exceções fazem parte de uma hierarquia de classes, exceções mais genéricas (mais próximas do topo da hierarquia) englobam aquelas que são mais específicas. Assim, a forma mais genérica de um bloco `try-catch` é

```
try { ... }  
catch (Exception e) { ... }
```

pois todas as exceções são derivadas de `Exception`. Se dois blocos `catch` especificam exceções em um mesmo ramo da hierarquia, a especificação do tratamento da exceção derivada deve preceder aquela da mais genérica.

2.6.2 Erros e exceções de *runtime*

Exceções consistem de um caso particular de um objeto da classe `Throwable`. Apenas objetos dessa classe ou de suas classes derivadas podem ser gerados, propagados e capturados através do mecanismo de tratamento de exceções.

Além de `Exception`, outra classe derivada de `Throwable` é a classe `Error`, que é a raiz das classes que indicam situações que a aplicação não tem como ou não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer. Entre os erros definidos em Java, no pacote `java.lang`, estão `StackOverflowError` e `OutOfMemoryError`. São situações onde não é possível uma correção a partir de um tratamento realizado pelo próprio programa que está executando.

Há também exceções que não precisam ser explicitamente capturadas e tratadas. São aquelas derivadas de `RuntimeException`, uma classe derivada diretamente de `Exception`. São exceções que podem ser geradas durante a operação normal de uma aplicação para as quais o compilador Java não irá exigir que o programador proceda a algum tratamento (ou que propague a exceção, como descrito na Seção 2.6.3). Entre essas incluem-se `ArithmeticException`, `IllegalArgumentException`, `IndexOutOfBoundsException` e `NullPointerException`.

2.6.3 Propagando exceções

Embora toda exceção que não seja derivada de `RuntimeException` deva ser tratada, nem sempre é possível tratar uma exceção no mesmo escopo do método cuja invocação gerou a exceção. Nessas situações, é possível propagar a exceção para um nível acima na pilha de invocações.

Para tanto, o método que está deixando de capturar e tratar a exceção faz uso da cláusula **throws** na sua declaração:

```
void métodoQueNãoTrataExceção() throws Exception {  
    invoca.métodoQuePodeGerarExceção();  
}
```

Nesse caso, `métodoQueNãoTrataExceção()` reconhece que em seu corpo há a possibilidade de haver a geração de uma exceção mas não se preocupa em realizar o tratamento dessa exceção em seu escopo. Ao contrário, ele repassa essa responsabilidade para o método anterior na pilha de chamadas.

Eventualmente, também o outro método pode repassar a exceção adiante. Porém, pelo menos no método `main()` as exceções deverão ser tratadas ou o programa pode ter sua interpretação interrompida.

2.6.4 Definindo e gerando exceções

Exceções são classes. Assim, é possível que uma aplicação defina suas próprias exceções através do mecanismo de definição de classes.

Por exemplo, considere que fosse importante para uma aplicação diferenciar a condição de divisão por zero de outras condições de exceções aritméticas. Neste caso, uma classe `DivideByZeroException` poderia ser criada:

```
public class DivideByZeroException  
    extends ArithmeticException {  
    public DivideByZeroException() {  
        super("O denominador na divisão inteira tem valor zero");  
    }  
}
```

Neste caso, o argumento para o construtor da superclasse especifica a mensagem que seria impressa quando o método `getMessage()` fosse invocado para essa exceção. Essa é a mesma mensagem que é apresentada para um `RuntimeException` que não é capturado e tratado.

Para gerar uma condição de exceção durante a execução de um método, um objeto dessa classe deve ser criado e, através do comando `throw`, propagado para os métodos anteriores na pilha de execução:

```
public double calculaDivisao (double numerador, int denominador)
    throws DivideByZeroException {
    if (denominador == 0)
        throw new DivideByZeroException();

    return numerador / denominador;
}
```

O mesmo comando `throw` pode ser utilizado para repassar uma exceção após sua captura — por exemplo, após um tratamento parcial da condição de exceção:

```
public void usaDivisao()
    throws DivideByZeroException {
    ...
    try {
        d = calculaDivisao(x, y);
    }
    catch (DivideByZeroException dbze) {
        ...
        throw dbze;
    }
    ...
}
```

Nesse caso, a informação associada à exceção (como o seu ponto de origem, registrado internamente no atributo do objeto que contém a pilha de invocações) é preservada. Caso fosse desejado mudar essa informação, uma nova exceção poderia ser gerada ou, caso a exceção seja a mesma, o método `fillInStackTrace()` poderia ser utilizado, como em

```
throw dbze.fillInStackTrace();
```

2.7 O ambiente de Java

O ambiente de desenvolvimento de software Java, Java SDK (*Software Development Kit* — antigamente, denominado JDK), é formado essencialmente pelas classes fundamentais da linguagem Java e por um conjunto de aplicativos que permite, entre outras tarefas, realizar a compilação e a execução de programas escritos na linguagem Java. Não é um ambiente integrado de desenvolvimento, não oferecendo editores ou ambientes de programação visual. No entanto, são suas funcionalidades que permitem a operação desses ambientes.

O Java SDK contém um amplo conjunto de APIs que compõem o núcleo de funcionalidades da linguagem Java. Uma API (Application Programming Interface) é uma biblioteca formada por código pré-compilado, pronto para ser utilizado no desenvolvimento de suas aplicações.

2.7.1 Ferramentas do Java SDK

As ferramentas essenciais do ambiente de desenvolvimento de *software* Java são: o compilador Java, **javac**; o interpretador de aplicações Java, **java**; e o interpretador de applets Java, **appletviewer**.

Um programa fonte em Java pode ser desenvolvido usando qualquer editor que permita gravar textos sem caracteres de formatação. Um arquivo contendo código Java constitui uma unidade de compilação, podendo incluir comentários, declaração relacionadas a pacotes e pelo menos uma definição de classe ou interface.

O resultado dessa execução, se o programa fonte estiver sem erros, será a criação de um arquivo `Hello.class` contendo o bytecode que poderá ser executado em qualquer máquina.

Além das ferramentas essenciais, o Java SDK oferece os aplicativos de desenvolvimento **javadoc**, um gerador de documentação para programas Java; **jar**, um manipulador de arquivos comprimidos no formato Java Archive, que opera juntamente com **extcheck**, o verificador de arquivos nesse formato; **jdb**, um depurador de programas Java; **javap**, um disassembler de classes Java; e **javah**, um gerador de arquivos header para integração a código nativo em C.

Java oferece também aplicativos para o desenvolvimento e execução de aplicações Java em plataformas de objetos distribuídos (ver Capítulo 5). Há também ferramentas para permitir o desenvolvimento de aplicações distribuídas, incorporando também o conceito de assinaturas digitais. A ferramenta **keytool** gerencia chaves e certificados; **jarsigner** gera e verifica assinaturas associadas a arquivos Java; e **policytool** é uma interface gráfica para gerenciar arquivos que determinam a política de segurança do ambiente de execução.

2.7.2 Geração de código portátil

Um dos grandes atrativos da plataforma tecnológica Java é a portabilidade do código gerado. Esta portabilidade é atingida através da utilização de *bytecodes*. **Bytecode** é um formato de código intermediário entre o código fonte, o texto que o programador consegue manipular, e o código de máquina, que o computador consegue executar.

Na plataforma Java, o *bytecode* é interpretado por uma máquina virtual Java. A portabilidade do código Java é obtida à medida que máquinas virtuais Java estão disponíveis para diferentes plataformas. Assim, o código Java que foi compilado em uma máquina pode ser executado em qualquer máquina virtual Java, independentemente de qual seja o sistema operacional ou o processador que executa o código.

A Máquina Virtual Java (JVM) é, além de um ambiente de execução independente de plataforma, uma máquina de computação abstrata. Programas escritos em Java e que utilizem as funcionalidades definidas pelas APIs dos pacotes da plataforma Java executam nessa máquina virtual.

Uma das preocupações associadas a execuções nessas máquinas virtuais é oferecer uma arquitetura de segurança para prevenir que *applets* e aplicações distribuídas executem fora de seu ambiente seguro (*sandbox*) a não ser quando assim habilitados. Um *framework* de segurança é estabelecido através de funcionalidades dos pacotes `java.security` e seus subpacotes `java.security.acl`, `java.security.cert`, `java.security.interfaces` e `java.security.spec`.

A máquina virtual Java opera com o carregamento dinâmico de classes, ou seja, *bytecodes* são carregados pela máquina virtual Java à medida que são solicitados pela aplicação.

Em uma aplicação operando localmente, o carregador de classes da máquina virtual procura por essas classes nos (sub-)diretórios especificados a partir da variável do sistema `CLASSPATH`. Se

encontrada, a classe é carregada para a máquina virtual e a operação continua. Caso contrário, a exceção `ClassNotFoundException` é gerada.

O carregamento do código de uma classe para a JVM é realizado pelo *class loader* da máquina virtual. O *class loader*, em si uma classe Java que provê essa funcionalidade, deve obedecer a uma política de segurança estabelecida para aquela máquina virtual.

O estabelecimento de uma política de segurança deve obedecer a um modelo de segurança específico. No modelo de segurança estabelecido a partir do JDK 1.2 (*JDK security specification*), todo código sendo carregado para uma máquina virtual Java requer o estabelecimento de uma política de segurança, visando evitar que algum objeto realize operações não-autorizadas na máquina local. Com a inclusão do conceito de política de segurança, é possível estabelecer permissões diferenciadas para as aplicações.

A política de segurança padrão é estabelecida em um arquivo do sistema, `java.policy`, localizado no diretório `<java_home>/lib/security/`. Cada usuário pode estabelecer adições a essa política através da criação de um arquivo particular de estabelecimento de política, `.java.policy`, em seu diretório *home*. Por exemplo, para permitir conexões soquete de qualquer máquina com origem no domínio `unicamp.br` a portas não-notáveis, o arquivo `.java.policy` deveria incluir

```
grant {  
    permission java.net.SocketPermission  
        "*.unicamp.br:1024-", "accept,connect";  
};
```

A sintaxe para o arquivo de políticas (*Policy files syntax*) permite estabelecer domínios de permissão com base na origem do código ou na sua assinatura. A ferramenta `policytool` permite criar um arquivo de políticas através de uma interface gráfica.

Para enforçar essas políticas alternativas de segurança, um `SecurityManager` deve ser criado. *Applets*, por *default*, utilizam um `SecurityManager` estabelecido pelo navegador em cujo contexto estão executando. Outras aplicações devem criar explicitamente esse objeto. Por exemplo, para criar um `SecurityManager` padrão para aplicações usando RMI (Seção 5.4.2), a seguinte linha de código deveria ser incluída antes de executar qualquer operação envolvendo classes remotas:

```
System.setSecurityManager(new RMISecurityManager());
```

Se o cliente RMI estiver em um *applet*, então não é necessário criar um `SecurityManager`, uma vez que o próprio navegador estabelece a política de segurança para *applets* remotos.

2.7.3 Desenvolvimento de aplicações

Aplicações Java são programas autônomos, cujo código gerado a partir de um programa fonte pode ser interpretado diretamente pela Máquina Virtual Java. Como tudo em Java está estruturado através de classes e objetos, para desenvolver uma aplicação é preciso desenvolver pelo menos uma classe que contenha um método denominado `main`.

Assim, uma classe que vá estabelecer o ponto de partida para a execução de uma aplicação na JVM deve conter pelo menos esse método. Esse exemplo clássico define uma aplicação que simplesmente envia uma *string* para a saída padrão, identificada pelo objeto público `System.out`:

```
1 public class Hello {  
2     public static void main(String[] args) {  
3         System.out.println("Oi!");  
4         System.exit(0);  
5     }  
6 }
```

Uma vez que o programa tenha sido salvo em um arquivo com extensão `.java`, é preciso compilá-lo. Para compilar um programa Java, a ferramenta oferecida pelo *kit* de desenvolvimento Java é o compilador Java, `javac`. Na forma mais básica de execução, o `javac` é invocado da linha de comando tendo por argumento o nome do arquivo com o código Java a ser compilado:

```
> javac Hello.java
```

A unidade de compilação é o arquivo com extensão `.java`; esse arquivo pode conter a definição de várias classes, mas apenas uma delas pode ser pública. A classe pública em um arquivo fonte define o nome desse arquivo, que obrigatoriamente deve ter o mesmo nome dessa classe.

Para cada definição de classe na unidade de compilação, o compilador Java irá gerar um arquivo com *bytecodes* com a extensão `.class`, tendo como nome o próprio nome da classe.

Uma vez que um programa Java tenha sido compilado e esteja pronto para ser executado, isto se dá através do comando `java` — o interpretador Java, que ativa a máquina virtual Java, carrega a classe especificada e ativa seu método `main`.

Por exemplo, para interpretar o arquivo `Hello.class` contendo o *bytecode* correspondente ao código fonte do arquivo `Hello.java`, utiliza-se a linha de comando

```
> java Hello
```

Observe que a extensão `.class` não é incluída nessa linha de comando — se o for, uma mensagem de erro será gerada, pois para a máquina virtual Java o caráter `'.'` está associado à definição de uma hierarquia de pacotes.

Se a máquina virtual Java do interpretador não encontrar um método de nome `main` com a assinatura correta (`public`, `static`, `void` e com um argumento do tipo `String[]`) na classe especificada, uma exceção será gerada em tempo de execução:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Essa mensagem pode ser emitida pela ausência completa de um método `main` na classe ou por uma declaração incorreta para o método.

Se um programa Java for desenvolvido como *applet* (ver Seção 4.3), ele não poderá ser executado diretamente através do interpretador Java mas sim através da ativação de uma página HTML. Para executar esse tipo de aplicação pode-se utilizar um navegador; porém, o ambiente de desenvolvimento Java oferece a aplicação `appletviewer` que extrai da página HTML apenas o espaço de exibição do *applet* e permite controlar sua execução através de comandos em sua interface gráfica.

Capítulo 3

Uso das classes da API padrão de Java

Um dos grandes atrativos da programação orientada a objetos é a possibilidade de adaptar funcionalidades oferecidas em classes existentes às necessidades de cada aplicação. Java, não diferentemente, oferece essa facilidade, aliando-a ao oferecimento de um amplo conjunto de classes organizadas nos pacotes da API padrão.

As classes que compõem o núcleo de funcionalidades Java estão organizadas em pacotes, grupos de classes, interfaces e exceções afins ou de uma mesma aplicação. Entre os principais pacotes oferecidos como parte do núcleo Java estão: `java.lang`, `java.util`, `java.io`, `java.awt`, `java.applet` e `java.net`.

Observe que esses nomes seguem a convenção Java, pela qual nomes de pacotes (assim como nomes de métodos) são grafados em letras minúsculas, enquanto nomes de classes têm a primeira letra (de cada palavra, no caso de nomes compostos) grafada com letra maiúscula.

3.1 Funcionalidades básicas

O pacote **java.lang** contém as classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa Java.

A classe `Object` expressa o conjunto de funcionalidades comuns a todos os objetos Java, sendo a raiz da hierarquia de classes Java.

As classes `Class` e `ClassLoader` representam, respectivamente classes Java e o mecanismo para carregá-las dinamicamente para a Máquina Virtual Java.

A classe `String` permite a representação e a manipulação de *strings* cujo conteúdo não pode ser modificado. Para manipular *string* modificáveis — por exemplo, através da inserção de um caráter na *string* — a classe `StringBuffer` é oferecida.

A classe `Math` contém a definição de métodos para cálculo de funções matemáticas (trigonométricas, logarítmicas, exponenciais, etc.) e de constantes, tais como `E` e `PI`. Todos os métodos e constantes definidos nessa classe são estáticos, ou seja, para utilizá-los basta usar como prefixo o nome da classe.

O pacote oferece também um conjunto de classes *wrappers*. Um objeto de uma classe *wrapper* contém um único valor de um tipo primitivo da linguagem, permitindo assim estabelecer uma ponte entre valores literais e objetos. Essas classes são `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` e `Double`. Além dos métodos para obter o valor associado ao objeto de cada

uma dessas classes, métodos auxiliares como a conversão de e para *strings* são suportados.

As classes `System`, `Runtime` e `Process` permitem interação da aplicação Java com o ambiente de execução. Por exemplo, a classe `System` tem três atributos públicos e estáticos associados aos arquivos padrão de um sistema operacional: `System.in`, para a entrada padrão; `System.out`, para a saída padrão; e `System.err`, para a saída padrão de erros. Adicionalmente, as classes `Thread` e `ThreadGroup`, assim como a interface `Runnable`, dão suporte à execução de múltiplas linhas de execução que podem ser associadas a um processo.

As classes que definem erros e exceções, respectivamente `Error` e `Exception`, são também definidas nesse pacote. A classe `Error` é a raiz para condições de erro não-tratáveis, tais como `OutOfMemoryError`. Já a classe `Exception` está associada à hierarquia de condições que podem ser detectados e tratados pelo programa, como `ArithmeticException` (divisão inteira por zero) e `ArrayIndexOutOfBoundsException` (acesso a elemento de arranjo além da última posição ou antes da primeira posição). Ambas as classes são derivadas de `Throwable`, também definida nesse pacote.

Sub-pacotes relacionados à `java.lang` incluem `java.lang.ref`, de referências a objetos, e o pacote `java.lang.reflect`, que incorpora funcionalidades para permitir a manipulação do conteúdo de classes, ou seja, identificação de seus métodos e campos.

Deve-se ressaltar ainda que funções matemáticas são definidas em `java.lang.Math`. Outro pacote, `java.math`, contém funcionalidades para manipular números inteiros e reais de precisão arbitrária.

3.2 Entrada e saída

Por entrada e saída subentende-se o conjunto de mecanismos oferecidos para que um programa executando em um computador consiga respectivamente obter e fornecer informação de dispositivos externos ao ambiente de execução, composto pelo processador e memória principal.

De forma genérica, havendo um dispositivo de entrada de dados habilitado, o programa obtém dados deste dispositivo através de uma operação `read()`. Similarmente, um dado pode ser enviado para um dispositivo de saída habilitado através de uma operação `write()`.

A manipulação de entrada e saída de dados em Java é suportada através de classes do pacote **java.io**. Essas classes oferecem as funcionalidades para manipular a entrada e saída de bytes, adequadas para a transferência de dados binários, e para manipular a entrada e saída de caracteres, adequadas para a transferência de textos.

Fontes de dados manipuladas como seqüências de bytes são tratadas em Java pela classe `InputStream` e suas classes derivadas. Similarmente, saídas de seqüências de bytes são tratadas por `OutputStream` e suas classes derivadas.

Aplicações típicas de entrada e saída envolvem a manipulação de arquivos contendo caracteres. Em Java, a especificação de funcionalidades para manipular esse tipo de arquivo estão contidas nas classes abstratas `Reader` (e suas classes derivadas) e `Writer` (e suas classes derivadas), respectivamente para leitura e para escrita de caracteres.

3.2.1 Transferência de texto

A classe `Reader` oferece as funcionalidades para obter caracteres de alguma fonte de dados — que fonte de dados é essa vai depender de qual classe concreta, derivada de `Reader`, está sendo utilizada. A classe `Reader` apenas discrimina funcionalidades genéricas, como o método `read()`, que retorna um `int` representando o caráter lido, e `ready()`, que retorna um `boolean` indicando, se verdadeiro, que o dispositivo está pronto para disponibilizar o próximo caráter.

Como `Reader` é uma classe abstrata, não é possível criar diretamente objetos dessa classe. É preciso criar objetos de uma de suas subclasses concretas para ter acesso à funcionalidade especificada por `Reader`.

`BufferedReader` incorpora um buffer a um objeto `Reader`. Como a velocidade de operação de dispositivos de entrada e saída é várias ordens de grandeza mais lenta que a velocidade de processamento, *buffers* são tipicamente utilizados para melhorar a eficiência dessas operações de leitura e escrita. Adicionalmente, na leitura de texto a classe `BufferedReader` adiciona um método `readLine()` para ler uma linha completa. `LineNumberReader` estende essa classe para adicionalmente manter um registro do número de linhas processadas.

`CharArrayReader` e `StringReader` permitem fazer a leitura de caracteres a partir de arranjos de caracteres e de objetos `String`, respectivamente. Assim, é possível usar a memória principal como uma fonte de caracteres da mesma forma que se usa um arquivo.

`FilterReader` é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado lido. Sua classe derivada, `PushbackReader`, incorpora a possibilidade de retornar um caráter lido de volta à sua fonte.

`InputStreamReader` implementa a capacidade de leitura a partir de uma fonte que fornece bytes, traduzindo-os adequadamente para caracteres. Sua classe derivada, `FileReader`, permite associar essa fonte de dados a um arquivo.

`PipedReader` faz a leitura a partir de um objeto `PipedWriter`, estabelecendo um mecanismo de comunicação inter-processos — no caso de Java, entre *threads* de uma mesma máquina virtual.

As funcionalidades de escrita de texto estão associadas à classe abstrata `Writer` e suas classes derivadas. Entre as funcionalidades genéricas definidas em `Writer` estão o método `write(int)`, onde o argumento representa um caráter, sua variante `write(String)` e o método `flush()`, que força a saída de dados pendentes para o dispositivo sendo acessado para escrita.

Assim como para a classe `Reader`, as funcionalidades da classe `Writer` são implementadas através de suas subclasses concretas.

`BufferedWriter` incorpora um buffer a um objeto `Writer`, permitindo uma melhoria de eficiência de escrita ao combinar várias solicitações de escrita de pequenos blocos em uma solicitação de escrita de um bloco maior.

`CharArrayWriter` e `StringWriter` permitem fazer a escrita em arranjos de caracteres e em objetos `StringBuffer`, respectivamente.

`FilterWriter` é uma classe abstrata para representar classes que implementam algum tipo de filtragem sobre o dado escrito.

`OutputStreamWriter` implementa a capacidade de escrita tendo como destino uma sequência de bytes, traduzindo-os adequadamente desde os caracteres de entrada. Sua classe derivada, `FileWriter`, permite associar esse destino de dados a um arquivo.

3.2.2 Transferência de bytes

A classe abstrata `InputStream` oferece a funcionalidade básica para a leitura de um byte ou de uma sequência de bytes a partir de alguma fonte. Os principais métodos dessa classe incluem `available()`, que retorna o número de bytes disponíveis para leitura, e `read()`, que retorna o próximo byte do dispositivo de entrada.

Como para a leitura de texto, `InputStream` é usado de fato através de uma de suas classes derivadas. `ByteArrayInputStream` permite ler valores originários de um arranjo de bytes, permitindo assim tratar uma área de memória como uma fonte de dados.

Outra funcionalidade associada à transferência de dados está relacionada à conversão de formatos, tipicamente entre texto e o formato interno de dados binários. Essa e outras funcionalidades são suportadas através do oferecimento de filtros, classes derivadas de `FilterInputStream`, que podem ser agregados aos objetos que correspondem aos mecanismos elementares de entrada e saída.

`FileInputStream` lê bytes que são originários de um arquivo em disco. Usualmente, um objeto dessa classe é usado em combinação com os filtros `BufferedInputStream` e `DataInputStream`. Outro exemplo de filtro é `PushbackInputStream`, que oferece métodos `unread()` que permitem repor um ou mais bytes de volta à sua fonte, como se eles não tivessem sido lidos.

A classe `BufferedInputStream` lê mais dados que aqueles solicitados no momento da operação `read()` para um buffer interno, melhorando a eficiência das operações subsequentes para essa mesma fonte de dados.

`DataInputStream` permite a leitura de representações binárias dos tipos primitivos de Java, oferecendo métodos tais como `readBoolean()`, `readChar()`, `readDouble()` e `readInt()`. É uma implementação da interface `DataInput`. Essa interface, também implementada pela classe `RandomAccessFile`, especifica métodos que possibilitam a interpretação de uma sequência de bytes como um tipo primitivo da linguagem Java. Adicionalmente, é também possível interpretar a sequência de bytes como um objeto `String`. Em geral, métodos dessa interface geram uma exceção `EOFException` se for solicitada a leitura além do final do arquivo. Em outras situações de erro de leitura, a exceção `IOException` é gerada.

A classe `SequenceInputStream` oferece a funcionalidade para concatenar dois ou mais objetos `InputStream`; o construtor especifica os objetos que serão concatenados e, automaticamente, quando o fim do primeiro objeto é alcançado os bytes passam a ser obtidos do segundo objeto.

A classe abstrata `OutputStream` oferece a funcionalidade básica para a transferência sequencial de bytes para algum destino. Os métodos desta classe incluem `write(int)` e `flush()`, suportados por todas suas classes derivadas.

`ByteArrayOutputStream` oferece facilidades para escrever para um arranjo de bytes interno, que cresce de acordo com a necessidade e pode ser acessado posteriormente através dos métodos `toByteArray()` ou `toString()`.

`FileOutputStream` oferece facilidades para escrever em arquivos, usualmente utilizadas em conjunção com as classes `BufferedOutputStream` e `DataOutputStream`.

`FilterOutputStream` define funcionalidades básicas para a filtragem de saída de dados, implementadas em alguma de suas classes derivadas: `BufferedOutputStream` armazena bytes em um *buffer* interno até que este esteja cheio ou até que o método `flush()` seja invocado. `DataOutputStream` é uma implementação da interface `DataOutput` que permite escrever valores de variáveis de tipos primitivos de Java em um formato binário portátil. A classe `PrintStream`

oferece métodos para apresentar representações textuais dos valores de tipos primitivos Java, através das várias assinaturas dos métodos `print()` (impressão sem mudança de linha) e `println()` (impressão com mudança de linha).

A classe `ObjectInputStream` oferece o método `readObject()` para a leitura de objetos que foram serializados para um `ObjectOutputStream`. Através desse mecanismo de serialização é possível gravar e recuperar objetos de forma transparente.

`PipedInputStream` oferece a funcionalidade de leitura de um *pipe* de bytes cuja origem está associada a um objeto `PipedOutputStream`. Já `PipedOutputStream` implementa a origem de um *pipe* de bytes, que serão lidos a partir de um objeto `PipedInputStream`. Juntos, objetos dessas classes oferecem um mecanismo de comunicação de bytes entre *threads* de uma mesma máquina virtual.

3.2.3 Manipulação de arquivos

Outro dispositivo de entrada e saída de vital importância é disco, manipulado pelo sistema operacional e por linguagens de programação através do conceito de arquivos. Um arquivo é uma abstração utilizada por sistemas operacionais para uniformizar a interação entre o ambiente de execução e os dispositivos externos a ele em um computador. Tipicamente, a interação de um programa com um dispositivo através de arquivos passa por três etapas: abertura ou criação de um arquivo; transferência de dados; e fechamento do arquivo.

Em Java, a classe `File` permite representar arquivos nesse nível de abstração. Um dos construtores desta classe recebe como argumento uma string que pode identificar, por exemplo, o nome de um arquivo em disco. Os métodos desta classe permitem obter informações sobre o arquivo. Por exemplo, `exists()` permite verificar se o arquivo especificado existe ou não; os métodos `canRead()` e `canWrite()` verificam se o arquivo concede a permissão para leitura e escrita, respectivamente; `length()` retorna o tamanho do arquivo e `lastModified()`, o tempo em que ocorreu a última modificação (em milissegundos desde primeiro de janeiro de 1970). Outros métodos permitem ainda realizar operações sobre o arquivo como um todo, tais como `delete()` e `deleteOnExit()`.

Observe que as funcionalidades de transferência seqüencial de dados a partir de ou para um arquivo não é suportada pela classe `File`, mas sim pelas classes `FileInputStream`, `FileOutputStream`, `FileReader` e `FileWriter`. Todas estas classes oferecem pelo menos um construtor que recebe como argumento um objeto da classe `File` e implementam os métodos básicos de transferência de dados suportados respectivamente por `InputStream`, `OutputStream`, `Reader` e `Writer`.

Para aplicações que necessitam manipular arquivos de forma não seqüencial (ou “direta”), envolvendo avanços ou retrocessos arbitrários na posição do arquivo onde ocorrerá a transferência, Java oferece a classe `RandomAccessFile`. Esta não é derivada de `File`, mas oferece um construtor que aceita como argumento de especificação do arquivo um objeto dessa classe. Outro construtor recebe a especificação do arquivo na forma de uma *string*. Para ambos construtores, um segundo argumento é uma *string* que especifica o modo de operação, “r” para leitura apenas ou “rw” para leitura e escrita.

Os métodos para a manipulação da posição corrente do ponteiro no arquivo são `seek(long pos)`, que seleciona a posição em relação ao início do arquivo para a próxima operação de leitura ou escrita, e `getFilePointer()`, que retorna a posição atual do ponteiro do arquivo. Além disso, o método `length()` retorna o tamanho do arquivo em bytes.

Para a manipulação de conteúdo do arquivo, todos os métodos especificados pelas interfaces `DataInput` e `DataOutput` são implementados por essa classe. Assim, é possível por exemplo usar os métodos `readInt()` e `writeInt()` para ler e escrever valores do tipo primitivo `int`, respectivamente.

3.2.4 Serialização

Sendo Java uma linguagem de programação orientada a objetos, seria de se esperar que, além das funcionalidades usuais de entrada e saída, ela oferecesse também alguma funcionalidade para transferência de objetos. O mecanismo de serialização suporta essa funcionalidade.

O processo de serialização de objetos permite converter a representação de um objeto em memória para uma sequência de bytes que pode então ser enviada para um `ObjectOutputStream`, que por sua vez pode estar associado a um arquivo em disco ou a uma conexão de rede, por exemplo.

Por exemplo, para serializar um objeto da classe `Hashtable`, definida no pacote `java.util`, armazenando seu conteúdo em um arquivo em disco, a sequência de passos é:

```
// criar/manipular o objeto
    Hashtable dados = new Hashtable();
    ...
// definir o nome do arquivo
    String filename = ...;
// abrir o arquivo de saída
    File file = new File(filename);
    if(!file.exists()){
        file.createNewFile();
    }
    FileOutputStream out = new FileOutputStream(file);
// associar ao arquivo o ObjectOutputStream
    ObjectOutputStream s = new ObjectOutputStream(out);
// serializar o objeto
    s.writeObject(dados);
```

O processo inverso, a desserialização, permite ler essa representação de um objeto a partir de um `ObjectInputStream` usando o método `readObject()`:

```
FileInputStream in = new FileInputStream(file);
ObjectInputStream s = new ObjectInputStream(in);
dados = (Hashtable) s.readObject();
```

Objetos de classes para os quais são previstas serializações e desserializações devem implementar a interface `Serializable`, do pacote `java.io`. Essa interface não especifica nenhum método ou campo — é um exemplo de uma interface *marker*, servindo apenas para registrar a semântica de serialização.

Serialização é um processo transitivo, ou seja, subclasses serializáveis de classes serializáveis são automaticamente incorporadas à representação serializada do objeto raiz. Para que o processo de desserialização possa operar corretamente, todas as classes envolvidas no processo devem ter um construtor *default* (sem argumentos).

3.3 Framework de coleções

Estruturas de dados são mecanismos para manipular coleções de elementos em um programa. O pacote `java.util` oferece algumas classes definidas na API padrão de Java que implementam funcionalidades associadas a estruturas de dados.

As classes de coleções da API Java compõem o chamado *framework* de coleções, que foi completamente redefinido a partir da versão 1.2 de Java. As classes até então existentes para a manipulação de conjuntos de objetos — `Vector`, `Stack` e `Hashtable`, atualmente denominadas “as classes de coleção históricas” — foram totalmente reprojatadas para se adequar ao novo *framework*. Elas foram mantidas por motivos de compatibilidade, embora a recomendação seja utilizar as novas classes de coleções.

Todas essas estruturas manipulam coleções de objetos, ou seja, qualquer objeto de qualquer classe de Java pode ser elemento de uma dessas coleções. No entanto, não é possível criar diretamente uma coleção de tipos primitivos, como `int` ou `double`. Para manipular coleções de tipos primitivos é necessário utilizar as classes *wrappers* (Seção 3.1).

O *framework* de coleções tem por raiz duas interfaces básicas, `Map` e `Collection`.

A interface `Map` especifica as funcionalidades necessárias para manipular um grupo de objetos mapeando chaves a valores. Entre outros, os seguintes métodos são especificados nessa interface:

- `Object put(Object key, Object value)`: associa um novo valor associado à chave especificada, retornando o valor antigo (ou `null` se não havia valor associado à chave).
- `Object get(Object key)`: retorna o valor associado à chave especificada.
- `boolean containsKey(Object key)`: indica se a chave especificada está presente na coleção.
- `int size()`: retorna a quantidade de elementos na coleção.

Se adicionalmente deseja-se que esse grupo de objetos seja manipulado de acordo com alguma ordem específica, a interface derivada `SortedMap` pode ser utilizada. Assim, além das funcionalidades acima, esta interface especifica métodos tais como:

- `Object firstKey()` para retornar o elemento da coleção com a chave de menor valor.
- `Object lastKey()` para retornar o elemento da coleção com a chave de maior valor.
- `SortedMap subMap(Object fromKey, Object toKey)` para obter o subconjunto dos elementos compreendidos entre as chaves especificadas.

Para essas interfaces, duas implementações básicas são também oferecidas nesse mesmo pacote `java.util`. A classe `HashMap` implementa o mecanismo básico de mapeamento entre chaves e valores. A classe `TreeMap` oferece a funcionalidade adicional de associar uma ordem aos elementos da coleção. Para alguns tipos de objetos, tais como `Integer`, `Double` ou `String`, uma ordem “natural” já é pré-estabelecida. Para objetos de classes da aplicação, a classe deverá implementar a interface `Comparable`.

Um objeto que implementa a interface `Collection` representa um grupo de objetos genérico, onde duplicações são permitidas. Entre os métodos básicos especificados nessa interface estão:

- `boolean add(Object element)` para adicionar o elemento especificado à coleção.
- `boolean remove(Object element)` para remover o elemento especificado da coleção.
- `boolean contains(Object element)` para verificar se a coleção contém o elemento especificado.
- `int size()` para obter a quantidade de elementos na coleção.
- `Iterator iterator()` para obter um objeto que permite varrer todos os elementos da coleção.

Duas outras interfaces, `Set` e `List`, são derivadas diretamente de `Collection`.

`Set` é uma extensão de `Collection` que não permite duplicações de objetos — nenhum método novo é introduzido, mas apenas a necessidade de garantir essa restrição. A interface `SortedSet` é uma extensão de `Set` que agrega o conceito de ordenação ao conjunto, introduzindo a especificação de métodos tais como `first()`, para obter o primeiro elemento do conjunto; `last()`, para obter o último elemento do conjunto; e `subSet()`, para obter o subconjunto com todos os elementos contidos entre os dois elementos especificados como argumentos do método.

A interface `Set` tem duas implementações já oferecidas no pacote `java.util`. A classe `HashSet` é uma implementação padrão que utiliza o valor de código *hash* para detectar e impedir duplicações. A classe `TreeSet` é uma implementação de `SortedSet` que usa uma estrutura de dados em árvore para manter ordenados os elementos do conjunto.

A interface `List` é uma extensão de `Collection` que introduz mecanismos de indexação. Além dos métodos básicos de coleções, adiciona métodos tais como `get()` para obter o elemento armazenado na posição especificada como argumento; `indexOf()` para obter a posição da primeira ocorrência do objeto especificado como argumento; e `subList()` para obter uma sublista contendo os elementos compreendidos entre as duas posições especificadas, incluindo o elemento da primeira posição mas não o da segunda.

Da mesma forma que para `Set`, há duas implementações padronizadas para a interface `List` já definidas em `java.util`. A classe `ArrayList` oferece uma implementação básica da interface, enquanto que a classe `LinkedList` oferece uma implementação otimizada para manipular listas dinâmicas, introduzindo os métodos `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()` e `removeLast()`.

Para percorrer os elementos de uma coleção no novo *framework*, um objeto que implementa a interface `Iterator` é utilizado. Essa interface especifica os métodos `hasNext()`, que retorna um valor booleano verdadeiro para indicar que há mais elementos na coleção, e `next()`, que retorna o objeto que é o próximo elemento da coleção. Adicionalmente, um método `void remove()` pode ser utilizado para retirar um elemento da coleção através da referência a um `Iterator`.

A interface `ListIterator` é uma extensão de `Iterator` que permite a varredura da coleção nas duas direções, especificando novos métodos tais como `boolean hasPrevious()` e `Object previous()`.

3.4 Extensões padronizadas

Além das funcionalidades básicas oferecidas por esses pacotes, há também APIs definidas para propósitos mais específicos compondo a extensão padronizada ao núcleo Java. Por exemplo, o **Input Method Framework** está associado ao pacote `java.awt.im` e contempla funcionalidades para manipular textos não-ocidentais usando teclados convencionais.

Funcionalidades para definir e utilizar componentes de *software* em Java são oferecidas através de **Java Beans**, um conjunto de APIs definidas através do pacote `java.beans` e seu subpacote `java.beans.context` para suportar os modelos de instrospecção, customização, eventos, propriedades e persistência desses componentes.

Uma das extensões mais significativas é definida pelo conjunto da **Java Foundation Classes**, extensão de AWT que oferece um conjunto de componentes de interface gráfica com usuário completamente portátil. Três pacotes compõem JFC: Swing, Drag and Drop e Java 2D. *Swing* define uma família de pacotes com o prefixo `javax.swing` com componentes e serviços GUI de aparência independente de plataforma. *Drag and Drop*, associado ao pacote `java.awt.dnd`, suporta o compartilhamento de dados baseado no padrão MIME, *Multipurpose Internet Mail Extension*. *Java 2D* acrescenta novas classes aos pacotes `java.awt` e `java.awt.image` para estender as funcionalidades associadas à manipulação de imagens bidimensionais e acrescenta novos pacotes `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.image.renderable` e `java.awt.print`. As funcionalidades para a construção de interfaces gráficas com usuários são apresentadas na Seção 4.

Outra funcionalidade acrescentada à Java foi a capacidade de acesso a bancos de dados relacionais usando a linguagem de consulta SQL. Através da API definida em **JDBC**, através do pacote `java.sql`, é possível enviar comandos SQL para um servidor de banco de dados e processar o retorno obtido nessa consulta. A utilização de JDBC é descrita na Seção 5.2.3.

Java IDL integra a funcionalidade da arquitetura CORBA a aplicações Java, permitindo obter inter-operabilidade e conectividade baseadas nesse padrão do OMG (*Object Management Group*). Além das classes presentes nos pacotes `org.omg.CORBA` e `org.omg.CosNaming`, alguns aplicativos são acrescentados ao ambiente de desenvolvimento e execução de Java, como o compilador de Java para IDL e um aplicativo que oferece o serviço de nomes. O desenvolvimento de aplicações usando Java IDL é descrito na Seção 5.4.3.

Capítulo 4

Desenvolvimento de aplicações gráficas

A linguagem Java oferece, dentre as funcionalidades incorporadas à sua API padrão, um extenso conjunto de classes e interfaces para o desenvolvimento de aplicações gráficas. Esse conjunto facilita a criação de saídas na forma gráfica e de interfaces gráficas com usuários (GUIs), tanto na forma de aplicações autônomas como na forma de applets.

Aplicações gráficas são criadas através da utilização de componentes gráficos, que estão agrupados em dois grandes pacotes: `java.awt` e `javax.swing`.

AWT é o *Abstract Windowing Toolkit*, sendo definido através das classes do pacote `java.awt` e seus subpacotes, tais como `java.awt.event`. Essas classes agrupam as funcionalidades gráficas que estão presentes desde a primeira versão de Java, que operam tendo por base as funcionalidades de alguma biblioteca gráfica do sistema onde a aplicação é executada.

Já o pacote `javax.swing` define uma extensão padronizada a partir de AWT que congrega componentes gráficos que utilizam exclusivamente Java (*lightweight components*), com funcionalidades e aparência independentes do sistema onde a aplicação é executada.

4.1 Apresentação gráfica

A base para tudo que é desenhado na tela de uma aplicação gráfica é o contexto gráfico, em Java um objeto da classe `Graphics` do pacote `java.awt`. São nesses objetos que linhas, retângulos, círculos, *strings*, etc. são desenhados, compondo a apresentação visual da aplicação gráfica.

Todo componente gráfico tem associado a si um contexto gráfico e alguns métodos chaves para manipulá-lo. A um objeto da classe `Component` pode-se aplicar os métodos `getGraphics()`, que permite obter uma referência para um objeto da classe `Graphics`; `paint(Graphics)`, que determina o que é feito no contexto gráfico; e `repaint()`, que solicita uma atualização no contexto gráfico.

Para determinar o conteúdo gráfico do contexto, os métodos da classe `Graphics` são utilizados. Esses métodos incluem facilidades para definição de cores, fontes e definição de figuras geométricas. O sistema de coordenadas para definir a posição do desenho no contexto tem origem no canto superior esquerdo da área gráfica, tomando valores inteiros para x e y representando quantidades de *pixels* na horizontal e na vertical, respectivamente.

A cor corrente para o contexto gráfico pode ser obtida e definida pelos métodos `getColor()` e `setColor()`, respectivamente. Esses métodos usam cores representadas por objetos da classe

`Color`, que oferece constantes para usar cores pré-definidas, métodos para definir cores especificando os níveis RGB (*red*, *green*, *blue*) ou HSB (*hue*, *saturation*, *brightness*) e métodos para manipular as cores existentes.

Similarmente, a fonte usada para a apresentação de *strings* no contexto pode ser obtida e definida através de métodos `getFont()` e `setFont()`. Fontes são representadas por objetos da classe `Font`.

Entre os diversos métodos usados para desenhar no contexto, há métodos para desenhar texto usando a fonte de texto e a cor corrente, `drawString()`; desenhar linhas retas, `drawLine()`; e desenhar diversas formas geométricas, tais como arcos, `drawArc()`; retângulos, `drawRect()`, `drawRoundRect()`, `draw3DRect()`; polígonos, `drawPolygon()`; ovais, `drawOval()`; e suas correspondentes versões preenchidas, `fillArc()`, `fillRect()`, `fillRoundRect()`, `fill3DRect()`, `fillPolygon()` e `fillOval()`.

4.2 Interfaces gráficas com usuários

Criar uma interface gráfica com usuários em Java envolve tipicamente a criação de um *container*, um componente que pode receber outros. Em uma aplicação gráfica autônoma, tipicamente o *container* usado como base é um *frame* (uma janela com bordas e barra de título), enquanto que em um *applet* o *container* base é um *panel* (uma área gráfica inserida em outra). Em aplicações complexas, qualquer tipo de combinação desses e de outros tipos de componentes é utilizado.

Após sua criação, os *containers* da aplicação recebem componentes de interface com usuários, que permitirão apresentar e receber dados aos e dos usuários. Embora seja possível posicionar esses componentes através de coordenadas absolutas em cada *container*, é recomendável que o projetista utilize sempre um gerenciador de *layout* para realizar essa tarefa. Deste modo, garante-se que a aplicação possa ser executada independentemente das características da plataforma onde será executada.

Finalmente, é preciso especificar quais devem ser os efeitos das ações dos usuários — tais como um "clique" do *mouse* ou uma entrada de texto — quando realizadas sobre cada um desses componentes. Isto se dá através da especificação de classes manipuladoras de eventos, projetadas para a aplicação. Ao associar objetos dessas classes aos componentes gráficos, o projetista determina o comportamento da aplicação.

4.2.1 Eventos da interface gráfica

No modelo de eventos suportado a partir da versão 1.1 de Java, um componente gráfico qualquer pode reconhecer alguma ação do usuário e a partir dela disparar um evento — indicando, por exemplo, que o botão do *mouse* foi pressionado ou que um texto foi modificado — que será capturado por um objeto registrado especificamente para registrar aquele tipo de evento ocorrido naquele componente.

O pacote `java.awt.event` define as diversas classes de eventos que podem ocorrer através das interfaces gráficas. Eventos gráficos são objetos derivados da classe `AWTEvent`, que por sua vez são derivados de objetos de evento genéricos definidos pela classe `EventObject`. Desta, eventos gráficos herdam o método `getSource()`, que permite identificar o objeto que deu origem ao evento.

Eventos gráficos genéricos são especializados de acordo com o tipo de evento sob consideração — por exemplo, pressionar um botão do *mouse* gera um objeto da classe `MouseEvent`. A mesma ação sobre um botão, no entanto, gera também um `ActionEvent`, enquanto que sobre o botão de minimizar na barra de título de um *frame* um `WindowEvent` seria ao invés adicionalmente gerado.

Outros eventos de interesse definidos em `java.awt.event` incluem `ItemEvent`, indicando que um item de uma lista de opções foi selecionado; `TextEvent`, quando o conteúdo de um componente de texto foi modificado; e `KeyEvent`, quando alguma tecla foi pressionada no teclado.

A resposta que uma aplicação dá a qualquer evento que ocorre em algum componente gráfico é determinada por métodos específicos, registrados para responder a cada evento. O nome e a assinatura de cada um desses métodos é determinado por uma interface Java do tipo *listener*. Assim, para responder a um `ActionEvent` será utilizado o método definido na interface `ActionListener`; para responder a um `WindowEvent`, os métodos de `WindowListener`. Similarmente, há interfaces `ItemListener`, `TextListener` e `KeyListener`. Os eventos do *mouse* são manipulados através de métodos especificados em duas interfaces, `MouseListener` (para ações sobre o *mouse*) e `MouseMotionListener` (para tratar movimentos realizados com o *mouse*).

Apesar de existir um grande número de eventos e possibilidades de resposta que a aplicação poderia dar a cada um deles, cada aplicação pode especificar apenas aqueles para os quais há interesse que sejam tratados. Para os eventos que o projetista da aplicação tem interesse de oferecer uma reação, ele deve definir classes manipuladoras de eventos (*handlers*), implementações de cada *listener* correspondente ao tipo de evento de interesse.

Para interfaces *listener* que especificam diversos métodos, **classes adaptadoras** são definidas. Essas classes adaptadoras são classes abstratas definidas no pacote de eventos AWT, com nome `XX-XXAdapter`, onde `XXX` seria o prefixo correspondente ao tipo de evento de interesse. Assim, para que a aplicação use uma classe adaptadora para tratar os eventos do tipo `WindowEvent`, por exemplo, uma classe que estende a classe `WindowAdapter` pode ser definida. Nessa classe derivada, os métodos relacionados aos eventos de interesse são redefinidos (pelo mecanismo de *overriding*). Como a classe adaptadora implementa a interface correspondente `WindowListener`, assim o fará a classe derivada. Os métodos não redefinidos herdarão a definição original, que simplesmente ignora os demais eventos.

A API de eventos AWT de Java define, além de `WindowAdapter`, as classes adaptadoras `MouseAdapter`, `MouseMotionAdapter` e `KeyAdapter`.

Uma vez que uma classe *handler* tenha sido criada para responder aos eventos de um componente, é preciso associar esse componente ao objeto *handler*. Para tanto, cada tipo de componente gráfico oferece métodos na forma `addXXXListener(XXXListener l)` e `removeXXXListener(XXXListener l)`, onde `XXX` está associado a algum evento do tipo `XXXEvent`. Por exemplo, para o componente `Window` (e por herança para todas os componentes derivados desse) são definidos os métodos `public void addWindowListener(WindowListener l)` e `public void removeWindowListener(WindowListener l)`, uma vez que nesse tipo de componente é possível ocorrer um `WindowEvent`.

Manipuladores de eventos de baixo nível

Eventos de *mouse* são tratados pelos métodos definidos em dois *listeners* distintos definidos em `java.awt.event`. A interface `MouseListener` especifica os métodos para os eventos de maior

destaque na interação do usuário com o mouse. A classe abstrata `MouseListener` oferece implementações vazias para todos esses métodos.

Um objeto associado a um evento do *mouse* descreve eventos notáveis ocorridos com o *mouse*, como o fato de ter entrado ou saído na área de interesse (uma janela, tipicamente), um “clique” ocorrido, se alguma tecla de modificação (ALT, SHIFT ou CONTROL) estava pressionada e a posição na qual o evento ocorreu.

A interface `MouseMotionListener` especifica métodos para manipular eventos de movimentação do *mouse*. A classe abstrata `MouseMotionAdapter` é uma adaptadora para esse *listener*. Nesse caso, qualquer movimentação do *mouse* é relatada, diferenciando entre eventos da forma `MOUSE_MOVED`, para movimentos com o *mouse* livre, e `MOUSE_DRAGGED`, para movimentos do *mouse* com algum botão pressionado.

A interface `KeyListener` especifica os métodos necessários para detectar e tratar eventos de teclado. São eles: `keyPressed()`, para indicar que a tecla foi pressionada; `keyReleased()`, para indicar que a tecla foi solta; e `keyTyped()`, para indicar que uma tecla que não de controle foi pressionada e solta. Todos os métodos recebem um objeto `KeyEvent` como parâmetro. Esse objeto indica o tipo de evento ocorrido (`KEY_PRESSED`, `KEY_TYPED` ou `KEY_RELEASED`), o código da tecla e se havia modificadores associados. Como essa interface *listener* especifica mais de um método, uma classe adaptadora, `KeyAdapter`, é oferecida.

Manipuladores de eventos de alto nível

Raramente uma aplicação está preocupada em tratar eventos no nível tão básico como qualquer movimento do *mouse* ou tecla pressionada. Java oferece um conjunto de funcionalidades que filtram esses eventos para outros mais próximos da visão da aplicação, como o pressionar de um botão ou entrar texto em um campo.

A interface `ActionListener` especifica o método `actionPerformed()` para tratar eventos do tipo ação. Objetos da classe `ActionEvent` representam eventos associados a uma ação aplicada a algum componente AWT.

Uma descrição (na forma de uma *string*) da ação pode ser obtida através do método `getActionCommand()`. Dessa forma, se mais de um componente compartilha um mesmo *listener*, esse método permite diferenciar qual componente foi o gerador da ação.

Outro atributo que está representado em um objeto evento diz respeito aos modificadores, teclas que podem estar apertadas simultaneamente à ocorrência do evento para requisitar um tratamento alternativo. A classe `ActionEvent` define o método `getModifiers()`, que retorna um valor inteiro que pode ser analisado em relação a quatro constantes da classe, `ALT_MASK`, `CTRL_MASK`, `META_MASK` e `SHIFT_MASK`.

Um objeto de uma classe que implemente `ActionListener` deve ser associado a um componente através do método `addActionListener()`, que é definido para componentes do tipo botão, listas e campos de texto.

Quando um evento desse tipo ocorre, o objeto registrado é notificado. O método `actionPerformed()` é então invocado, recebendo como argumento o objeto que representa o evento.

Eventos associados à manipulação de janelas são definidos na interface `WindowListener`. Essa interface especifica os seguintes métodos:

`windowOpened()` trata evento que é apenas gerado quando a janela é criada e aberta pela primeira

vez

windowClosing() usuário solicitou que a janela fosse fechada, seja através de um menu do sistema ou através de um botão da barra de títulos ou através de uma sequência de teclas do sistema.

windowClosed() trata evento que indica que a janela foi fechada.

windowIconified() trata evento que indica que a janela foi iconificada.

windowDeiconified() trata evento indicando que o ícone da janela foi aberto.

windowActivated() trata evento que indica que a janela ganhou o foco do teclado e tornou-se a janela ativa.

windowDeactivated() trata evento que indica que a janela deixou de ser a janela ativa, provavelmente porque outra janela foi ativada.

Todos esses métodos têm como argumento um objeto da classe `WindowEvent`, que representa um evento de janela.

Como a interface `WindowListener` especifica diversos métodos, uma classe adaptadora abstrata, `WindowAdapter`, é fornecida com implementações vazias de todos os métodos.

Para implementar a interface `ItemListener` é preciso definir a implementação do método `itemStateChanged()`, que recebe como argumento um objeto evento do tipo `ItemEvent`. Um `ItemEvent` sinaliza que um elemento de algum tipo de lista de opções foi selecionado ou desselecionado. Componentes que emitem esse tipo de evento implementam a interface `ItemSelectable`.

A interface `TextListener` especifica o método que trata eventos do tipo texto. Um objeto da classe `TextEvent` está relacionado a um evento indicando que o usuário modificou o valor de texto que aparece em um `TextComponent` — qualquer modificação no texto apresentado no componente gera esse evento. Um método apenas é especificado na interface, `textValueChanged(TextEvent e)`. O corpo desse método determina o que deve ser executado quando o texto em algum componente de texto é alterado.

4.2.2 Componentes gráficos

A criação de uma interface gráfica com usuários utiliza tipicamente uma série de componentes pré-estabelecidos para tal fim, tais como rótulos (*labels*, textos que são apresentados mas não alteráveis) e botões, um componente que pode ser ativado pelo usuário através do *mouse*.

Java oferece um amplo conjunto de classes pré-definidas e re-utilizáveis que simplificam o desenvolvimento de aplicações gráficas. A raiz desse conjunto de classes gráficas no pacote `java.awt` é a classe abstrata `Component`, que representa qualquer objeto que pode ser apresentado na tela e ter interação com usuários. Similarmente, o pacote `Swing` tem por raiz a classe `JComponent`, que (indiretamente) é em si uma extensão de `Component`.

Os métodos da classe `Component` definem funcionalidades que dizem respeito à manipulação de qualquer componente gráfico em Java. O método `getSize()` permite obter a dimensão do componente, expressa na forma de um objeto da classe `Dimension` — um objeto dessa classe tem

campos públicos `width` e `height` que indicam respectivamente as dimensões horizontais e verticais do componente em *pixels*. Já o método `setSize()` permite definir a dimensão do componente.

Outro grupo de métodos importantes dessa classe é composto pelos métodos que permitem determinar os manipuladores de eventos que são relevantes para qualquer tipo de objeto gráfico, como os eventos de *mouse* (`addMouseListener()`, `addMouseMotionListener()`, `removeMouseListener()`, `removeMouseMotionListener()`) e de teclado (`addKeyListener()`, `removeKeyListener()`).

Subclasses de `Component` que são amplamente utilizadas no desenvolvimento de aplicações gráficas incluem os *containers*, um objeto gráfico que pode conter outros componentes, e os componentes de interface gráfica com usuários.

Para manipular texto em uma interface gráfica há componentes dos tipos campos de texto e áreas de texto, que permitem a entrada e/ou apresentação de texto em uma janela gráfica, respectivamente para uma única linha ou para diversas linhas de texto.

Quando a aplicação oferece ao usuário uma série de opções pré-estabelecidas, os componentes tipicamente utilizados em interfaces gráficas são de escolha, que pode ser única, quando a seleção de uma opção exclui as demais, ou múltipla, quando várias opções podem ser selecionadas simultaneamente.

Rótulos

Rótulos são campos de texto incluídos em uma janela gráfica com o único objetivo de apresentação. Dessa forma, rótulos não reagem a interações com usuários.

A classe `Label` define um componente de AWT que apresenta uma única linha de texto não-alterável. Um dos atributos associados a um `Label` é o texto a ser apresentado, que pode ser especificado em um dos construtores ou através do método `setText()`. O método `getText()` retorna o valor desse atributo. Outro atributo de `Label` é o alinhamento, que pode ser `Label.CENTER`, `Label.LEFT` ou `Label.RIGHT`. O alinhamento pode ser definido em um dos construtores (juntamente com o texto) ou através do método `setAlignment()`. O método `getAlignment()` retorna o valor desse atributo.

A classe `JLabel` define o componente Swing para implementar rótulos. Além de oferecer facilidades equivalentes àsquelas de `Label`, `JLabel` oferece a possibilidade de definir o alinhamento vertical (o padrão é centralizado) e de usar um ícone gráfico como rótulo, em adição ou em substituição a um texto.

Botões

Um botão corresponde a um componente gráfico que pode ser “pressionado” através de um “clique” do *mouse*, podendo assim responder à ativação por parte de um usuário.

A classe `Button` implementa um botão no AWT. Para criar um botão com algum rótulo, basta usar o construtor que recebe como argumento uma *string* com o texto do rótulo. Alternativamente, pode-se usar o construtor padrão (sem argumentos) e definir o rótulo do botão com o método `setLabel()`. O rótulo que foi definido para o botão pode ser obtido do método `getLabel()`.

Em Swing, o componente que define um botão é `JButton`. Além de poder ser definido com rótulos de texto, um botão Swing pode conter também ícones.

Para manipular o evento associado à seleção de um botão, é preciso definir uma classe que realize o tratamento de um evento do tipo ação implementando um `ActionListener`.

Campos de texto

Um campo de texto é uma área retangular, de tamanho correspondente a uma linha, que pode ser utilizada para apresentar ou adquirir *strings* do usuário.

A classe `TextField` é uma extensão da classe `TextComponent` que implementa um campo de texto no *toolkit* AWT. Os construtores dessa classe permitem a criação de um campo de texto que pode estar vazio ou conter algum texto inicial. O número de colunas no campo também pode ser especificado através dos construtores.

O componente correspondente a esse em Swing é `JTextField`. Além de funcionalidades associadas a `TextField`, esse componente oferece outras que permitem definir campos de texto customizados.

Quando o usuário entra um texto em um campo de texto e tecla ENTER, é gerado um evento do tipo ação. Para indicar o objeto manipulador para esse evento, os métodos `addActionListener()` e `removeActionListener()` são utilizados. Ambos têm como argumento um objeto `ActionListener`. Eventos relativos à modificação de um valor em um campo de texto (sem pressionar a tecla ENTER) estão associados a eventos do tipo texto. Desse modo, é possível monitorar se alguma modificação ocorreu no campo de texto.

É possível não ecoar a apresentação dos caracteres de entrada para a tela definindo um caráter alternativo que deve ser apresentado. Em AWT, utiliza-se o método `setEchoChar()` da classe `TextField`. Em Swing, para tal fim deve ser utilizado o componente `JPasswordField`.

Áreas de texto

Uma área de texto permite apresentar e opcionalmente editar textos de múltiplas linhas. É possível adicionar texto a uma área através do método `append()`, que recebe uma *string* como argumento.

Em AWT, a classe `TextArea` oferece essa funcionalidade, sendo também uma extensão da classe `TextComponent`. Assim como para um campo de texto de uma linha, o texto apresentado inicialmente e o número de colunas visíveis pode ser definido em construtores. Adicionalmente, o número de linhas visíveis pode ser definido no momento da construção do objeto. Para a apresentação de textos que ocupam uma área maior que a visível, um construtor tem um argumento que indica se e quantas barras de rolagem estarão presentes na área de texto. Os valores possíveis, definidos como constantes estáticas dessa classe, são `SCROLLBARS_BOTH` (barras de rolagem nas direções vertical e horizontal), `SCROLLBARS_HORIZONTAL_ONLY` (apenas na horizontal), `SCROLLBARS_VERTICAL_ONLY` (apenas na vertical) e `SCROLLBARS_NONE` (sem barras de rolagem).

O componente correspondente a esse em Swing é o `JTextArea`. Em Swing, o comportamento *default* para a área de texto é não ter barras de rolagem — se isso for necessário, a aplicação deve usar um componente decorador `JScrollPane`. Outras diferenças dizem respeito à habilidade de mudar de linha (automático em AWT) e à geração de eventos do tipo texto (não presente em Swing).

O *framework* Swing oferece, no entanto, outros componentes mais elaborados que também manipulam áreas de texto. Componentes das classes `JEditorPane` e `JTextPane` conseguem mani-

pular vários tipos de conteúdo, como por exemplo texto em HTML.

Componentes de seleção

Java apresenta três mecanismos básicos para definir um componente que permite ao usuário a seleção de opções apresentadas: *drop-down*, lista ou caixa de seleção.

Um componente de interação em **drop-down** é aquele que deixa visível uma única opção (aquela que está selecionada) ao usuário, mas que permite que esse visualize e selecione as outras opções através da ativação de um mini-botão associado ao componente. Quando esse tipo de componente permite também a entrada de valores em um campo de texto combinado à lista, é usualmente chamado de *combo box*. Em componentes do tipo *combo box*, a seleção é restrita a uma única opção, com uma lista de itens em apresentação *drop-down* onde apenas o elemento selecionado é visível. Os demais itens podem ser apresentados para seleção através de interação com o *mouse*.

Em AWT, a funcionalidade de lista em *drop-down* é suportada através da classe `Choice`, que permite definir uma lista de itens não-editáveis. Para essa classe, apenas o construtor padrão é oferecido. Os principais métodos para manipular objetos dessa classe são: `add(String s)`, que permite adicionar itens à lista; `getSelectedIndex()`, que retorna a posição numérica do item selecionado na lista (o primeiro elemento tem índice 0); `getSelectedItem()`, que retorna o rótulo do item selecionado na lista; e `getItem(int index)`, retorna o rótulo do item cujo índice foi especificado.

A classe `Choice` implementa a interface `ItemSelectable`, o que permite associar objetos dessa classe a um manipulador de eventos do tipo `ItemListener`.

Em Swing, o mecanismo de lista em *drop-down* é suportado pela classe `JComboBox`, que pode ser configurado para ser editável ou não.

A segunda opção para seleção de itens é oferecida através de **listas**, onde vários itens são apresentados simultaneamente em uma janela. Na construção desse tipo de componente, o programador pode estabelecer se a seleção será exclusiva ou múltipla. Um componente do tipo lista oferece a funcionalidade de apresentar uma lista de opções dos quais um ou vários itens podem ser selecionados.

Em AWT, a classe `List` oferece essa funcionalidade. Além do construtor padrão, essa classe oferece um construtor que permite especificar quantas linhas da lista serão visíveis. Um terceiro construtor permite especificar um valor booleano indicando se o modo de seleção de vários itens está habilitado (`true`) ou não. Se não especificado, apenas um item pode ser selecionado da lista.

Os principais métodos dessa classe incluem: `add(String item)`, que acrescenta o item à lista; `add(String item, int index)`, que acrescenta um item à lista na posição especificada; `getSelectedIndex()`, que retorna a posição numérica do item selecionado na lista, sendo que o primeiro item está na posição 0; `getSelectedItem()`, que retorna o rótulo do item selecionado na lista; e `getItem(int index)`, que retorna o rótulo do item cujo índice foi especificado. Há ainda métodos para a manipulação de seleções múltiplas: `getSelectedIndexes()`, que retorna um arranjo de inteiros correspondendo às posições selecionadas, e `getSelectedItems()`, que retorna um arranjo de `String` correspondente aos rótulos dos itens selecionados.

Assim como a classe `Choice`, a classe `List` implementa a interface `ItemSelectable`, que permite associar um objeto dessa classe a um manipulador de eventos do tipo `Item`, ou seja, a um objeto `ItemListener`. Esse tipo de evento é gerado quando o item é alvo de um único toque do *mouse*. Além disso, um item da lista pode reagir a um toque duplo do *mouse* através da geração de um evento do tipo `Ação`, usando para o tratamento do evento nesse caso um manipulador do

tipo `ActionListener`. O tratamento de múltiplas seleções através de eventos de itens não é amigável através dessa interface, pois a cada seleção um novo evento é gerado. Por esse motivo, a documentação da API Java recomenda que esse tipo de seleção múltipla em AWT seja efetivada através de algum controle externo, como um botão.

Em Swing, funcionalidade similar à apresentada por `List` é oferecida pela classe `JList`, embora algumas diferenças devam ser observadas. Por exemplo, ao contrário de `List`, um `JList` não suporta barras de rolagem diretamente, o que deve ser acrescentado através de um objeto decorador `JScrollPane`.

A terceira opção para selecionar itens em uma interface gráfica é utilizar componentes do tipo **caixas de seleção**, que podem ser independentemente selecionadas ou organizadas em grupos de caixas de seleção das quais apenas uma pode estar selecionada. Uma caixa de seleção oferece um mecanismo de seleção em que uma opção pode assumir um de dois valores, selecionada ou não-selecionada. Diz-se que tais componentes têm comportamento booleano.

Em AWT, essa funcionalidade é suportada pela classe `Checkbox`. Além do construtor padrão, essa classe oferece construtores que permitem expressar um rótulo (`String`) para o botão e o estado inicial (`boolean`). O método `getState()` permite obter o estado da seleção; o método `getLabel()`, o rótulo associado à caixa de seleção. `Checkbox` implementa a interface `ItemSelectable`, permitindo tratar reações a eventos do tipo item através de um `ItemListener`.

Em Swing, essa funcionalidade é suportada de forma similar pela classe `JCheckBox`.

É também possível definir grupos de caixas de seleção das quais apenas uma pode estar selecionada em um dado instante, constituindo assim opções exclusivas entre si. Em AWT, esse tipo de funcionalidade é oferecido através do conceito de um grupo de *checkboxes*, suportado através da classe `CheckboxGroup`. Um `CheckboxGroup` não é um componente gráfico, como um *container*, mas sim um agrupamento lógico entre os componentes gráficos do tipo `Checkbox`, que devem especificar na sua construção ou através do método `setCheckboxGroup()` a qual grupo pertencem, se for o caso de pertencer a algum.

Funcionalidade similar a essa é oferecida em Swing através das classes `JRadioButton` e `ButtonGroup`.

4.2.3 Containers

Difícilmente uma aplicação gráfica é composta por um único componente, mas sim por vários componentes inter-relacionados. Para este tipo de aplicação, um componente fundamental é a área onde os demais componentes da aplicação estarão dispostos. Um componente que pode conter outros componentes é denominado um *container*.

Em Java, a classe `Container` é a classe abstrata que define as funcionalidades básicas associadas a um *container*, tais como adicionar e remover componentes, o que é possível através dos métodos `add()` e `remove()`, respectivamente. É possível também estabelecer qual a estratégia de disposição de componentes no *container*, ou seja, qual o método de gerência de *layout*, através do método `setLayout()`.

`Window` é uma classe derivada de `Container` cujos objetos estão associados a janelas. Cada objeto `Window` é uma janela independente em uma aplicação, embora a essa janela não estejam associadas as funcionalidades usualmente oferecidas por gerenciadores de janela. Raramente um objeto desse é usado diretamente, mas objetos dessa classe são muito utilizados através de suas subclasses, tais como `Frame`.

Outra classe derivada de `Container` de extensa aplicação é `Panel`, que define uma área de composição de componentes contida em alguma janela. A classe `Applet` é uma extensão de `Panel` que permite criar *applets* (Seção 4.3).

Embora a classe `JComponent`, raiz da hierarquia para todos os componentes do novo *framework* para interfaces gráficas de Java, seja derivada da classe `Container`, não se pode acrescentar diretamente um componente gráfico a qualquer componente Swing. Para as classes de Swing que correspondem a *containers* no sentido definido em AWT, ou seja, às quais podem ser acrescentados outros componentes, deve-se obter uma referência ao objeto `Container` através do método `getContentPane()`.

4.2.4 Janelas

Janelas são áreas retangulares que podem ser exibidas em qualquer parte da tela gráfica de um usuário. Em AWT, objetos da classe `Window` estão associados a essa funcionalidade, sendo que nessa classe são definidos os métodos aplicáveis a qualquer tipo de janela. O *framework* Swing oferece a classe `JWindow`, derivada da classe `Window`, com esse mesmo tipo de funcionalidade porém adaptada ao novo *framework*.

Um objeto `Window`, no entanto, corresponde a uma janela sem nenhum tipo de “decoração”. É mais comum que aplicações gráficas criem janelas através de uma das classes derivadas de `Window`, tais como um *frame* ou uma janela de diálogo. Convém destacar da classe `Window` os métodos `show()`, para trazer a janela para a frente da tela; `dispose()`, para eliminar uma janela e liberar os recursos usados por ela; e `addWindowListener()` e `removeWindowListener()` para associar ou remover objetos manipuladores de eventos de janelas.

Um *frame* agrega a uma janela as funcionalidades mínimas para que ela possa ser identificada e manipulada pelo usuário da aplicação, tais como uma borda e a barra de títulos com botões de controle da janela. Um *frame* é uma janela com propriedades adicionais que a habilitam a ter uma “vida independente” no ambiente de janelas gráficas. Para que um *frame* seja apresentado pela aplicação, um objeto desse tipo de classe deve ser criado e algumas de suas propriedades básicas — tais como sua dimensão e o fato de estar visível ou não — devem ser estabelecidas.

Em AWT, *frames* estão associados a objetos da classe `Frame`. Esse exemplo ilustra as operações essenciais para que um *frame* AWT seja criado e exibido em um ambiente gráfico. O resultado da execução desse código é a criação e exibição de uma janela gráfica com conteúdo vazio:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class Janela extends Frame {
4      class WindowHandler extends WindowAdapter {
5          public void windowClosing(WindowEvent we) {
6              dispose();
7              System.exit(0);
8          }
9          public void windowActivated(WindowEvent we) {
10             we.getWindow().validate();
11         }
12     }
```



```
13     public Janela() {
14         this("Janela");
15     }
16     public Janela(String titulo) {
17         setTitle(titulo);
18         setSize(320,200);
19         addWindowListener(new WindowHandler());
20     }
21     public static void main(String[] args) {
22         Janela j = new Janela();
23         j.setVisible(true);
24     }
25 }
```

Se o tamanho da janela (em *pixels*, horizontal e vertical) não for definido (método `setSize()`), então o *default* ($x=0$, $y=0$) é assumido, criando uma janela de dimensões nulas. Mesmo que o tamanho seja definido, a janela quando criada é por padrão invisível — para exibi-la é preciso invocar o método `setVisible()`.

`Frame` é uma extensão da classe `Window` que, por sua vez, é uma concretização de `Container`, que é uma extensão de `Component`. Assim, é importante tomar contato com os diversos métodos definidos ao longo dessa hierarquia para ter uma noção mais precisa sobre o que é possível realizar com um objeto `Frame`. Por exemplo, para tratar eventos relativos a janelas é preciso associar ao *frame* um `WindowListener` através do método `addWindowListener()`, definido na classe `Window`.

A classe `Frame` propriamente dita define os métodos que dizem respeito ao que um *frame* tem de específico que uma janela (objeto da classe `Window`) não tem — por exemplo, `setTitle()` para definir um título para a barra da janela, `setMenuBar()` para incluir uma barra de menus e `setCursor()` para determinar o tipo de cursor a ser utilizado. O tipo de cursor é definido por uma das alternativas especificadas como constantes da classe `Cursor`, tais como `HAND_CURSOR` ou `Cursor.TEXT_CURSOR`.

O pacote `Swing` também oferece a sua versão de um *frame* através da classe `JFrame`, que é uma extensão da classe `Frame`. Esse código ilustra a criação de uma janela vazia, como acima, usando `JFrame`:

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  public class SJanela extends JFrame {
4      class WindowHandler extends WindowAdapter {
5          public void windowClosing(WindowEvent we) {
6              dispose();
7              System.exit(0);
8          }
9      }
10     public SJanela() {
11         this("Janela");
```

```
12     }
13     public SJanela(String title) {
14         setSize(200,120);
15         setTitle(title);
16         addWindowListener(new WindowHandler());
17     }
18     public static void main(String[] args) {
19         SJanela je = new SJanela();
20         je.show();
21     }
22 }
```

4.2.5 Gerenciadores de *layout*

Quando um *container* tem mais de um componente, é preciso especificar como esses componentes devem ser dispostos na apresentação. Em Java, um objeto que implemente a interface `LayoutManager` é responsável por esse gerenciamento de disposição. A interface `LayoutManager2` é uma extensão de `LayoutManager` que incorpora o conceito de restrições de posicionamento de componentes em um *container*.

Os principais métodos da classe `Container` relacionados ao *layout* de componentes são `setLayout()`, que recebe como argumento o objeto que implementa `LayoutManager` e que determina qual a política de gerência de disposição de componentes adotada; e `validate()`, usado para rearranjar os componentes em um *container* se o gerenciador de *layout* sofreu alguma modificação ou novos componentes foram adicionados.

O pacote `java.awt` apresenta vários gerenciadores de *layout* pré-definidos. As classes `FlowLayout` e `GridLayout` são implementações de `LayoutManager`; as classes `BorderLayout`, `CardLayout` e `GridBagLayout` são implementações de `LayoutManager2`. Swing acrescenta ainda um gerenciador de *layout* através da classe `BoxLayout`.

Para determinar que o gerenciador de *layout* em um *container* `c` seja do tipo `XXXLayout`, é preciso invocar `setLayout()` passando como argumento um objeto gerenciador:

```
c.setLayout(new XXXLayout());
```

Containers derivados da classe `Window` têm como padrão um gerenciador de *layout* do tipo `BorderLayout`, enquanto aqueles derivados de `Panel` usam como padrão `FlowLayout`.

FlowLayout

`FlowLayout` é uma classe gerenciadora de *layout* que arranja os componentes sequencialmente na janela, da esquerda para a direita, do topo para baixo, à medida que os componentes são adicionados ao *container*.

Os componentes são adicionados ao *container* da forma similar a um texto em um parágrafo, permitindo que cada componente mantenha seu tamanho natural. Como padrão, os componentes são horizontalmente centralizados no *container*. É possível mudar esse padrão de alinhamento especificando um valor alternativo como parâmetro para um dos construtores da classe ou para o método

`setAlignment()`. Esse parâmetro pode assumir um dos valores constantes definidos na classe, tais como `LEFT` ou `RIGHT`.

É possível também modificar a distância em *pixels* entre os componentes arranjados através desse tipo de gerenciador com os métodos `setHgap()` e `setVgap()`; alternativamente, esses valores podem também ser especificados através de construtores da classe `FlowLayout`. Para obter os valores utilizados, há métodos `getHgap()` e `getVgap()`.

Esse é o gerenciador de *layout* padrão para *containers* derivados de `Panel`, tais como `Applet`.

Esse exemplo ilustra o uso desse tipo de gerenciador de *layout* para dispor um conjunto de botões em um *frame*:

```
1  import java.awt.*;
2  public class JanelaFlow extends Frame {
3      public JanelaFlow() {
4          setTitle("FlowLayout");
5          setSize(240,100);
6          setLayout(new FlowLayout());
7      }
8      public void addButton(int count) {
9          for(int i=1; i <= count; ++i)
10             add(new Button("B"+i));
11     }
12     public static void main(String[] args) {
13         JanelaFlow j = new JanelaFlow();
14         int qtde = 10;
15         try {
16             if (args.length > 0)
17                 qtde = Integer.parseInt(args[0]);
18         }
19         catch (Exception e) {
20         }
21
22         j.addButton(qtde);
23         j.validate();
24         j.setVisible(true);
25     }
26 }
```

GridLayout

`GridLayout` é uma implementação de `LayoutManager` que permite distribuir componentes ao longo de linhas e colunas. A distribuição dá-se na ordem de adição do componente ao *container*, da esquerda para a direita e de cima para baixo.

Essa classe oferece um construtor que permite especificar o número de linhas e colunas desejado para o *grid* — para o construtor padrão, um *grid* de uma única coluna é usado. Adicionalmente, há

outro construtor que permite especificar, além da quantidade de linhas e colunas, o espaço em *pixels* entre esses componentes nas direções horizontal e vertical.

Deve-se observar que, nesse tipo de *layout*, as dimensões dos componentes são ajustadas para ocupar o espaço completo de uma posição no *grid*.

Esse exemplo mostra como `GridLayout` dispõe um conjunto de dez botões em um *grid* de três linhas e quatro colunas:

```
1  import java.awt.*;
2
3  public class JanelaGrid extends Frame {
4      private final int rows=3, cols=4;
5      public JanelaGrid() {
6          setTitle("GridLayout");
7          setSize(240,100);
8          setLayout(new GridLayout(rows, cols));
9      }
10
11     public void addButton(int count) {
12         int max = rows*cols;
13         if (count < max)
14             max = count;
15         for(int i=1; i <= max; ++i)
16             add(new Button("B"+i));
17     }
18
19     public static void main(String[] args) {
20         JanelaGrid j = new JanelaGrid();
21         int qtde = 10;
22         try {
23             if (args.length > 0)
24                 qtde = Integer.parseInt(args[0]);
25         }
26         catch (Exception e) {
27         }
28
29         j.addButton(qtde);
30         j.validate();
31         j.setVisible(true);
32     }
33 }
```

BorderLayout

`BorderLayout` é uma implementação de `LayoutManager2` adequado para janelas com até cinco componentes. Ele permite arranjar os componentes de um *container* em cinco regiões, cujo

posicionamento é representado pelas constantes `BorderLayout.NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`:

```
1  import java.awt.*;
2  public class JanelaBorder extends Frame {
3      public JanelaBorder() {
4          setTitle("BorderLayout");
5          setSize(240,100);
6          add(new Button("North"), BorderLayout.NORTH);
7          add(new Button("South"), BorderLayout.SOUTH);
8          add(new Button("East"), BorderLayout.EAST);
9          add(new Button("West"), BorderLayout.WEST);
10         add(new Button("Center"), BorderLayout.CENTER);
11     }
12     public static void main(String[] args) {
13         JanelaBorder j = new JanelaBorder();
14         j.validate();
15         j.setVisible(true);
16     }
```

Observe a utilização do método `add(Component c, Object o)` da classe `Container`, que incorpora a especificação das restrições de posicionamento.

Além do construtor padrão, outro construtor permite especificar o *gap* horizontal e vertical (em *pixels*) entre os componentes.

Esse tipo de *layout* é o padrão para *containers* do tipo `Frame`. Para utilizar esse tipo de gerenciador para janelas com mais de cinco componentes, basta definir que o componente inserido em um `BorderLayout` seja um `Panel`, que é um *container* que pode ter seu próprio gerenciador de *layout*.

CardLayout

Um gerenciador do tipo `CardLayout` empilha os componentes de um container de tal forma que apenas o componente que está no topo permanece visível. Esse gerenciador implementa a interface `LayoutManager2`.

Os métodos do gerenciador estabelecem funcionalidades que permitem “navegar” entre os componentes empilhados, determinando qual item deve estar visível em um dado momento. Os métodos de navegação `first()`, `next()`, `previous()` e `last()` permitem realizar uma varredura sequencial pelos componentes de container especificado, cujo gerenciador de *layout* deve ser do tipo `CardLayout`.

O método `show()` permite selecionar um componente para exposição diretamente através de uma *string*, que é especificada como uma restrição quando da adição do componente ao *container*, como no seguinte fragmento:

```
Panel p = new Panel();
cm = new CardLayout();
```

```
p.setLayout(cm);  
Button b = new Button("Teste");  
p.add(b, b.getLabel());
```

que permitiria a exibição desse botão com a invocação

```
cm.show(p, b.getLabel());
```

Tipicamente, os componentes manipulados por um gerenciador do tipo `CardLayout` são *containers*, os quais por sua vez utilizam qualquer outro tipo de gerenciador de *layout*.

Esse exemplo ilustra o uso de `CardLayout` para organizar cinco rótulos distintos em um *container* alocado ao elemento central de um frame com `BorderLayout`. Os quatro elementos periféricos do `BorderLayout` são usados para navegar entre os rótulos do `CardLayout` central:

```
1  import java.awt.*;  
2  import java.awt.event.*;  
3  public class JanelaCard extends Frame {  
4      Panel central = new Panel();  
5      CardLayout cl = new CardLayout();  
6      class FirstHandler implements ActionListener {  
7          public void actionPerformed(ActionEvent ae) {  
8              cl.first(central);  
9          }  
10     }  
11     class LastHandler implements ActionListener {  
12         public void actionPerformed(ActionEvent ae) {  
13             cl.last(central);  
14         }  
15     }  
16     class NextHandler implements ActionListener {  
17         public void actionPerformed(ActionEvent ae) {  
18             cl.next(central);  
19         }  
20     }  
21     class PreviousHandler implements ActionListener {  
22         public void actionPerformed(ActionEvent ae) {  
23             cl.previous(central);  
24         }  
25     }  
26     public JanelaCard() {  
27         setTitle("CardLayout");  
28         setSize(240,200);  
29         Button next = new Button("Proximo");  
30         next.addActionListener(new NextHandler());  
31         Button previous = new Button("Anterior");  
32         previous.addActionListener(new PreviousHandler());
```

```
33         Button first = new Button("Primeiro");
34         first.addActionListener(new FirstHandler());
35         Button last = new Button("Ultimo");
36         last.addActionListener(new LastHandler());
37         add(first, BorderLayout.NORTH);
38         add(last, BorderLayout.SOUTH);
39         add(previous, BorderLayout.WEST);
40         add(next, BorderLayout.EAST);
41         add(central, BorderLayout.CENTER);
42         central.setLayout(cl);
43         central.add(new Label("Primeiro painel"), "Primeiro");
44         central.add(new Label("Segundo painel"), "Segundo");
45         central.add(new Label("Terceiro painel"), "Terceiro");
46         central.add(new Label("Quarto painel"), "Quarto");
47         central.add(new Label("Quinto painel"), "Quinto");
48     }
49     public static void main(String[] args) {
50         JanelaCard jc = new JanelaCard();
51         jc.setVisible(true);
52     }
53 }
```

Nesse exemplo, o rótulo “Primeiro painel” é inicialmente exibido por estar no topo da pilha do objeto `Panel` com gerenciador do tipo `CardLayout`. Se o botão “Próximo” for pressionado, o rótulo “Segundo painel” será exibido.

GridBagLayout

`GridBagLayout` é o gerenciador de *layout* mais flexível dentre aqueles pré-definidos em `java.awt`; é também, em decorrência dessa flexibilidade, o mais complexo. É uma implementação de `LayoutManager2` que permite, como `GridLayout`, arranjar componentes ao longo de uma matriz de linhas e colunas. No entanto, componentes podem ser acrescentados em qualquer ordem e podem também variar em tamanho, podendo ocupar mais de uma linha ou coluna.

Uma vez que o desenho da interface tenha sido especificado, a chave para a utilização desse gerenciador é a criação de um objeto de restrição de posicionamento. Esse objeto é da classe `GridBagConstraints`. Objetos da classe `GridBagConstraints` determinam como um gerenciador do tipo `GridBagLayout` deve posicionar um dado componente em seu container.

Uma vez que esse objeto tenha sido criado e suas restrições especificadas, basta associar essas restrições ao componente usando o método `setConstraints()` e adicioná-lo ao *container* com o método `add()` com o segundo parâmetro de restrições. A especificação das restrições de posicionamento, tamanho e propriedades de um componente nesse tipo de gerenciador é determinada através da atribuição de valores a campos públicos do objeto da classe `GridBagConstraints`.

O posicionamento é especificado pelas variáveis `gridx` e `gridy`, respectivamente para indicar a coluna e a linha onde o componente deve ser posicionado. Para `gridx`, o valor 0 indica a coluna mais à esquerda. Do mesmo modo, para `gridy` o valor 0 indica a linha mais ao topo. Além de

valores absolutos de posicionamento, essa classe define a constante `RELATIVE` para posicionamento relativo, após o último componente incluído, sendo esse o valor padrão para esses campos.

O número de células que o componente ocupa no *grid* é indicado pelas variáveis `gridwidth` e `gridheight`, relacionadas respectivamente ao número de colunas e ao número de linhas que será ocupado pelo componente. O valor `REMAINDER` para esses campos indica que o componente será o último dessa linha ou coluna, devendo ocupar a largura ou altura restante. O valor padrão desses campos é 1.

Outras variáveis de restrição definidas nessa classe incluem `weightx`, `weighty`, `fill` e `anchor`. Os atributos `weightx` e `weighty` indicam o peso, ou a prioridade, que o componente terá para receber porções de espaço extra horizontalmente ou verticalmente, respectivamente, quando o *container* é redimensionado e espaço adicional torna-se disponível. O padrão é um componente não receber espaço extra (valor 0).

O atributo `fill` é utilizado quando a área para a apresentação do componente é maior que o tamanho natural do componente. Indica como a apresentação do componente irá ocupar a área disponível, podendo assumir os valores definidos em constantes da classe: `NONE`, não modifica o tamanho do componente (o padrão); `VERTICAL`, ocupa o espaço vertical mas não altera a largura do componente; `HORIZONTAL`, ocupa o espaço disponível na horizontal mas não altera a altura do componente; e `BOTH`, ocupa o espaço disponível nas duas dimensões.

O atributo `anchor` é utilizado quando o tamanho do componente é menor que a área da célula à qual ele foi alocado para indicar a posição do componente na célula. O padrão é `CENTER`, mas outros valores possíveis são `NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST`.

Esse exemplo de código ilustra a criação de uma janela com três regiões, sendo que a primeira região contém uma lista, a segunda um grupo de três botões e a terceira uma área de texto:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class JanelaGridBag extends Frame {
4      GridBagLayout gb = new GridBagLayout();
5      private final int noLinhas = 3;
6      public JanelaGridBag() {
7          setTitle("GridBagLayout");
8          setSize(320,200);
9          setLayout(gb);
10         List lEsq = new List(noLinhas, true);
11         lEsq.add("Um");
12         lEsq.add("Dois");
13         lEsq.add("Tres");
14         lEsq.add("Quatro");
15         lEsq.add("Cinco");
16         lEsq.add("Seis");
17         lEsq.add("Sete");
18         Button add = new Button(">>");
19         Button clear = new Button("Clear");
20         Button close = new Button("Close");
```



```
21         TextArea tDir = new TextArea("", noLinhas,
22                                     10, TextArea.SCROLLBARS_NONE);
23         GridBagConstraints gc = new GridBagConstraints();
24         gc.gridx = 0;
25         gc.gridy = 0;
26         gc.gridwidth = 1;
27         gc.gridheight = 3;
28         gc.fill = GridBagConstraints.VERTICAL;
29         add(lEsq, gc);
30         gc.gridx = 2;
31         add(tDir, gc);
32         Insets margens = new Insets(4, 3, 4, 3);
33         gc.gridx = 1;
34         gc.gridy = 0;
35         gc.gridwidth = 1;
36         gc.gridheight = 1;
37         gc.fill = GridBagConstraints.BOTH;
38         gc.ipadx = 4;
39         gc.ipady = 4;
40         gc.insets = margens;
41         add(add, gc);
42         gc.gridy = 1;
43         add(clear, gc);
44         gc.gridy = 2;
45         add(close, gc);
46     }
47     public static void main(String[] args) {
48         JanelaGridBag jgb = new JanelaGridBag();
49         jgb.setVisible(true);
50     }
51 }
```

BoxLayout

Swing oferece um gerenciador de *layout* simples, com alto grau de flexibilidade, que é o *BoxLayout*. Nesse tipo de *layout*, componentes podem ser dispostos em uma única linha ou em uma única coluna, porém arranjos de componentes bem complexos podem ser obtidos através da combinação desses mecanismos.

Em *BoxLayout* os componentes mantêm sua dimensão natural, como em *FlowLayout*. A direção na qual os componentes serão dispostos — se da esquerda para a direita ou se de cima para baixo — pode ser especificada no construtor da classe, através respectivamente das constantes *X_AXIS* ou *Y_AXIS*.

Tipicamente, esse tipo de *layout* não é utilizado diretamente, mas sim através de um *container* do tipo *Box*, que adota *BoxLayout* como padrão único de gerenciamento de *layout*.

Esse código ilustra a construção de uma interface similar àquela usando `GridBagLayout`, porém construída usando caixas aninhadas:

```
1  import javax.swing.*;
2  public class JanelaBox extends JFrame {
3      public JanelaBox() {
4          setTitle("BoxLayout");
5          setSize(240,120);
6          Box h = Box.createHorizontalBox();
7          Box v = Box.createVerticalBox();
8          String[] lista = {"Um", "Dois", "Tres", "Quatro",
9                          "Cinco", "Seis", "Sete"};
10         JList jl = new JList(lista);
11         jl.setFixedCellWidth(70);
12         int mis = ListSelectionModel.MULTIPLE_INTERVAL_SELECTION;
13         jl.setSelectionMode(mis);
14         JScrollPane lEsq = new JScrollPane(jl);
15         lEsq.setMinimumSize(new java.awt.Dimension(100,100));
16         JButton add = new JButton(">>");
17         JButton clear = new JButton("Clear");
18         JButton close = new JButton("Close");
19         JTextArea tDir = new JTextArea();
20         v.add(add);
21         v.add(clear);
22         v.add(close);
23         h.add(lEsq);
24         h.add(v);
25         h.add(tDir);
26         getContentPane().add(h);
27     }
28     public static void main(String[] args) {
29         JanelaBox jb = new JanelaBox();
30         jb.setVisible(true);
31         jb.validate();
32     }
33 }
```

4.3 Desenvolvimento de applets

Applets são programas projetados para ter uma execução independente dentro de alguma outra aplicação, eventualmente interagindo com esta — tipicamente, um *browser* (navegador) Web. Assim, *applets* executam no contexto de um outro programa, o qual interage com o applet e determina assim sua seqüência de execução. Funcionalidades associadas a *applets* Java são agregadas no pacote `java.applet`.

O processo de criação de um *applet* é similar ao processo de criação de uma aplicação — o programa deve ser criado por um editor de programas e o arquivo de *bytecodes* (com extensão `.class`) deve ser gerado a partir da compilação do arquivo fonte.

Uma vez criado ou disponibilizado o *bytecode*, é preciso incluir uma referência a ele em alguma página Web. Essa página Web, uma vez carregada em algum navegador, irá reconhecer o elemento que faz a referência ao *applet*, transferir seu *bytecode* para a máquina local e dar início à sua execução.

4.3.1 Criação de *applet*

Para criar um *applet*, é preciso desenvolver uma classe que estenda a classe `Applet`:

```
import java.applet.*;
public class MeuApplet extends Applet {
    ...
}
```

A classe `Applet` é uma extensão de uma classe *container* do tipo `Panel`. Assim, o desenvolvimento de um *applet* segue as estratégias de desenvolvimento de qualquer aplicação gráfica.

Um *applet* deve ser referenciado a partir de uma página Web. Tipicamente, para referenciar um *applet* cujo *bytecode* esteja em um arquivo `MeuApplet.class` a partir de uma página usando HTML na versão 3.2, o seguinte elemento deve ser incluído na página:

```
<applet code="MeuApplet.class"
        width=200 height=150>
</applet>
```

Atributos opcionais para o elemento `APPLET` incluem, entre outros, `CODEBASE`, que indica o diretório (na forma de um URL) onde está localizado o código do *applet*, e `NAME`, uma *string* para identificar o *applet*.

Para navegadores compatíveis com HTML na versão 4.01 ou superiores ou ainda que usem XHTML, a forma de referenciar um *applet* na página seria através da *tag* `OBJECT`:

```
<object codetype="application/java"
        classid="java:MeuApplet.class"
        width="200" height="150">
</object>
```

O navegador irá então criar um espaço gráfico de 200 *pixels* de largura por 150 *pixels* de altura na página apresentada, irá transferir o código do *applet* do mesmo local de onde a página se originou para a máquina local e dará início à execução do *applet* dentro deste espaço usando a máquina virtual Java associada ao navegador.

É possível também passar argumentos para o código do *applet* a partir da página HTML que o referencia. Para tanto, o tag `PARAM` é utilizado no interior do elemento `APPLET`, ocorrendo tantas vezes quantos forem os argumentos para o *applet*. Cada ocorrência de `PARAM` tem dois atributos, `name` e `value`, que permitirão a identificação do argumento no código do *applet*:

```
<applet ...>
<param name="background" value="white">
</applet>
```

4.3.2 Execução de *applets*

Ao contrário das aplicações Java, a execução de um *applet* em uma página não é iniciada pelo método `main()`. Um *applet* é executado como uma *thread* subordinada ao navegador (ou à aplicação `appletviewer`, presente no ambiente de desenvolvimento Java). O navegador será responsável por invocar os métodos da classe `Applet` que controlam a execução de um *applet*.

Quando o *applet* é carregado pela primeira vez pelo navegador, o método `init()` é invocado pelo navegador. Esse método pode ser considerado funcionalmente equivalente ao método construtor em aplicações, pois será apenas executado no momento em que o código for carregado.

Após o carregamento do *applet* para a máquina local e a execução do método `init()`, o método `start()` é invocado. Esse método será invocado também a cada vez que a área do *applet* torna-se visível no navegador, dando reinício a operações que eventualmente tenham sido paralisadas pelo método `stop()`.

O método `stop()` é invocado cada vez que o *applet* torna-se temporariamente invisível, ou seja, que sua área não esteja visível no navegador. É uma forma de evitar que operações que demandem muitos ciclos de CPU continuem a executar desnecessariamente. Também é invocado imediatamente antes de `destroy()`, que é invocado quando o *applet* está para ser eliminado do navegador. Este método serve para liberar recursos — além de memória — que o *applet* tenha eventualmente alocado para sua execução.

Esse exemplo ilustra a definição de um *applet* que implementa apenas esses quatro métodos. Quando carregada em um navegador Web, a página que referencia esse código apresenta apenas uma área vazia; porém, se o console Java do navegador for aberto, será possível visualizar as mensagens:

```
1  import java.applet.*;
2  public class MeuApplet extends Applet {
3      public void init() {
4          System.out.println("MeuApplet inicializado com dimensao "
5                               + getSize());
6      }
7      public void start() {
8          System.out.println("MeuApplet deve executar");
9      }
10     public void stop() {
11         System.out.println("MeuApplet deve parar");
12     }
13     public void destroy() {
14         System.out.println("Adeus, MeuApplet");
15     }
```

Sendo `Applet` um componente gráfico do tipo *container*, todas as funcionalidades descritas para o desenvolvimento de aplicações gráficas aplicam-se a *applets*. Por exemplo, é possível que um *applet* apresente formas geométricas em seu contexto gráfico usando o método `paint()` ou incluir componentes de interface com usuário e tratar a manipulação de eventos.

Alguns métodos da classe `Applet` permitem a comunicação do *applet* com o navegador no qual ele está inserido. O método `getParameter()` permite a obtenção dos parâmetros passados pelo navegador para o *applet*. O método `showStatus()` exibe uma mensagem na linha de *status* do

navegador (na barra inferior da janela) que executa o *applet*. O método `getAppletContext()` permite obter o contexto de execução do *applet*, o que permite por exemplo estabelecer a comunicação entre dois *applets* de uma mesma página.

Como o código de um *applet* é geralmente obtido através de uma conexão remota, sua execução está restrita aos princípios de segurança impostos por Java para a execução de código em ambientes distribuídos. Tipicamente, um *applet* não pode acessar arquivos e informações do ambiente onde está executando nem pode estabelecer conexões remotas com outras máquinas, a não ser aquela de onde o próprio código foi obtido.

4.3.3 Passagem de parâmetros

O método `getParameter()` da classe `Applet` permite obter parâmetros passados para o *applet* através do elemento `PARAM` em uma página HTML. O argumento para esse método é o nome do parâmetro, estabelecido no atributo `name` da *tag* `PARAM`, e o retorno é uma *string* com o valor do argumento — ou seja, o conteúdo do atributo `value` de `PARAM`.

Esse *applet* recebe como parâmetro a cor de fundo, que pode ser branca, amarela ou cinza (o padrão):

```
1  import java.applet.*;
2  import java.awt.*;
3  public class AppletParameter extends Applet {
4      private int altura;
5      private int largura;
6      private int incremento;
7      public void init() {
8          Dimension d = getSize();
9          altura = d.height - 2;
10         largura = d.width;
11         incremento = altura/3;
12         Color c = Color.gray;
13         String cor = getParameter("background");
14         try {
15             if (cor.equalsIgnoreCase("yellow"))
16                 c = Color.yellow;
17             else if (cor.equalsIgnoreCase("white"))
18                 c = Color.white;
19         }
20         catch (Exception e) {
21         }
22
23         setBackground(c);
24     }
25     public void paint(Graphics g) {
26         int x=1;
27         int y=1;
```

```
28         int finalPos = largura - altura;
29         while (x < finalPos) {
30             g.setColor(new Color((float)Math.random(),
31                                   (float)Math.random(),
32                                   (float)Math.random()));
33             g.drawRect(x, y, altura, altura);
34             x += incremento;
35         }
36     }
37 }
```

Se na página HTML o elemento especificado for

```
<p>
<APPLET code="AppletParameter.class" width="640" height="20">
  <PARAM name="background" value="White">
</APPLET>
```

o *applet* será apresentado com fundo branco; com

```
<p>
<APPLET code="AppletParameter.class" width="640" height="20">
  <PARAM name="background" value="yellow">
</APPLET>
```

será apresentado com fundo amarelo. Se for invocado sem argumento, tendo na página HTML o elemento

```
<p>
<APPLET code="AppletParameter.class" width="640" height="20">
</APPLET>
```

os retângulos serão desenhados sobre um fundo cinza.

4.3.4 Contexto de execução

Em algumas situações pode ser de interesse fazer com que o *applet* interaja com o contexto no qual ele está executando (usualmente, o navegador Web). A interface `AppletContext` especifica algumas funcionalidades que permitem essa interação.

Para obter o contexto no qual o *applet* está executando, o método `getAppletContext()` da classe `Applet` é utilizado. Esse método retorna um objeto `AppletContext`, a partir do qual é possível obter referências a outros *applets* no mesmo contexto através de uma enumeração, usando o método `getApplets()`. Alternativamente, se ao *applet* foi atribuído um nome usando o atributo `name` na tag `APPLET`, uma referência a esse *applet* pode ser obtida através do método `getApplet()`.

Como exemplo, considere a execução de dois *applets* em uma página, onde um *applet* tem um campo de texto para obter uma entrada do usuário e o outro tem uma área de texto para exibir as

entradas que o usuário digitou na outra janela. O primeiro *applet* é definido em uma classe `Entrada`, enquanto que o segundo é definido em uma classe `TextAreaApplet`. A inclusão em uma página HTML dá-se através de dois elementos `APPLET`:

```
<p>
  <applet code="Entrada.class"
          width="200" height="100"
          name="entra">
  </applet>
  <applet code="TextAreaApplet.class"
          width="200" height="100"
          name="mostra">
  </applet>
</p>
```

O *applet* `TextAreaApplet` tem simplesmente uma área para exibição de texto:

```
1  import java.awt.*;
2  import java.applet.*;
3  public class TextAreaApplet extends Applet {
4      TextArea ta = new TextArea(5, 30);
5      public void init() {
6          add(ta);
7      }
8      public void append(String s) {
9          ta.append("\n" + s);
10     }
11 }
```

O *applet* `Entrada` define um campo para entrada de texto e estabelece a conexão entre os contextos em seu método de inicialização:

```
1  import java.applet.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  public class Entrada extends Applet {
5      TextField tf = new TextField(30);
6      TextAreaApplet ta;
7      public void init() {
8          add(tf);
9          ta = (TextAreaApplet) getAppletContext().getApplet("mostra");
10         tf.addActionListener(new ActionListener() {
11             public void actionPerformed(ActionEvent ae) {
12                 ta.append(tf.getText());
13             }
14         });
15     }
```

```
15     }  
16 }
```

É possível também determinar que o navegador deve carregar um novo documento a partir de um *applet*, usando o método `showDocument()` de `AppletContext`. O argumento para esse método é um objeto localizador uniforme de recursos, do tipo `java.net.URL`.

Capítulo 5

Desenvolvimento de aplicações distribuídas

Entre os atrativos de Java está a facilidade que essa linguagem oferece para desenvolver aplicações para execução em sistemas distribuídos. Já em sua primeira versão, Java oferecia facilidades para o desenvolvimento de aplicações cliente-servidor usando os mecanismos da Internet, tais como os protocolos TCP/IP e UDP.

Se o cliente na aplicação distribuída precisa acessar um servidor de banco de dados relacional, Java oferece uma API específica para tal fim, JDBC. Através das classes e interfaces desse pacote é possível realizar consultas expressas em SQL a um servidor de banco de dados e manipular as tabelas obtidas como resultado dessas consultas.

Em termos de desenvolvimento voltado para a World-Wide Web, Java oferece o já clássico mecanismo de *applets*, código Java que executa em uma máquina virtual no lado do cliente Web, como descrito na Seção 4.3. O mecanismo de *servlets* permite associar o potencial de processamento da plataforma Java a servidores Web, permitindo construir assim uma camada *middleware* baseada no protocolo HTTP e em serviços implementados em Java.

Aplicações distribuídas mais elaboradas podem ser desenvolvidas usando uma arquitetura de objetos distribuídos, onde aplicações orientadas a objetos lidam diretamente com referências a objetos em processos remotos. Java oferece duas alternativas nessa direção, RMI (*Remote Method Invocation*), uma solução 100% Java, e Java IDL, uma solução integrada à arquitetura padrão CORBA. Um passo adiante na evolução desse tipo de sistema é a utilização do conceito de agentes móveis, onde não apenas referências a objetos são manipuladas remotamente mas os próprios objetos — código e estado — movem-se pela rede.

5.1 Programação cliente-servidor

O paradigma de programação distribuída através da separação das aplicações entre servidores (aplicações que disponibilizam algum serviço) e clientes (aplicações que usam esses serviços) foi a arquitetura de distribuição predominante nos anos 1990. Um dos seus atrativos é o aumento da confiabilidade (a falha de uma máquina não necessariamente inviabiliza a operação do sistema como um todo) e redução de custo (máquinas mais simples podem executar os serviços isoladamente, ao invés de ter uma grande máquina fazendo todos os serviços).

As aplicações clientes e servidoras são programas executando em máquinas distintas, trocando informação através de uma rede de computadores. Para que os serviços possam ser solicitados, a aplicação cliente deve conhecer quem fornece o serviço (o endereço da aplicação servidora) e qual o protocolo pré-estabelecido para realizar a solicitação.

Entre as vantagens citadas para o modelo de programação cliente-servidor destacam-se:

- Relaciona a execução de processos distintos.
- Oferece uma estruturação do processamento distribuído baseado no conceito de serviços, tendo um servidor, o provedor de serviços, e um cliente, o consumidor de serviços oferecidos.
- Permite compartilhamento de recursos, com o servidor atendendo a vários clientes.
- Oferece transparência de localização, com tratamento uniforme independentemente de processos estarem na mesma máquina ou em máquinas distintas.
- Permite a comunicação através da troca de mensagens, oferecendo uma arquitetura fracamente acoplada através das mensagens para solicitações (cliente para servidor) e respostas (servidor para cliente).
- Encapsula serviços, pois o cliente não precisa saber como servidor implementa o serviço, mas apenas a interface para solicitação e resposta.

Estaremos estudando aqui como Java oferece e simplifica o suporte a esse tipo de programação através das funcionalidades do pacote `java.net`. A partir da apresentação de alguns conceitos preliminares, apresenta-se os fundamentos Java para criar aplicações distribuídas usando os mecanismos de TCP/IP, UDP e HTTP.

5.1.1 Conceitos preliminares

A programação em redes de computadores é atualmente a regra, não a exceção. A principal vantagem nesse modelo de programação é a possibilidade de distribuir tarefas computacionalmente pesadas e conjuntos extensos de dados e informações entre diversas máquinas.

No entanto, há um custo associado a essa distribuição. Há necessidade de trocar mensagens entre as máquinas envolvidas no processamento, com um custo (tempo) adicional necessário para efetivar essa troca.

É importante também que os dispositivos envolvidos na troca de mensagens comuniquem-se usando uma mesma linguagem, ou protocolo. Protocolos são organizados em camadas (ou pilhas) de diferentes níveis de abstração. O conjunto de protocolos mais comuns na programação em rede é aquele estabelecido pela arquitetura TCP/IP, que opera com um *software* de suporte oferecido pelo sistema operacional de uma máquina ligada em rede.

A arquitetura TCP/IP organiza uma rede de computadores em quatro camadas:

Interface de rede: define padrões de conexão à rede física, seja local (Ethernet-CSMA/CD, Token Ring, FDDI, ATM) ou de longa distância (HDLC, X.25, ATM). Opera com endereços físicos, realizando a conversão entre endereços físicos e lógicos através dos protocolos ARP (*Address Resolution Protocol*) e RARP (*Reverse ARP*).

Inter-redes: protocolos para transporte não-confiável de mensagens (IP — *Internet Protocol*), para controle da comunicação e informe de erros (ICMP — *Internet Control Message Protocol*) e para roteamento de mensagens (EGP — *Exterior Gateway Protocol*, RIP — *Routing Information Protocol*). Endereços para comunicação *host* a *host* são lógicos (endereços IP).

Transporte: protocolos para transporte confiável de dados por conexão (TCP/IP — *Transfer Control Protocol/IP*) e para transporte de datagramas, sem conexão (UDP — *User Datagram Protocol*). Introduz o conceito de porta (endereço que identifica a aplicação na máquina). Endereços nesse nível são descritos por um par (*host*, *port*).

Aplicação: define conjunto de serviços manipulados por usuários. Serviços utilizam filosofia cliente-servidor, com os servidores estabelecendo portas para disponibilização do serviço. Algumas portas de serviços TCP/IP já são pré-definidas, sendo denominadas de portas notáveis. Exemplos de portas notáveis incluem: 7, echo (reenvia o que recebe); 21, ftp (transferência de arquivos); 23, telnet (terminal virtual); 25, smtp (correio eletrônico); e 37, time (envia hora e data local da máquina).

Na versão corrente do protocolo, endereços IP ocupam 32 bits e são divididos em cinco classes. As classes A, B e C têm seus endereços estruturados em um prefixo de identificação de classe (binários 0, 10 e 110, respectivamente), identificador de subrede (7, 14 e 21 bits) e identificador de *host*. A classe D (prefixo 1110) é utilizada para multicast, enquanto que endereços da classe E (11110) são reservados para uso futuro. Usualmente, endereços IP são representados por uma quádrupla de valores decimais correspondente aos quatro grupos de 8 bits do endereço. Nessa forma de representação, endereços iniciados por valores entre 0 e 127 são da classe A; entre 128 e 191, classe B; entre 192 e 223, classe C; entre 224 e 239, classe D; e entre 240 e 247, classe E.

Existe também uma forma de representação simbólica de endereços IP baseada em nomes de domínios. Domínios são partições da rede Internet organizados hierarquicamente em estruturas de domínios e sub-domínios. Existe um mapeamento entre endereços IP representados simbolicamente e numericamente, o qual é realizado por servidores de nome distribuídos pela Internet através do Sistemas de Nomes de Domínio (DNS).

O desenvolvimento de *software* na arquitetura TCP/IP segue a filosofia de particionamento em múltiplos processos concorrentes. Isso permite a simplificação de projeto, implementação e manipulação de *software* para ambientes distribuídos, assim como a gerência independente de protocolos em diversos níveis.

O *software* é organizado na forma de processos independentes. No nível mais próximo da máquina, isso permite o isolamento dos dispositivos físicos através da utilização de *device drivers*.

No nível da camada de transporte, dois tipos de processos são suportados. A informação em processos TCP (entrada ou saída) é manipulada através do conceito de *streams* (arquivos seqüenciais). Já a informação em processos UDP é manipulada através do conteúdo de pacotes datagramas, enviados sem verificação de conteúdo ou garantia de recebimento. O processo IP atua como chaveador de datagramas.

As portas estabelecem a ligação entre o processo da aplicação e os processos IP, estando associadas a *buffers* finitos com acesso controlado. O acesso a portas pode sofrer bloqueio devido a uma tentativa de ler de uma porta com *buffer* vazio ou de escrever para porta com *buffer* cheio.

5.1.2 Aplicações TCP/IP

Para estabelecer a conexão TCP, é preciso identificar as extremidades dessa conexão tanto no processo cliente como no processo servidor. Essas extremidades são soquetes, identificados por um endereço de rede e um número de porta. A conexão pode ser interpretada como uma ligação direta entre os dois processos, através da qual os bytes podem fluir nos dois sentidos.

Um soquete TCP estabelece uma conexão stream bidirecional entre os endereços (*hostC, portC*) e (*hostS, portS*), ou seja, entre uma aplicação cliente em execução na máquina *hostC* controlando a porta *portC* e outra aplicação servidora em execução na máquina *hostS* monitorando a porta *portS* de *hostS*. A aplicação cliente utiliza a porta *portC* da máquina *hostC* para enviar solicitações de serviços e para receber retornos a suas solicitações. A aplicação servidora monitora constantemente a porta *portS* da máquina *hostS* aguardando a chegada de solicitações de serviço. Quando alguma solicitação é recebida, a aplicação servidora executa o serviço e utiliza a conexão para enviar o retorno com os resultados do serviço.

Java suporta a troca de bytes entre um cliente e um servidor TCP através do estabelecimento de uma conexão entre eles. Todas as funcionalidades de Java referentes ao estabelecimento de uma conexão TCP estão agregadas no pacote `java.net`.

Clientes TCP em Java

Em Java, a classe que permite o estabelecimento de uma conexão pelo lado do cliente é `Socket`. Para criar um soquete, o construtor da classe tipicamente utilizado é:

```
public Socket(InetAddress address, int port) throws IOException
```

Uma vez que a conexão entre cliente e servidor tenha sido estabelecida pela criação dos correspondentes soquetes, os dados da aplicação podem fluir através dos *streams* a ela associados.

Os argumentos do construtor estabelecem o endereço IP da máquina remota na conexão. A classe `InetAddress` de `java.net` permite representar endereços IP como objetos Java. Uma vez que um objeto Java esteja representando um endereço IP, ele pode ser utilizado como parâmetro para métodos de outras classes que manipulam transferências de dados através dos protocolos TCP/IP.

Os métodos estáticos dessa classe permitem definir tais objetos associados à representação simbólica ou numérica de endereços IP — `InetAddress.getByName(String host)` — ou associados à máquina local — `InetAddress.getLocalHost()`.

Este exemplo ilustra a manipulação de endereços IP através dos métodos dessa classe:

```
1  import java.net.*;
2  public class WhoIs {
3      public static void main(String[] args) {
4          try {
5              InetAddress myself = InetAddress.getLocalHost();
6              System.out.println("Local host is " +
7                               myself.getHostName() +
8                               " at IP address " +
9                               myself.getHostAddress());
10         }
11     }
```

```
11     catch (UnknownHostException uhe) {
12         System.err.println(uhe);
13     }
14     // Processamento dos argumentos na linha de comando
15     int count = 0;
16     InetAddress otherHost;
17     while (count < args.length) {
18         try{
19             otherHost = InetAddress.getByName(args[count]);
20             System.out.println("Host " + otherHost.getHostName() +
21                               " is at IP address " +
22                               otherHost.getHostAddress());
23         }
24         catch (UnknownHostException uhe) {
25             System.err.println(uhe);
26         }
27         ++count;
28     }
29 }
30 }
```

A classe `Socket` oferece também métodos para obter informações sobre os endereços (máquina e porta) envolvidos na conexão e para estabelecer *timeouts* associados à conexão.

Quando um soquete é criado, automaticamente são estabelecidos *streams* de entrada e de saída para a transferência de dados pela conexão. Os métodos `getInputStream()` e `getOutputStream()` da classe `Socket` permitem identificar os objetos associados a esses *streams*.

Streams implementam o conceito de cadeias unidirecionais de dados (FIFO, *First-In, First-Out*) apenas de escrita ou apenas de leitura. Assim, uma aplicação pode obter dados do início de um *stream* de entrada e pode enviar dados para o final de um *stream* de saída de dados, sempre sequencialmente.

Streams em Java são suportados por classes do pacote `java.io` (Seção 3.2). Para leitura sequencial de bytes utiliza-se um objeto da classe `InputStream`; para enviar bytes para um *stream* utiliza-se um objeto `OutputStream` e seus métodos `write()`, para agregar bytes ao *stream*, e `flush()`, para assegurar que os bytes inseridos sejam efetivamente encaminhados a seu destino.

Servidores TCP em Java

O processo servidor na arquitetura TCP deve estar preparado para responder a solicitações de conexões por parte dos clientes, permanecendo em estado de espera entre solicitações. Assim, um servidor TCP/IP deve realizar duas tarefas básicas: permanecer em execução aguardando (*listening*) a chegada de requisições em alguma porta pré-especificada; e responder à solicitação através de uma conexão estabelecida com o cliente em função da requisição recebida.

Em Java, a classe que permite a criação de servidores com essa funcionalidade é `ServerSocket`, também do pacote `java.net`. O principal método desta classe é `accept()`, que implementa a espera bloqueada por uma solicitação no endereço de porta especificado na construção do objeto.

O retorno desse método é um objeto da classe `Socket` que estabelece a conexão com a aplicação cliente.

Esse exemplo mostra o código para um servidor que responde a qualquer solicitação com uma mensagem fixa, cujo conteúdo é a sequência de bytes que compõe um endereço URL:

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4  public class TCPServer1 {
5      public static void main(String[] args) {
6          ServerSocket ss = null;
7          Socket cliente = null;
8          OutputStream os = null;
9          try {
10             ss = new ServerSocket(0);
11             System.out.println("Server: Aguardando na porta " +
12                             ss.getLocalPort());
13             while (true) {
14                 cliente = ss.accept();
15                 os = cliente.getOutputStream();
16                 System.out.println("Server: " +
17                                 "Processando solicitacao de " +
18                                 cliente.getInetAddress().getHostName());
19                 String data =
20                     "http://www.dca.fee.unicamp.br/cursos/PooJava/";
21                 byte[] buffer = data.getBytes();
22                 System.out.println("Server: Enviando \"" +
23                                 new String(buffer) + "\"");
24                 os.write(buffer);
25                 os.flush();
26             }
27         }
28         catch (Exception e) {
29             System.err.println(e);
30         }
31         finally {
32             try {
33                 os.close();
34                 cliente.close();
35                 ss.close();
36             }
37             catch (Exception e) {
38                 e.printStackTrace();
39             }
40         }
41     }
42 }
```

```
41     }  
42 }
```

Uso de múltiplas *threads*

Tipicamente, por razões de eficiência, um servidor TCP é implementado como um processo *multithreaded*. Um dos possíveis problemas na execução de aplicações segundo o modelo cliente-servidor está associado com o tempo de atendimento a uma solicitação pelo servidor. Se o servidor for um processo monolítico, ele estará indisponível para receber novas requisições enquanto a solicitação não for completamente atendida. A solução para este problema depende da possibilidade de se estabelecer um processamento independente para o atendimento de cada solicitação ao servidor, liberando tão cedo quanto possível o servidor para receber novas solicitações.

O conceito de processamento independente é parte integrante da linguagem Java, através de *multithreading*. Todo processamento em Java está associado a alguma *thread*, sendo que novas *threads* de execução podem ser criadas a partir de qualquer *thread*. A criação de novas *threads* é em geral associada a classes que implementam a interface `Runnable` do pacote `java.lang`. Essa interface especifica o método `run()`, os quais são utilizados para criar objetos da classe `Thread`, com métodos `start()` e `stop()`, entre outros.

Com as facilidades suportadas pela linguagem, torna-se atrativo implementar servidores *multithreaded*, cujo corpo principal de processamento resume-se a um laço eterno para aceitar solicitações na porta especificada e criar um objeto *thread* para atender à solicitação recebida. A funcionalidade do serviço que será executado pela *thread* é definida no corpo do método `run()` que implementa a interface `Runnable` associada à *thread* criada. Cada *thread* criada existe exclusivamente durante o tempo necessário para atender à solicitação.

O seguinte exemplo revisita o servidor TCP anteriormente apresentado usando o mecanismo de múltiplas *threads* de execução:

```
1  import java.io.*;  
2  import java.net.*;  
3  import java.util.*;  
4  class DataProvider implements Runnable {  
5      Socket client;  
6      OutputStream os = null;  
7      public DataProvider(Socket s) throws IOException {  
8          client = s;  
9          os = client.getOutputStream();  
10     }  
11     public void run() {  
12         String data =  
13             "http://www.dca.fee.unicamp.br/courses/PooJava/";  
14         byte[] buffer = data.getBytes();  
15         try {  
16             os.write(buffer);  
17             os.flush();  
18             os.close();
```

```
19         client.close();
20     }
21     catch (Exception e) {
22         System.err.println(e);
23     }
24 }
25 }
26 public class TCPServer2 {
27     public static void main(String[] args) {
28         ServerSocket ss = null;
29         Socket cliente = null;
30         try {
31             ss = new ServerSocket(0);
32             System.out.println("Server: Aguardando na porta " +
33                             ss.getLocalPort());
34             while (true) {
35                 cliente = ss.accept();
36                 System.out.println("Server: " +
37                                 "Processando solicitacao de " +
38                                 cliente.getInetAddress().getHostName());
39                 DataProvider dp = new DataProvider(cliente);
40                 new Thread(dp).start();
41             }
42         }
43         catch (Exception e) {
44             System.err.println(e);
45         }
46         finally {
47             try {
48                 ss.close();
49             }
50             catch (Exception e) {
51                 System.err.println(e);
52             }
53         }
54     }
55 }
```

5.1.3 Aplicações UDP

Aqui serão apresentadas as funcionalidades que Java oferece para a programação cliente-servidor usando o protocolo de transporte UDP.

A principal diferença em relação à programação cliente-servidor em TCP é que o protocolo UDP não suporta o conceito da transferência por streams de dados. UDP trabalha diretamente com o conceito de pacotes (datagramas). Assim, UDP não oferece a garantia de envio ou recepção e nem de

ordenação correta dos pacotes. Por outro lado, a ausência desses mecanismos permite uma transferência mais rápida.

O endereçamento em UDP dá-se como para a programação TCP, usando a classe Java `InetAddress`.

Soquetes UDP

Assim como para o protocolo TCP, UDP estabelece uma conexão entre o processo da aplicação e a rede através de um soquete.

Em Java, soquetes UDP são manipulados através de objetos da classe `DatagramSocket`. O construtor padrão para essa classe cria um soquete local na primeira porta disponível. Alternativamente, outro construtor permite especificar o número da porta desejado.

Ao contrário da classe `Socket`, um `DatagramSocket` não estabelece uma conexão com uma máquina remota, mas simplesmente um acesso local para a rede que pode ser utilizada para enviar e receber pacotes, através dos métodos `send()` e `receive()`, respectivamente.

Há um método `connect()` associado a objetos da classe `DatagramSocket`; no entanto, esse método atua como um filtro, só permitindo enviar ou receber pacotes para ou de um endereço IP ao qual o soquete foi conectado. O método `disconnect()` permite desconectar um soquete de um destino pré-especificado.

Datagramas

Os pacotes enviados e recebidos através dos soquetes UDP são objetos Java da classe `DatagramPacket`. Objetos dessa classe podem ser construídos de duas maneiras, dependendo se serão enviados ou recebidos através do soquete UDP:

Pacotes a enviar. Nesse caso, deve ser utilizado o construtor que incorpora em seus argumentos o arranjo de bytes a enviar, seu tamanho, e o endereço de destino (máquina, especificada pelo seu endereço IP, e porta).

Pacotes a receber. Nesse caso, os argumentos especificam apenas o arranjo de bytes para onde o conteúdo do pacote será transferido e o limite no tamanho do pacote que será recebido nesse arranjo.

Uma vez que um pacote tenha sido recebido, a informação sobre sua origem pode ser obtida através dos métodos `getAddress()` e `getPort()`. Os dados efetivamente recebidos podem ser extraídos do pacote usando o método `getData()`; o método `getLength()` permite determinar a dimensão dos dados.

Multicast

Um `MulticastSocket` é uma especialização de um `DatagramSocket` que permite que uma aplicação receba pacotes datagramas associados a um endereço *multicast* (classe D, endereços entre 224.0.0.1 e 239.255.255.255). Não é preciso nenhuma funcionalidade especial para apenas enviar datagramas para um endereço *multicast*.

Todos os soquetes *multicast* que estejam inscritos em um endereço *multicast* recebem o datagrama que foi enviado para esse endereço e porta. Para gerenciar a inscrição de um soquete em um endereço *multicast*, dois métodos são oferecidos na classe `MulticastSocket`.

O primeiro, `joinGroup(InetAddress m)`, permite à aplicação juntar-se a um grupo *multicast*. É o método que inscreve o soquete no grupo associado ao endereço *multicast* especificado como argumento.

O outro método, para desligar-se de um grupo *multicast*, é `leaveGroup(InetAddress m)`, que desconecta o soquete do grupo *multicast* especificado.

5.1.4 Aplicações HTTP

A *World Wide Web* (WWW ou simplesmente Web) é a primeira concretização de uma rede mundial de informação através de computadores. Proposta em 1989 no CERN (Suíça) por Tim-Berners Lee, ela interconecta principalmente documentos hipertexto (expressos em HTML — *HyperText Markup Language*) usando a infra-estrutura da Internet para a transferência de informação. Sua difusão expandiu-se principalmente após a popularização de interfaces gráficas com o usuário para navegação em hipertexto, iniciada a partir do lançamento do aplicativo Mosaic no NCSA (EUA).

A Web pode ser vista como um serviço de aplicação da arquitetura TCP/IP. Como tal, a arquitetura da Web define um esquema de endereçamento (URL) no nível da aplicação; estabelece protocolos de sessão (soquetes TCP/IP) e apresentação (HTML e auxiliares); define um protocolo (HTTP) no nível da aplicação; e segue o modelo cliente-servidor.

As aplicações clientes na Web são usualmente navegadores (*browsers*) responsáveis pela apresentação de documentos HTML e pela solicitação de um recurso a servidores Web. O recurso pode ser um documento HTML ou um arquivo contendo informação texto ou binária (imagem, áudio, vídeo, *applet* Java, etc), conforme estabelecido pelo elemento que originou a solicitação.

Não necessariamente o *software* navegador precisa saber manipular todos os tipos de recursos, podendo ele ativar rotinas auxiliares de exibição para tipos não reconhecidos. As rotinas auxiliares podem ser ativadas sob a forma de programas executáveis externos ao navegador ou através de *plugins*. O conceito de *plug-in* foi desenvolvido pela Netscape, sendo constituído por uma interface de programação (API) padronizada para ativar funcionalidades carregadas dinamicamente.

Servidores Web são responsáveis por atender a solicitações de clientes, por default operando na porta notável 80. As funcionalidades básicas de servidores Web incluem: organizar e gerenciar os recursos HTTP; prover acesso seguro a esses recursos; e processar *scripts* (extensões CGI, por exemplo — ver Seção 5.3).

Java também oferece suporte ao desenvolvimento de aplicações sobre a Web usando recursos do pacote `java.net`. Dentre os conceitos oferecidos pela linguagem Java para a programação distribuída há uma grande ênfase na programação direcionada para a Web, com suporte para a manipulação de recursos Web através de URL, a manipulação de conexões HTTP, a conversão de *strings* para o formato de codificação *www-urlencoded* e suporte à manipulação de conteúdos de diferentes tipos MIME.

Endereçamento por URL

Um endereço localizador uniforme de recursos (URL) é definido por uma sequência de caracteres que identifica de forma global e precisa um recurso na Web. A forma genérica de um URL é

`esquema:parte_específica`

onde o `esquema` identifica o protocolo utilizado para manipular o recurso, tais como `http`, `ftp`, `mailto`, `telnet` e `nntp`. A parte específica descreve o endereço do recurso de acordo com a infraestrutura e o tipo de recurso.

Na Internet, a parte específica de um URL toma a forma genérica

`//user:password@host:port/url-path`

onde nem todos os campos devem necessariamente estar presentes para todos os recursos endereçados. Quando `port` é omitido, a porta notável para o esquema especificado é utilizada (21 para `ftp`, 23 para `telnet`, 25 para `mailto`, 80 para `http`). O campo `url-path` é dependente do esquema.

A classe `java.net.URL` oferece a funcionalidade de nível mais alto (menor detalhamento) para a especificação de recursos Web. Cada recurso está associado a um objeto dessa classe, sendo que o localizador (URL) do recurso é especificado na construção do objeto.

Uma vez que o objeto URL esteja instanciado, há três maneiras de realizar a transferência do conteúdo do recurso para a aplicação local.

Na primeira forma, através do método `openStream()`, obtém-se um fluxo de leitura de bytes que permite transferir o conteúdo do recurso. Outra possibilidade é usar o método `openConnection()`, que retorna um objeto da classe (abstrata) `URLConnection`. Esta classe permite manipular um maior número de detalhes referentes à conexão URL, tais como obter dimensão, tipo e codificação do conteúdo, manipulação do conteúdo associado a um stream de entrada, obtenção do cabeçalho e outras. Finalmente, é possível usar o método `getContent()`, que obtém o conteúdo do recurso diretamente. Nesse caso, um objeto `ContentHandler` específico para o tipo de recurso recebido será ativado.

Conexões HTTP

O protocolo no nível da aplicação para a transferência de hipertexto (HTTP, *HyperText Transfer Protocol*) opera sobre o protocolo TCP/IP para estabelecer um mecanismo de serviço com estrutura requisição-resposta. Uma das características peculiares de HTTP é a composição flexível do cabeçalho, composto por diversas linhas, o que permite sua utilização como integrador de diversos formatos e não apenas de documentos HTML.

Essa flexibilidade reflete-se também na maior complexidade desse protocolo. No entanto, é possível estabelecer servidores HTTP operando com configurações simplificadas, onde nem todos os serviços previstos no protocolo são implementados.

Os principais serviços de HTTP incluem:

GET: solicita ao servidor o envio de um recurso; é o serviço essencial para o protocolo.

HEAD: variante de GET que solicita ao servidor o envio apenas de informações sobre o recurso.

PUT: permite que o cliente autorizado armazene ou altere o conteúdo de um recurso mantido pelo servidor.

POST: permite que o cliente envie mensagens e conteúdo de formulários para servidores que irão manipular a informação de maneira adequada.

DELETE: permite que o cliente autorizado remova um recurso mantido pelo servidor.

Um cabeçalho HTTP é composto por uma linha contendo a especificação do serviço e recurso associado, seguida por linhas contendo parâmetros. Um exemplo de requisição gerada por um cliente HTTP é:

```
GET http://www.dca.fee.unicamp.br/  
Accept: text/html, image/gif, image/jpeg  
User-Agent: Mozilla/3.0
```

para a qual o cabeçalho da resposta poderia ser:

```
HTTP/1.1 200 OK  
Date: Wed, 24 Mar 1999 23:23:45 GMT  
Server: Apache/1.2b6  
Connection: close  
Content-Type: text/html  
Content-length: 648
```

A indicação do tipo de conteúdo do recurso (usada nos parâmetros `Accept` e `Content-Type`) segue a especificação no padrão MIME (*Multipurpose Internet Mail Extensions*).

A classe `URLConnection` é uma especialização da classe `URLConnection`. Quando um objeto da classe `URL` invoca `openConnection()` é esse o tipo de conexão retornada quando o protocolo é HTTP.

Além das funcionalidades de conexões URL, essa classe define várias constantes associadas especificamente ao protocolo HTTP (tais como os códigos de erros) e alguns poucos métodos específicos de conexão HTTP.

Codificação e decodificação de dados

A tradução de strings para o formato esperado por um servidor Web a partir de um formulário, *x-www-form-urlencoded*, é suportada através da classe `URLEncoder`. Essa classe oferece o método estático `encode(String s)`, retornando uma *string* com o conteúdo codificado do argumento.

O formato *www-urlencoded* agrega em uma única *string* uma série de pares na forma `atributo=valor` separados pelo símbolo `'&'`. Nesse formato, espaços são convertidos para o símbolo `'+'` e caracteres com conotação especial — tais como `+`, `=` e `&` — são representados por sequência de escape `'%xx'` para a representação hexadecimal do valor ASCII do caráter. Assim, o caráter `'='` que faça parte do nome de um atributo ou parte do conteúdo de um valor será codificado na *string* na forma `'%3d'`.

O processo de tradução a partir de uma *string* nesse formato, que seria o necessário para implementar um serviço em Java que recebesse dados de um formulário, é oferecido pela classe `URLDecoder`, através do método estático `decode(String)`.

5.2 Acesso a bancos de dados

A linguagem Java vem se destacando como uma alternativa viável para a integração de aplicações novas e legadas na Internet. Essa infra-estrutura de integração não estaria completa se não contemplasse o acesso a sistemas de bancos de dados.

Um sistema de banco de dados é constituído por uma coleção organizada de dados (a base de dados) e pelo *software* que coordena o acesso a esses dados (o sistema gerenciador de banco de dados, ou SGBD). Utilizar um sistema de banco de dados ao invés de simples arquivos para armazenar de forma persistente os dados de uma aplicação apresenta diversas vantagens, das quais se destacam:

- o desenvolvedor da aplicação não precisa se preocupar com os detalhes do armazenamento, trabalhando com especificações mais abstratas para a definição e manipulação dos dados;
- tipicamente, um SGBD incorpora mecanismos para controle de acesso concorrente por múltiplos usuários e controle de transações;
- é possível compartilhar dados comuns entre mais de uma aplicação mesmo que a visão dos dados de cada aplicação seja distinta.

Existem soluções para integrar aplicações Java a bancos de dados orientados a objetos e a bancos de dados relacionais. Neste caso, a solução é padronizada através da especificação JDBC.

5.2.1 Bancos de dados relacionais

Um banco de dados relacional organiza seus dados em relações. Cada relação pode ser vista como uma tabela, onde cada coluna corresponde a atributos da relação e as linhas correspondem às tuplas ou elementos da relação. Em uma nomenclatura mais próximas àquela de sistemas de arquivos, muitas vezes as tuplas são denominadas registros e os atributos, campos.

Um conceito importante em um banco de dados relacional é o conceito de atributo chave, que permite identificar e diferenciar uma tupla de outra. Através do uso de chaves é possível acelerar o acesso a elementos (usando índices) e estabelecer relacionamentos entre as múltiplas tabelas de um sistema de banco de dados relacional.

Essa visão de dados organizados em tabelas oferece um conceito simples e familiar para a estruturação dos dados, sendo um dos motivos do sucesso de sistemas relacionais de dados. Certamente, outros motivos para esse sucesso incluem o forte embasamento matemático por trás dos conceitos utilizados em bancos de dados relacionais e a uniformização na linguagem de manipulação de sistemas de bancos de dados relacionais através da linguagem SQL.

Sob o ponto de vista matemático, uma relação é o subconjunto do produto cartesiano dos domínios da relação. Sendo um conjunto, é possível realizar operações de conjuntos — tais como união, interseção e diferença — envolvendo duas relações de mesma estrutura.

No entanto, um dos pontos mais fortes do modelo relacional está nos mecanismos de manipulação estabelecidos pela Álgebra Relacional. Os três principais operadores da álgebra relacional são seleção, projeção e junção.

A operação de seleção tem como argumento uma relação e uma condição (um predicado) envolvendo atributos da relação e/ou valores. O resultado é uma outra relação contemplando apenas as tuplas para as quais a condição foi verdadeira.

A operação de projeção tem como argumento uma relação e uma lista com um subconjunto dos atributos da relação. O resultado é outra relação contendo todas as tuplas da relação mas apenas com os atributos especificados.

Observe que se a lista de atributos não englobar a chave da relação, o resultado dessa operação poderia gerar tuplas iguais. Sob o ponto de vista estritamente matemático, os elementos duplicados devem ser eliminados, pois não fazem sentido para um conjunto.

A operação de junção recebe como argumentos duas relações e uma condição (um predicado) envolvendo atributos das duas relações. O resultado é uma relação com os atributos das duas relações contendo as tuplas que satisfizeram o predicado especificado. A operação de junção não é uma operação primitiva, pois pode ser expressa em termos da operação de produto cartesiano e da seleção, mas é uma das operações mais poderosas da álgebra relacional.

A forma mais usual de junção é aquela na qual a condição de junção é a igualdade entre valores de dois atributos das relações argumentos. Essa forma é tão usual que recebe o nome de junção natural. Nesse caso, o atributo comum aparece apenas uma vez na relação resultado, já que ele teria para todas as tuplas o mesmo valor nas duas colunas.

5.2.2 SQL

SQL é uma linguagem padronizada para a definição e manipulação de bancos de dados relacionais. Tipicamente, um SGBD oferece um interpretador SQL que permite isolar a aplicação dos detalhes de armazenamento dos dados. Se o projetista da aplicação tiver o cuidado de usar apenas as construções padronizadas de SQL, ele poderá desenvolver a aplicação sem se preocupar com o produto SGBD que estará sendo utilizado depois.

As três componentes de SQL são:

1. uma linguagem de definição de dados (DDL) para definir e revisar a estrutura de bancos de dados relacionais;
2. uma linguagem de controle de dados (DCL) para especificar mecanismos de segurança e integridade dos dados; e
3. uma linguagem de manipulação de dados (DML) para ler e escrever os dados.

A DDL supre as facilidades para a criação e manipulação de esquemas relacionais. Uma das necessidades de uma aplicação que irá armazenar seus dados em um sistema de banco de dados relacional é como criar uma identidade para o conjunto de tabelas de sua aplicação. Esse mecanismo não é padronizado em SQL, podendo variar de fornecedor a fornecedor de banco de dados. Algumas possibilidades incluem

```
CREATE SCHEMA name [AUTHORIZATION {user | group}]
```

ou

```
CREATE DATABASE name [On {default | device}]  
[Log On device [= size]]
```

Para criar uma tabela, o comando `CREATE TABLE` é utilizado:

```
CREATE TABLE name ( ATTRIBUTE DOMAIN [UNIQUE] [{NULL | NOT NULL}])
```

Alguns fabricantes adotam, ao invés da forma `UNIQUE` para indicação de chave da relação, um comando `Primary Key` após a criação da tabela para indicar qual atributo formará ou quais atributos comporão a chave primária da relação.

O campo `ATTRIBUTE` especifica o nome para a aplicação da coluna da tabela, enquanto `DOMAIN` especifica o tipo de dado para a coluna. Alguns tipos de dados usuais em SQL são `INTEGER`,

DECIMAL(p,q) — q dos p dígitos são casas decimais, FLOAT(p) — p é a precisão, CHARACTER(n) — string de n caracteres, BIT(n) — arranjo de n valores booleanos, DATE, TIME e TIMESTAMP.

Além desses comandos, ALTER TABLE permite modificar a estrutura de uma tabela existente e DROP TABLE permite remover uma tabela.

O principal comando da DML de SQL é SELECT. Por exemplo, para obter todos os dados de uma relação utiliza-se a forma básica:

```
SELECT * FROM table
```

Através do mesmo comando, é possível especificar projeções e seleções de uma tabela:

```
SELECT columns FROM table WHERE condition
```

A condição pode envolver comparações entre atributos e/ou valores constantes, podendo para tanto utilizar comparadores tais como igual (=), maior que (>), menor que (<), maior ou igual que (>=), menor ou igual que (<=), diferente (!= ou <>) e comparadores de strings (LIKE 'string', onde 'string' pode usar os caracteres '_' e '%' para indicar um caráter qualquer ou uma seqüência qualquer de caracteres, respectivamente). As comparações podem ser combinadas através dos conectivos lógicos And, Or e Not.

É possível expressar as quatro operações aritméticas (+, -, *, /), tanto para a condição como para a especificação de recuperação dos dados.

É possível também especificar a ordem desejada de apresentação dos dados, usando para tal a forma SELECT...ORDER BY.... Uma alternativa a esse comando é SELECT...GROUP BY..., que ordena os dados e não apresenta elementos que tenham o mesmo valor para a cláusula de agrupamento.

Uma importante categoria de funções de SQL incluem as funções de agregação, que permitem computar a média (Avg), a quantidade (Count), o maior ou menor valor (Max ou Min) e o total (Sum) das expressões especificadas.

Através do mesmo comando Select, é possível especificar consultas envolvendo múltiplas tabelas, como em

```
SELECT nome, data FROM Pedidos, Clientes
WHERE Pedidos.NumCliente = Clientes.NumCliente
```

Nesse caso, a junção das duas tabelas Pedidos e Clientes é realizada tendo como Num-Cliente como atributo de junção.

É possível especificar consultas internas a uma consulta, como em

```
SELECT NumProduto From Produtos
WHERE PrecoUnit > (SELECT Avg(PrecoUnit) FROM Produtos)
```

Para qualificar os resultados de uma subconsulta, podem ser utilizadas as funções All (a condição foi verdadeira para todos os elementos resultantes da subconsulta), Any (verdade para pelo menos um elemento), Exists (verdade se resultado da consulta tem pelo menos um elemento), In (verdade se o valor especificado faz parte do resultado), Not Exists (verdade se subconsulta teve resultado vazio) e Not In (verdade se valor especificado não faz parte do resultado da subconsulta).

SQL oferece ainda mecanismos para inserir elementos em uma tabela,

```
INSERT INTO table (columns) VALUES (values)
```

para modificar valores de colunas de um elemento,

```
UPDATE table SET column = value [, column = value]*  
WHERE condition
```

e para remover elementos de uma tabela,

```
DELETE FROM table  
WHERE condition
```

SQL pode ser utilizado diretamente pelo usuário, quando o SGBD oferece um interpretador SQL interativo, ou através de comandos embutidos em uma aplicação desenvolvida em uma linguagem de programação. No caso dessa linguagem ser Java, a forma de interagir com o banco de dados é especificado por JDBC.

5.2.3 JDBC

Java permite o acesso a bancos de dados relacionais através das funcionalidades definidas no pacote `java.sql`, que define o “produto JDBC”.

JDBC é uma API para execução e manipulação de resultados a consultas SQL através de Java. Para desenvolver uma aplicação com Java e bancos de dados relacionais, é preciso ter disponível:

- O pacote JDBC (padrão na distribuição da plataforma de desenvolvimento Java desde sua versão 1.1);
- Acesso ao servidor, o sistema gerenciador de banco de dados que entende SQL; e
- O *driver* JDBC adequado ao tipo de SGBD acessado.

Uma vez que esses recursos estejam disponíveis, a aplicação Java tem acesso ao banco de dados relacional através da execução dos seguintes passos:

1. Habilitar o *driver*;
2. Estabelecer uma conexão com o banco de dados;
3. Executar consulta SQL; e
4. Apresentar resultados da consulta.

Driver JDBC

Do ponto de vista da aplicação Java, um *driver* nada mais é do que uma classe cuja funcionalidade precisa ser disponibilizada para a aplicação. A funcionalidade básica que um *driver* deve oferecer é especificada através da interface `Driver`.

A classe `DriverManager` estabelece um conjunto básico de serviços para a manipulação de *drivers* JDBC. Como parte de sua inicialização, essa classe tentará obter o valor da propriedade

`jdbc.drivers` de um arquivo de definição de propriedades e carregar os *drivers* especificados pelos nomes das classes.

Alternativamente, um *driver* pode ser carregado explicitamente para a JVM; a forma usual para executar essa tarefa é através do método `forName()` da classe `Class`, como em

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Conexão com banco de dados

Uma vez que o *driver* esteja carregado, a aplicação Java pode estabelecer uma conexão com o gerenciador de banco de dados. Para especificar com qual banco de dados deseja-se estabelecer a conexão, é utilizada uma *string* na forma de um URL na qual o protocolo é `jdbc:` e o restante da *string* é dependente do *driver*. Por exemplo, o *driver* `jdbc:odbc` especifica o formato `jdbc:odbc:dsn`, onde `dsn` é o *data source name* que identifica o banco de dados.

Identificado o banco de dados, a sessão a ser estabelecida para o acesso ao banco de dados será controlada por um objeto de uma classe que implementa a interface `Connection`. O `DriverManager` oferece o método `getConnection()` para executar essa tarefa.

O encerramento de uma sessão é sinalizado pelo método `close()` da conexão:

```
import java.sql.*;
...
String DB = "jdbc:...";
...
Connection c = DriverManager.getConnection(DB);
...
c.close(); // encerra a sessão
```

Execução da consulta

Estabelecida a conexão ao banco de dados, é possível criar uma consulta e executá-la a partir da aplicação Java. Para representar uma consulta, o JDBC utiliza um objeto de uma classe que implementa a interface `Statement`. Um objeto dessa classe pode ser obtido através do método `createStatement()` da classe `Connection`.

Uma vez que um objeto `Statement` esteja disponível, é possível aplicar a ele o método `executeQuery()`, que recebe como argumento uma *string* representando uma consulta SQL.

O resultado da execução da consulta é disponibilizado através de um objeto `ResultSet`.

```
import java.sql.*;
...
Connection c;
c = ...;
Statement s = c.createStatement();
String query;
query = ...;
ResultSet r = s.executeQuery(query);
...
s.close();
```

Os métodos da interface `ResultSet` permitem a manipulação dos resultados individuais de uma tabela de resultados. Métodos como `getDouble()`, `getInt()`, `getString()` e `getTime()`, que recebem como argumento a especificação de uma coluna da tabela, permitem acessar o valor da coluna especificada na tupla corrente para os diversos tipos de dados suportados.

Para varrer a tabela, um cursor é mantido. Inicialmente, ele está posicionado antes do início da tabela, mas pode ser manipulado pelos métodos `first()`, `next()`, `previous()`, `last()` e `absolute(int row)`.

Por exemplo,

```
ResultSet r = s.executeQuery("Select * from Clientes");
System.out.println("ID          NOME");
while (r.next())
    System.out.println(r.getString("ClienteID")+
                       "    " + r.getString("Nome"));
r.close();
```

Para lidar com atributos que podem assumir valores nulos, o método `wasNull()` é oferecido. Ele retorna verdadeiro quando o valor obtido pelo método `getXXX()` for nulo, onde `XXX` é um dos tipos SQL.

A interface `ResultSetMetaData` permite obter informação sobre a tabela com o resultado da consulta. Um objeto desse tipo pode ser obtido através da aplicação do método `getMetaData()` ao `ResultSet`:

```
ResultSetMetaData m = r.getMetaData();
```

Uma vez obtido esse objeto, a informação desejada pode ser obtida através de métodos tais como `getColumnCount()`, `getColumnLabel()`, `getColumnTypeName()` e `getColumnType()`. O último método retorna tipos que podem ser identificados a partir de constantes definidas para a classe `java.sql.Types`.

Além da forma `Statement`, JDBC oferece duas formas alternativas que permitem respectivamente ter acesso a comandos SQL pré-compilados (`PreparedStatement`) e a procedimentos armazenados no banco de dados (`CallableStatement`).

Exemplo completo

Este exemplo ilustra o mecanismo básico para uma aplicação Java acessar um banco de dados. O objetivo é apresentar o conteúdo da seguinte relação, a tabela “notas” do banco de dados “poojava”:

```
> java PoojavaDB
```

fname	lname	activ	grd
Joao	Silva	1	6
Joao	Silva	2	7
Pedro	Souza	1	8
Pedro	Souza	2	5
Maria	Santos	1	7
Maria	Santos	2	8

O resultado acima foi obtido a partir da execução da seguinte aplicação, acessando um gerenciador de banco de dados PostgreSQL:

```
1  import java.sql.*;
2  public class PoojavaDB {
3      public static void main(String[] args) {
4          try {
5              // carrega driver
6              Class.forName("postgresql.Driver");
7              // estabelece conexao com banco de dados
8              Connection c =
9                  DriverManager.getConnection("jdbc:postgresql:poojava",
10                      "ricarte","");
11              // monta e executa consulta
12              Statement s = c.createStatement();
13              ResultSet r = s.executeQuery("Select * from notas");
14              // apresenta estrutura da tabela
15              ResultSetMetaData m = r.getMetaData();
16              int colCount = m.getColumnCount();
17              for (int i=1; i<=colCount; ++i)
18                  System.out.print(m.getColumnName(i) + "\t\t");
19              System.out.println();
20              // apresenta resultados da consulta
21              while (r.next())
22                  System.out.println(r.getString(1) + " " +
23                      r.getString(2) + " " +
24                      r.getInt(3) + "\t\t" +
25                      r.getInt(4));
26              r.close();
27              // fecha conexao com banco de dados
28              c.close();
29          }
30          catch (Exception e) {
31              System.err.println(e);
32          }
33      }
34  }
```

5.3 Servlets

Servlets oferecem uma maneira alternativa a CGI para estender as funcionalidades de um servidor Web. Na verdade, a API de *servlet* de Java oferece mecanismos adequados à adaptação qualquer servidor baseado em requisições e respostas, mas é em aplicações Web que *servlets* têm sido mais utilizados.

CGI (*Common Gateway Interface*) é a especificação de uma interface que permite que servidores Web tenham acesso a funcionalidades oferecidas por programas executando no ambiente da máquina servidora. Através de programas conectados a essa interface é possível por exemplo conectar uma base de dados à Web ou gerar dinamicamente o conteúdo de uma página HTML.

O servidor Web reconhece uma requisição CGI quando o URL especificado na solicitação identifica um arquivo executável (programa ou *script*) localizado em um diretório específico dentro do espaço Web de recursos disponibilizados aos clientes. Parâmetros podem ser repassados ao programa CGI especificando-os no URL, separados do nome do recurso pelo caráter ' ? '.

Tipicamente um programa CGI pode ser desenvolvido em qualquer linguagem de programação que tenha acesso à leitura de variáveis de ambiente e à manipulação dos *streams* padrões de entrada e saída de dados do sistema operacional (*stdin*, *System.in*; *stdout*, *System.out*).

Com o uso de *servlets*, a arquitetura da Web torna-se um base atrativa para o desenvolvimento de aplicações distribuídas em Java. A utilização de browsers HTML simplifica o desenvolvimento das aplicações cliente. Servidores Web suportam os mecanismos básicos de conexão ao cliente. Assim, o desenvolvimento irá se concentrar na extensão dos serviços através dos *servlets*.

5.3.1 Ciclo de vida de um *servlet*

A execução de um *servlet* não difere muito de uma aplicação CGI em sua forma de interação com o servidor. As quatro principais etapas nessa interação são:

1. cliente envia solicitação ao servidor;
2. servidor invoca *servlet* para a execução do serviço solicitado;
3. *servlet* gera o conteúdo em resposta à solicitação do cliente; e
4. servidor envia resultado do *servlet* ao cliente.

Quando um *servlet* é carregado pela primeira vez para a máquina virtual Java do servidor, o seu método *init()* é invocado. Esse método tipicamente prepara recursos para a execução do serviço (por exemplo, abrir arquivos ou ler o valor anterior de um contador de número de acessos) ou estabelece conexão com outros serviços (por exemplo, com um servidor de banco de dados). O método *destroy()* permite liberar esses recursos (fechar arquivos, escrever o valor final nessa sessão do contador de acessos), sendo invocado quando o servidor estiver concluindo sua atividade.

Uma diferença fundamental entre um *servlet* e uma aplicação CGI é que a classe que implementa o *servlet* permanece carregada na máquina virtual Java após concluir sua execução. Um programa CGI, ao contrário, inicia um novo processo a cada invocação — por este motivo, CGI deve utilizar mecanismos adicionais para manter o estado entre execuções, sendo a forma mais comum a utilização de arquivos em disco. Com um *servlet*, tais mecanismos são necessários apenas na primeira vez que é carregado e ao fim da execução do servidor, ou eventualmente como um mecanismo de *checkpoint*.

Servlets também oferecem como vantagem o fato de serem programas Java. Assim, eles permitem a utilização de toda a API Java para a implementação de seus serviços e oferecem adicionalmente portabilidade de plataforma.

5.3.2 Fundamentos da API de *servlets*

O suporte a *servlets* é uma extensão padronizada ao pacote Java, não sendo parte da distribuição básica do Java SDK. Assim, quem quiser desenvolver *servlets* deve obter o JSDK, o *Java Servlet Development Kit*. Adicionalmente, o servidor Web deve suportar o acesso a *servlets*, o que pode ocorrer de forma nativa (como no caso do *Java Web Server*) ou através de módulos *add-on* (como no caso de *apache*).

O JSDK inclui as classes que implementam a API de *servlets* organizadas em pacotes, dos quais os principais são `javax.servlet` e `javax.servlet.http`. Adicionalmente, uma aplicação `servletrunner` permite desenvolver e testar *servlets* antes de integrá-los a servidores.

Os três mecanismos alternativos básicos para criar um *servlet* são:

1. Estender a classe `javax.servlet.GenericServlet`, quando o *servlet* não for implementar nenhum protocolo específico de comunicação;
2. Estender a classe `javax.servlet.HttpServlet`, para *servlets* que manipulam dados específicos do protocolo HTTP; ou
3. Implementar a interface `javax.servlet.Servlet`.

Na primeira forma básica de implementar um *servlet*, estendendo a classe `GenericServlet`, o método `service()` deve ser definido. Esse método descreve o serviço que o *servlet* estará oferecendo.

Esse exemplo ilustra o código de um *servlet* que responde à solicitação de serviço com uma mensagem fixa:

```
1  import javax.servlet.*;
2  import java.io.*;
3  public class OiServlet extends GenericServlet {
4      public void service(ServletRequest solicitacao,
5                          ServletResponse resposta)
6          throws ServletException, IOException {
7          resposta.setContentType("text/plain");
8          PrintWriter saida = resposta.getWriter();
9          saida.println("Oi!");
10     }
11 }
```

Nesse exemplo, o método `getWriter()` é utilizado para estabelecer o canal de envio de dados desde o *servlet* — no caso, um objeto `PrintWriter`, permitindo o envio de textos. Alternativamente, dados binários poderiam ser enviados através de um `OutputStream`, obtido através do método `getOutputStream()`.

A classe `HttpServlet` é uma extensão de `GenericServlet` especificamente projetada para a conexão de *servlets* a servidores HTTP. Assim, métodos dedicados a lidar com solicitações HTTP, tais como `doGet()`, `doPost()` e `doPut()`, são definidos. A implementação padrão do método `service()` reconhece qual o tipo de solicitação recebida e invoca o método correspondente.

Este exemplo ilustra a utilização de um *servlet* que envia uma mensagem fixa no corpo de uma página HTML em resposta a uma requisição GET ao servidor Web, usando para tal o método `doGet()`:

```
1  import javax.servlet.*;
2  import javax.servlet.http.*;
3  import java.io.*;
4  public class OiHttpServlet extends HttpServlet {
5      public void doGet(HttpServletRequest request,
6                          HttpServletResponse response)
7          throws ServletException, IOException {
8          response.setContentType("text/html");
9          PrintWriter saida = response.getWriter();
10         saida.println("<HTML>");
11         saida.print("<HEAD><TITLE>");
12         saida.print("Resposta do servlet");
13         saida.println("</TITLE></HEAD>");
14         saida.println("<BODY><P>Oi!</P></BODY>");
15         saida.println("</HTML>");
16     }
17 }
```

Outros métodos que suportam a interação do *servlet* através de solicitações HTTP incluem `getLastModified()`, que é invocado pelo servidor para obter a data da última modificação do “documento” (um número negativo se não houver informação ou `long` com o número de segundos desde 1 de janeiro de 1970 GMT); e `getServletInfo()`, que retorna uma *string* de documentação sobre o *servlet*, tal como nome, autor e versão.

A passagem de dados de um formulário do cliente para o *servlet* pode se dar através do método `getParameter()`, que permite obter uma *string* com o valor do campo especificado. Por exemplo, se um formulário HTML especificasse um campo de texto para entrada do nome do usuário, como em

```
<INPUT type="text" name="username" size=20>
```

esse valor poderia ser obtido no código do *servlet* do exemplo anterior através da invocação

```
String nome = solicitacao.getParameter("username");
```

Além de `getParameter()`, o método `getParameterValues()` retorna um arranjo de *strings* com todos os valores de um determinado parâmetro. Outros métodos são oferecidos para obter informação das aplicações que estão invocando o *servlet*, tais como `getRemoteHost()`, `getServerName()`, `getServerPort()` e, especificamente para HTTP, `getHeaderNames()` e `getHeader()`.

A forma de integrar o *servlet* ao servidor Web é dependente da implementação; por exemplo, alguns servidores especificam um diretório (tal como `servlet`) onde as classes *servlets* são concentradas e a invocação dá-se pela invocação direta da URL, como em

```
http://site/servlet/OiHttpServlet
```

5.4 Programação com objetos distribuídos

Na programação distribuída usando a arquitetura cliente-servidor, clientes e servidores podem ser implementados usando qualquer paradigma de programação. Assim, é possível que um serviço específico seja executado por um método de algum objeto. No entanto, mesmo que o cliente também tenha sido desenvolvido orientação a objetos, na comunicação entre o cliente e o servidor esse paradigma deve ser esquecido, devendo ser utilizado algum protocolo pré-estabelecido de troca de mensagens para a solicitação e resposta ao serviço.

Um sistema de objetos distribuídos é aquele que permite a operação com objetos remotos. Dessa forma é possível, a partir de uma aplicação cliente orientada a objetos, obter uma referência para um objeto que oferece o serviço desejado e, através dessa referência, invocar métodos desse objeto — mesmo que a instância desse objeto esteja em uma máquina diferente daquela do objeto cliente.

O conceito básico que suporta plataformas de objetos distribuídos é o conceito de arquiteturas de objetos. Essencialmente, uma arquitetura orientada a objetos estabelece as regras, diretrizes e convenções definindo como as aplicações podem se comunicar e inter-operar. Dessa forma, o foco da arquitetura não é em como a implementação é realizada, mas sim na infra-estrutura e na interface entre os componentes da arquitetura.

Na plataforma Java, dois mecanismos são oferecidos para o desenvolvimento de aplicações usando o conceito de objetos distribuídos: Java RMI e Java IDL. RMI (invocação remota de métodos) é um mecanismo para desenvolver aplicações com objetos distribuídos que opera exclusivamente com objetos Java. Java IDL utiliza a arquitetura padrão CORBA para integração de aplicações Java a aplicações desenvolvidas em outras linguagens.

5.4.1 Arquiteturas de objetos distribuídos

No paradigma de arquiteturas de objetos, há três elementos principais. A arquitetura OO fornece uma descrição abstrata do *software* — que categorias de objetos serão utilizadas, como estarão particionados e como interagirão. As interfaces distribuídas são as descrições detalhadas das funcionalidades do *software*. Finalmente, a implementação é composta por módulos de *software* que suportam as funcionalidades especificadas nas interfaces distribuídas.

O uso de interfaces distribuídas permite isolar a arquitetura de um sistema de sua implementação. Dessa forma, o sistema pode ser construído com um alto grau de independência em relação às implementações específicas de suas funcionalidades, ou seja, é possível substituir implementações específicas com pequeno impacto sobre o sistema como um todo.

A adoção do paradigma de arquitetura de objetos permite também atingir um alto grau de interoperabilidade através da adoção de uma infra-estrutura padronizada de comunicação entre objetos através das interfaces. Assim, cada componente da arquitetura deve se preocupar apenas em como se dará sua comunicação com a infra-estrutura de comunicação, estabelecida através de um objeto wrapper. Sem essa padronização, seria necessário estabelecer os mecanismos de comunicação com todos os demais componentes do sistema.

Em uma arquitetura de objetos, alguma forma deve ser estabelecida para que clientes possam localizar serviços que estão sendo oferecidos. Isso é usualmente oferecido na forma de um serviço básico da plataforma de objetos distribuídos. Do ponto de vista de quem oferece o serviço, é preciso habilitar o objeto para que seus métodos possam ser invocados remotamente. Isto é realizado através das seguintes atividades:

Descrever o serviço. Na arquitetura de objetos, a descrição ou especificação de serviços é determinada através das interfaces. Em Java, isto é realizado através da especificação oferecida por uma interface.

Implementar o serviço. Isto é realizado através do desenvolvimento de uma classe Java que implemente a interface especificada.

Anunciar o serviço. Quando um objeto que implementa o serviço torna-se ativo, é preciso que ele seja registrado em um “diretório de serviços” de forma que potenciais clientes possam localizá-lo.

Sob o ponto de vista do objeto cliente, que vai usar o serviço do objeto remoto, é preciso **localizar o serviço**, o que é feito acessando o registro de serviços. Portanto, é necessário que servidores e clientes estejam de acordo com o local (a máquina) onde o registro é realizado. Como resultado dessa tarefa, obtém-se uma referência ao objeto remoto que pode ser utilizada como se fosse uma referência para o objeto local.

A utilização desse mecanismo de localizar o serviço através de um diretório ou registro permite que as máquinas clientes ignorem totalmente em que máquinas os serviços solicitados estão operando — uma facilidade conhecida como transparência de localização.

Um dos principais objetivos em uma plataforma de objetos distribuídos é atingir transparência de localização, tornando uniforme a forma de utilização de objetos independentemente desses objetos estarem na máquina local da aplicação ou em máquinas distintas.

A fim de que se atinja transparência de localização, as seguintes funcionalidades devem ser oferecidas:

1. Localizar e carregar classes remotas;
2. Localizar e obter referências a objetos remotos; e
3. Habilitar a invocação de métodos de objetos remotos.

A primeira funcionalidade, não muito diferente do que ocorre em sistemas com objetos locais, é necessária para que a aplicação conheça as facilidades oferecidas pelo objeto remoto.

Referências a objetos também são utilizadas em sistemas com objetos locais, porém com diferenças significativas. Em um sistema local, as referências a objetos são tipicamente manipuladores com especificação de endereços de memória. No caso de objetos remotos, esses endereços da memória de outra máquina não têm validade na máquina local. Assim, é preciso oferecer mecanismos que traduzam essas referências entre máquinas de forma transparente para o programador.

Para a invocação de métodos de um objeto remoto, além da necessidade de se localizar a referência ao método é preciso oferecer mecanismos para tornar transparente a passagem de argumentos para e o retorno de valores desde o método.

Além dessas funcionalidades, a comunicação de falhas no oferecimento da transparência de localização ao programador é essencial. Assim, funcionalidades para comunicar exceções entre máquinas também devem ser suportadas pela plataforma de objetos distribuídos.

5.4.2 Java RMI

RMI (*Remote Method Invocation*) é uma das abordagens da tecnologia Java para prover as funcionalidades de uma plataforma de objetos distribuídos. Esse sistema de objetos distribuídos faz parte do núcleo básico de Java desde a versão JDK 1.1, com sua API sendo especificada através do pacote `java.rmi` e seus subpacotes.

Através da utilização da arquitetura RMI, é possível que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais.

A arquitetura RMI oferece a transparência de localização através da organização de três camadas entre os objetos cliente e servidor:

1. A camada de *stub/skeleton* oferece as interfaces que os objetos da aplicação usam para interagir entre si;
2. A camada de referência remota é o *middleware* entre a camada de *stub/skeleton* e o protocolo de transporte. É nesta camada que são criadas e gerenciadas as referências remotas aos objetos;
3. A camada do protocolo de transporte oferece o protocolo de dados binários que envia as solicitações aos objetos remotos pela rede.

Desenvolvimento da aplicação RMI

No desenvolvimento de uma aplicação cliente-servidor usando Java RMI, como para qualquer plataforma de objetos distribuídos, é essencial que seja definida a interface de serviços que serão oferecidos pelo objeto servidor.

A especificação de uma interface remota é equivalente à definição de qualquer interface em Java, a não ser pelos seguintes detalhes: a interface deverá, direta ou indiretamente, estender a interface `Remote`; e todo método da interface deverá declarar que a exceção `RemoteException` (ou uma de suas superclasses) pode ser gerada na execução do método.

Esse exemplo ilustra a definição de uma interface remota para um objeto que contém um contador inteiro:

```
1  import java.rmi.*;
2  public interface Count extends Remote {
3      void set(int val) throws RemoteException;
4      void reset() throws RemoteException;
5      int get() throws RemoteException;
6      int increment() throws RemoteException;
7  }
```

Esse contador é manipulado por quatro métodos: `set()`, para definir um valor inicial para o contador; `reset()`, para reiniciar o contador com o valor 0; `get()`, para consultar o valor do contador sem alterá-lo; e `increment()`, que lê o valor atual do contador e incrementa-o.

Os serviços especificados pela interface RMI deverão ser implementados através de uma classe Java. Nessa implementação dos serviços é preciso indicar que objetos dessa classe poderão ser acessados remotamente.

A implementação do serviço se dá através da definição de uma classe que implementa a interface especificada. No entanto, além de implementar a interface especificada, é preciso incluir as funcionalidades para que um objeto dessa classe possa ser acessado remotamente como um servidor.

A implementação da interface remota se dá da mesma forma que para qualquer classe implementando uma interface Java, ou seja, a classe fornece implementação para cada um dos métodos especificados na interface.

As funcionalidades de um servidor remoto são especificadas na classe abstrata `RemoteServer`, do pacote `java.rmi.server`. Um objeto servidor RMI deverá estender essa classe ou, mais especificamente, uma de suas subclasses. Uma subclasse concreta de `RemoteServer` oferecida no mesmo pacote é `UnicastRemoteObject`, que permite representar um objeto que tem uma única implementação em um servidor (ou seja, não é replicado em vários servidores) e mantém uma conexão ponto-a-ponto com cada cliente que o referencia.

Tipicamente, a declaração de uma classe que implementa um servidor remoto RMI terá a forma

```
public class ... extends UnicastRemoteObject implements ... {  
    ...  
}
```

Esse exemplo oferece uma possível implementação para a interface remota previamente especificada:

```
1  import java.rmi.*;  
2  import java.rmi.server.UnicastRemoteObject;  
3  public class CountImpl extends UnicastRemoteObject  
4      implements Count {  
5      private int sum;  
6      public CountImpl() throws RemoteException {  
7      }  
8      public void set(int val) throws RemoteException {  
9          sum = val;  
10     }  
11     public void reset() throws RemoteException {  
12         sum = 0;  
13     }  
14     public int get() throws RemoteException {  
15         return sum;  
16     }  
17     public int increment() throws RemoteException {  
18         return sum++;  
19     }  
20 }
```

Clientes e servidores RMI

Uma vez que a interface remota esteja definida e a classe que implementa o serviço remoto tenha sido criada, o próximo passo no desenvolvimento da aplicação distribuída é desenvolver o servidor

RMI, uma classe que crie o objeto que implementa o serviço e cadastre esse serviço na plataforma de objetos distribuídos.

Um objeto servidor RMI simples deve realizar as seguintes tarefas: criar uma instância do objeto que implementa o serviço; e disponibilizar o serviço através do mecanismo de registro.

O desenvolvimento de um cliente RMI requer essencialmente a obtenção de uma referência remota para o objeto que implementa o serviço, o que ocorre através do cadastro realizado pelo servidor. Uma vez obtida essa referência, a operação com o objeto remoto é indistinguível da operação com um objeto local.

Usando o serviço de nomes

O aplicativo `rmiregistry` faz parte da distribuição básica de Java. Tipicamente, esse aplicativo é executado como um processo de fundo (em *background*) que fica aguardando solicitações em uma porta, que pode ser especificada como argumento na linha de comando. Se nenhum argumento for especificado, a porta 1099 é usada como padrão.

O aplicativo `rmiregistry` é uma implementação de um serviço de nomes para RMI. O serviço de nomes é uma espécie de diretório, onde cada serviço disponibilizado na plataforma é registrado através de um nome do serviço, uma *string* única para cada objeto que implementa serviços em RMI.

Para ter acesso ao serviço de nomes a partir de uma classe Java, são oferecidos dois mecanismos básicos. O primeiro utiliza a classe `Naming`, do pacote `java.rmi`. O segundo mecanismo utiliza as facilidades oferecidas através das classes no pacote `java.rmi.registry`.

A classe `Naming` permite a realização da busca de um serviço pelo nome (*lookup*) usando o método estático `lookup(String nome)`, que retorna uma referência para o objeto remoto. O serviço de registro aonde a busca se realiza é especificado pela *string* usando uma sintaxe similar à URL:

```
rmi://objreg.host:port/objname
```

O protocolo padrão é `rmi`, sendo no momento o único suportado através desse método. Se não especificado, o *host* é a máquina local e a porta é 1099. O nome de registro do objeto é a única parte obrigatória desse argumento.

Além de `lookup()` os métodos `bind()`, `rebind()`, `unbind()` e `list()`, descritos na sequência, são também suportados.

Outra alternativa para ter acesso ao serviço de nomes a partir da aplicação Java é utilizar as funcionalidades do pacote `java.rmi.registry`, que oferece uma classe e uma interface para que classes Java tenham acesso ao serviço de nomes RMI.

A interface `Registry` representa uma interface para o registro de objetos RMI operando em uma máquina específica. Através de um objeto dessa classe, é possível invocar o método `bind()` que associa um nome de serviço (um `String`) ao objeto que o implementa.

Para obter uma referência para um objeto `Registry` são utilizados os métodos da classe `LocateRegistry`, todos estáticos, tais como `getRegistry()`. Há quatro versões básicas desse método:

1. `getRegistry()`: obtém referência para o registro local operando na porta default;
2. `getRegistry(int port)`: obtém referência para o registro local operando na porta especificada;

3. `getRegistry(String host)`: obtém referência para o registro remoto operando na porta default;
4. `getRegistry(String host, int port)`: obtém referência para o registro remoto operando na porta especificada.

O método estático `createRegistry(int port)` pode ser utilizado para iniciar um serviço de registro na máquina virtual Java corrente na porta especificada como argumento, retornando também um objeto da classe `Registry`.

Inicialmente, é preciso obter uma referência para o serviço de registro, através da invocação do método:

```
Registry r = LocateRegistry.getRegistry();
```

Observe que a referência para `Registry` é em si uma referência para um objeto remoto, uma vez que a interface `Registry` é uma extensão da interface `Remote`.

Uma vez que a referência para o serviço de registro tenha sido obtida, é possível acessar as funcionalidades desse serviço através dos métodos da interface `Registry`. Particularmente, para registrar um novo serviço utiliza-se o método `bind()`:

```
r.bind(serviceName, myCount);
```

O objeto que está sendo registrado deve implementar também a interface `Remote`, que identifica todos os objetos que podem ser acessados remotamente.

Outros serviços disponíveis através dos métodos de `Registry` incluem atualização, remoção e busca dos serviços lá registrados. Para atualizar um registro já existente, o método `rebind()` pode ser utilizado. Para eliminar um registro, utiliza-se o método `unbind()`. Dado o nome de um serviço, o objeto `Remote` que o implementa pode ser obtido pelo método `lookup()`. O método `list()` retorna um arranjo de `String` com os nomes de todos os serviços registrados.

Implementação do servidor RMI

Como observado, um objeto servidor RMI simples deve realizar as seguintes tarefas:

1. Criar uma instância do objeto que implementa o serviço; e
2. Disponibilizar o serviço através do mecanismo de registro.

Esse exemplo de servidor RMI para o contador remoto cria uma instância da implementação do serviço e coloca-a à disposição de potenciais clientes, registrando-o no *registry* RMI:

```
1 import java.rmi.registry.*;
2 public class CountServer {
3     public static void main(String[] args) {
4         try {
5             String serviceName = "Count001";
6             CountImpl myCount = new CountImpl();
7             Registry r = LocateRegistry.getRegistry();
```

```
8         r.bind(serviceName, myCount);
9         System.out.println("Count Server ready.");
10    }
11    catch (Exception e) {
12        System.out.println("Exception: " + e.getMessage());
13        e.printStackTrace();
14    }
15 }
16 }
```

Cliente RMI

A principal etapa no desenvolvimento de uma aplicação cliente RMI é a obtenção da referência remota para o objeto (remoto) que implementa o serviço desejado. Para tanto, o cliente RMI usa o serviço padrão oferecido pelo mecanismo de registro de nomes de serviços.

Uma vez que a referência remota seja obtida, ela pode ser convertida (*downcast*) para uma referência para a interface que especifica o serviço. A partir de então, os métodos oferecidos pelo serviço remoto são invocados da mesma forma que ocorre para objetos locais.

Esses exemplos ilustram o desenvolvimento de código cliente em RMI. No primeiro exemplo desenvolve-se um cliente RMI que simplesmente invoca o método `reset()` através de uma referência remota para o objeto servidor:

```
1  import java.rmi.registry.*;
2  public class CountReset {
3      public static void main(String args[]) {
4          try {
5              Registry r = LocateRegistry.getRegistry();
6              Count myCount = (Count) r.lookup("Count001");
7              myCount.reset();
8          }
9          catch(Exception e) {
10             e.printStackTrace();
11          }
12          System.exit(0);
13      }
14  }
```

Nesse outro exemplo, o cliente utiliza os métodos para modificar e obter o valor do contador remoto. Ele também ilustra a interação de um código com o registro RMI através da classe `Naming`:

```
1  import java.rmi.*;
2  public class CountClient {
3      public static void main(String args[]) {
4          try {
5              Remote remRef = Naming.lookup("Count001");
6              Count myCount = (Count) remRef;
```

```
7         int initValue = myCount.get();
8         System.out.print("De " + initValue + " para ");
9         long startTime = System.currentTimeMillis();
10        for (int i = 0 ; i < 1000 ; i++ )
11            myCount.increment();
12        long stopTime = System.currentTimeMillis();
13        System.out.println(myCount.get());
14        System.out.println("Avg Ping = "
15                            + ((stopTime - startTime)/1000f)
16                            + " msecs");
17    }
18    catch(Exception e) {
19        e.printStackTrace();
20    }
21    System.exit(0);
22 }
23 }
```

Esse terceiro exemplo ilustra a utilização de RMI a partir de um cliente desenvolvido como um *applet*. Nesse *applet*, um campo de texto mostra o valor do contador no objeto servidor. Dois botões são fornecidos, um para incrementar o valor mil vezes (*Start*) e outro para obter o valor atual do contador (*Get*):

```
1  import java.rmi.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.applet.*;
5  public class AppletClient extends Applet
6      implements ActionListener {
7      Count remCount;
8      TextField tfCnt;
9      Button bStart, bGet;
10     String bslabel = "Start";
11     String bglabel = "Get";
12     public void init() {
13         try {
14             setLayout(new GridLayout(2,2));
15             add(new Label("Count:"));
16             tfCnt = new TextField(7);
17             tfCnt.setEditable(false);
18             add(tfCnt);
19             bStart = new Button(bslabel);
20             bStart.addActionListener(this);
21             bGet = new Button(bglabel);
22             bGet.addActionListener(this);
```

```
23         add(bStart);
24         add(bGet);
25         showStatus("Binding remote object");
26         remCount = (Count) Naming.lookup("Count001");
27         tfCnt.setText(Integer.toString(remCount.get()));
28     }
29     catch (Exception e) {
30         e.printStackTrace();
31     }
32 }
33 public void paint() {
34     try {
35         tfCnt.setText(Integer.toString(remCount.get()));
36     }
37     catch (Exception e) {
38         e.printStackTrace();
39     }
40 }
41 public void actionPerformed (ActionEvent ev) {
42     try {
43         String botao = ev.getActionCommand();
44         if (botao.equals(bslabel)) {
45             showStatus("Incrementing...");
46             for (int i = 0 ; i < 1000 ; i++ )
47                 remCount.increment();
48             showStatus("Done");
49         }
50         else {
51             showStatus("Current count");
52             paint();
53         }
54     }
55     catch (Exception e) {
56         e.printStackTrace();
57     }
58 }
59 }
```

Definindo *stubs* e *skeletons*

Para que um serviço oferecido por um objeto possa ser acessado remotamente através de RMI, é preciso também as classes auxiliares internas de *stubs* e *skeletons*, responsáveis pela comunicação entre o objeto cliente e o objeto que implementa o serviço, conforme descrito na apresentação da arquitetura RMI.

Uma vez que a interface e a classe do serviço tenham sido criadas e compiladas para *byteco-*

des usando um compilador Java convencional, é possível criar os correspondentes *stubs* e *skeletons*. Para tanto, utiliza-se o aplicativo compilador RMI, `rmic`, disponibilizado juntamente com o *kit* de desenvolvimento Java.

Um exemplo ilustra o processo de compilação RMI para o serviço do contador remoto. Considere a implementação do serviço que foi previamente definida. O primeiro passo para a criação do *stub* e do *skeleton* para esse serviço é obter a classe compilada, que por sua vez precisa da classe da interface:

```
> javac Count.java
> javac CountImpl.java
```

Com a classe `CountImpl.class` disponível, a execução do comando

```
> rmic CountImpl
```

gera as classes `CountImpl_Stub.class` e `CountImpl_Skel.class`, correspondendo respectivamente ao *stub* e ao *skeleton* para o serviço. O *stub* deverá ser disponibilizado junto ao código do cliente RMI, enquanto que o *skeleton* deverá estar disponível junto ao código do servidor.

Uma classe *stub* oferece implementações dos métodos do serviço remoto que são invocadas no lado do cliente. Internamente, esses métodos empacotam *marshall*) os argumentos para o método e os envia ao servidor. A implementação correspondente no lado servidor, no *skeleton*, desempacota (*unmarshall*) os dados e invoca o método do serviço. Obtido o valor de retorno do serviço, o método no *skeleton* empacota e envia esse valor para o método no *stub*, que ainda estava aguardando esse retorno. Obtido o valor de retorno no *stub*, esse é desempacotado e retornado à aplicação cliente como resultado da invocação remota.

Internamente, o processo de *marshalling* utiliza o mecanismo de serialização de Java. Assim, argumentos e valores de retorno de métodos remotos invocados através de RMI estão restritos a tipos primitivos de Java e a objetos de classes que implementam `Serializable`.

Usando fábricas de objetos remotos

Pode haver situações em que não seja interessante registrar cada implementação de um serviço no *registry* — por exemplo, quando o servidor não sabe quantos objetos criar de antemão ou quando a quantidade de pequenos serviços registrados é tão grande que pode tornar a busca por um serviço ineficiente.

Nessas situações, pode ser interessante utilizar uma fábrica de objetos remotos. Nesse caso, o servidor que está registrado em `rmiregistry` não é uma implementação individual do serviço, mas sim um gerenciador de instâncias de implementação do serviço. Esse gerenciador deve implementar uma interface remota que permita que o cliente obtenha uma referência remota para o serviço desejado em duas etapas:

1. obtendo a referência para o gerenciador através da invocação do método `lookup()`; e
2. obtendo a referência para o serviço propriamente dito através da invocação do método do gerenciador que retorna a referência.

Esses exemplos usando contadores inteiros ilustram a utilização do conceito de fábrica de objetos remotos. Além da implementação do serviço e da sua correspondente interface, é preciso inicialmente definir uma interface para a fábrica. Nesse exemplo, essa interface especifica a funcionalidade de um “gerenciador de contadores”, que recebe o nome do contador e retorna uma referência remota para um objeto contador:

```
1  import java.rmi.*;
2  public interface CountManager extends Remote {
3      Count getCount(String nome) throws RemoteException;
4  }
```

No lado servidor, o que muda em relação ao exemplo anterior é que agora não é mais o objeto que implementa o contador que deve ser cadastrado no *registry*, mas sim o objeto fábrica, uma implementação da interface especificada para o “gerenciador de contadores”. Essa fábrica, por sua vez, mantém um registro interno dos objetos criados para poder retornar as referências solicitadas pelos clientes remotos. Essa classe combina as funcionalidades da implementação de uma interface remota com aquelas de um servidor RMI:

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3  import java.rmi.server.*;
4  import java.util.*;
5  public class CManagerImpl extends UnicastRemoteObject
6      implements CountManager {
7      private Hashtable counters = new Hashtable();
8      public CManagerImpl() throws RemoteException {
9      }
10     public Count getCount(String nome) throws RemoteException {
11         Count rem = null;
12         if (counters.containsKey(nome))
13             rem = (Count) counters.get(nome);
14         else {
15             rem = new CountImpl();
16             counters.put(nome, rem);
17             System.out.println("New counter: " + nome);
18         }
19         return rem;
20     }
21     public static void main(String[] args) {
22         try {
23             String serviceName = "CountFactory";
24             CManagerImpl myCM = new CManagerImpl();
25             Registry r = LocateRegistry.getRegistry();
26             r.bind(serviceName, myCM);
27             System.out.println("CountFactory ready.");
28         }
```

```
29     catch (Exception e) {
30         e.printStackTrace();
31     }
32 }
33 }
```

No lado do cliente há referências agora a duas interfaces remotas, uma para o “gerenciador de contadores” e outra para o contador. A primeira delas é resolvida através do serviço de registro do RMI, enquanto que a referência para o objeto do segundo tipo de interface é obtido a partir dessa referência para o gerenciador que foi obtida:

```
1  import java.rmi.*;
2  public class CountClient {
3      public static void main(String args[]) {
4          String nome = "Count001";
5          try {
6              CountManager cm =
7                  (CountManager) Naming.lookup("CountFactory");
8              if (args.length > 0)
9                  nome = args[0];
10             Count myCount = cm.getCount(nome);
11             int initValue = myCount.get();
12             System.out.print("De " + initValue + " para ");
13             long startTime = System.currentTimeMillis();
14             for (int i = 0 ; i < 1000 ; i++ )
15                 myCount.increment();
16             long stopTime = System.currentTimeMillis();
17             System.out.println(myCount.get());
18             System.out.println("Avg Ping = "
19                               + ((stopTime - startTime)/1000f)
20                               + " msecs");
21         }
22         catch(Exception e) {
23             e.printStackTrace();
24         }
25         System.exit(0);
26     }
27 }
```

Execução com RMI

A execução da aplicação cliente-servidor em RMI requer, além da execução da aplicação cliente e da execução da aplicação servidor, a execução do serviço de registro de RMI. Além do princípio básico de execução de aplicações RMI, a arquitetura RMI oferece facilidades para operação com código disponibilizado de forma distribuída e ativação dinâmica, além de outros serviços distribuídos.

O registro RMI (`rmiregistry`) executa isoladamente em uma máquina virtual Java. O servidor da aplicação, assim como a implementação do serviço, estão executando em outra máquina virtual Java; sua interação com o registro (ao invocar o método `bind()`) se dá através de uma referência remota. Da mesma forma, cada aplicação cliente pode ser executada em sua própria máquina virtual Java; suas interações com o registro (método `lookup()`) e com a implementação do serviço (usando os correspondentes *stub* e *skeleton*) dão-se também através de referências remotas.

Portanto, para executar uma aplicação RMI é preciso inicialmente disponibilizar o serviço de registro RMI. Para tanto, o aplicativo `rmiregistry` deve ser executado.

Com o `rmiregistry` disponível, o servidor pode ser executado. Para tanto, essa máquina virtual Java deverá ser capaz de localizar e carregar as classes do servidor, da implementação do serviço e do *skeleton*.

Após a execução do comando que ativa a aplicação servidor, a mensagem “Count Server ready.” deverá surgir na tela, indicando que o servidor obteve sucesso na criação e registro do serviço e portanto está apto a responder às solicitações de clientes.

Finalmente, com o servidor já habilitado para responder às solicitações, o código cliente pode ser executado. Essa máquina virtual deverá ser capaz de localizar e carregar as classes com a aplicação cliente, a interface do serviço e o *stub* para a implementação do serviço. Seria possível também ter várias ativações simultâneas de `CountClient` em diferentes máquinas virtuais Java.

No caso mais simples de execução, as diversas máquinas virtuais Java estarão executando em uma mesma máquina, compartilhando um `CLASSPATH` comum. No entanto, há mecanismos para permitir o carregamento de classes em uma aplicação RMI envolvendo classes remotas.

Operação com objetos em máquinas remotas

Na descrição da operação de aplicações distribuídas usando RMI, assumiu-se que as aplicações clientes, servidor e de registro eram processos distintos; porém considerou-se que todas as classes necessárias para a operação das aplicações estavam localizadas em algum diretório do `CLASSPATH` local.

No caso de execução em máquinas separadas, há duas formas de fazer a distribuição das classes de modo que clientes e servidores possam executar corretamente. Na primeira forma, a estratégia é distribuir explicitamente as classes necessárias e incluí-las em diretórios onde elas possam ser localizadas quando necessário. No lado cliente, essas classes complementares seriam a interface do serviço e o *stub* para a implementação do serviço. No lado servidor, seriam essas as classes de implementação do serviço e o correspondente *skeleton*.

A outra forma é utilizar os mecanismos de carregamento dinâmico de classes distribuídas, em alternativa ao *class loader* padrão da máquina virtual Java. Por exemplo, se a execução do cliente se dá através de um *applet*, o `AppletClassLoader` oferece as funcionalidades necessárias para localizar uma classe que está localizada no mesmo diretório de onde foi carregada a classe original.

Em RMI, há uma alternativa adicional de se utilizar o `RMIClassLoader`, que permite o carregamento de *stubs* e *skeletons* a partir de um URL (especificado através da propriedade `java.rmi.server.codebase`). Essa propriedade deve ser estabelecida para a máquina virtual Java que irá executar o servidor, como em

```
>java -Djava.rmi.server.codebase=http://mhost/mdir/ CountServer
```

Deste modo, quando o servidor realizar o cadastro do serviço no *registry*, esse *codebase* será embutido na referência do objeto. Quando o cliente obtiver a referência ao objeto remoto do *registry* e seu *class loader* falhar em localizar a classe *stub* no CLASSPATH local, sua máquina virtual Java fará uma conexão HTTP com *mhost* para obter a classe correspondente — assim como outras classes eventualmente necessárias para execução do serviço no lado cliente.

De forma similar, caso o *rmiregistry* estivesse operando em outra máquina, distinta daquela onde as aplicações clientes e servidor estivessem executando, seria necessário especificar no código das aplicações a máquina que executa *rmiregistry*, seja através do método `getRegistry()` da classe `LocateRegistry` ou através da especificação de URL no protocolo RMI nos métodos da classe `Naming`.

Como para qualquer situação na qual a máquina virtual Java irá carregar classes localizadas de forma distribuída, é preciso adicionalmente estabelecer qual a política de segurança para operar com código proveniente das outras máquinas. Essa política será enforcada pelo gerenciador de segurança, que pode ser definido pela invocação do método correspondente antes de qualquer invocação a métodos de RMI:

```
System.setSecurityManager(new RMISecurityManager());
```

O uso dessas facilidades pode ser apreciado nos exemplos modificados para o código do servidor e cliente da aplicação do contador distribuído. No caso do servidor:

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.rmi.registry.*;
4  import java.net.SocketPermission;
5  public class CountServer {
6      public static void main(String[] args) {
7          // Create and install the security manager
8          System.setSecurityManager(new RMISecurityManager());
9          catch (Exception e) {
10             e.printStackTrace();
11         }
12         try {
13             String serviceName = "Count001";
14             // Create CountImpl
15             CountImpl myCount = new CountImpl();
16             Registry r = LocateRegistry.getRegistry(args[0]);
17             r.rebind(serviceName, myCount);
18             System.out.println("Count Server ready.");
19         }
20         catch (Exception e) {
21             System.out.println("Exception: " + e.getMessage());
22             e.printStackTrace();
23         }
24     }
25 }
```

No caso da aplicação cliente, o código modificado é apresentado abaixo:

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3  public class CountClient {
4      public static void main(String args[]) {
5          // Create and install the security manager
6          System.setSecurityManager(new RMISecurityManager());
7          try {
8              Count myCount = (Count)Naming.lookup("rmi://" +
9                  args[0] + "/Count001");
10             // Calculate Start time
11             long startTime = System.currentTimeMillis();
12             // Increment 1000 times
13             System.out.print("Incrementing... ");
14             for (int i = 0 ; i < 1000 ; i++ )
15                 myCount.increment();
16             System.out.println(myCount.get());
17             // Calculate stop time; print out statistics
18             long stopTime = System.currentTimeMillis();
19             System.out.println("Avg Ping = "
20                 + ((stopTime - startTime)/1000f)
21                 + " msecs");
22         }
23         catch(Exception e) {
24             System.err.println("System Exception" + e);
25         }
26         System.exit(0);
27     }
28 }
```

A especificação da interface e a implementação do serviço permanecem inalteradas para esses exemplos.

Ativação dinâmica

Na primeira especificação de RMI (JDK 1.1), era necessário que um serviço oferecido por um objeto fosse explicitamente ativado em alguma máquina virtual Java e então fosse cadastrado em um serviço de registro.

O problema com essa abordagem ocorre principalmente quando, por algum motivo não previsto, o serviço torna-se indisponível — não há como sinalizar o cliente ou o registro sobre esse fato. O serviço de ativação, oferecido a partir de JDK 1.2, permite contornar essa limitação. As principais facilidades oferecidas pelo mecanismo de ativação remota incluem:

- a possibilidade de criar automaticamente um objeto remoto devido a solicitações de obtenção de referências ao objeto;

- o suporte a grupos de ativação, permitindo a ativação de vários objetos remotos executando em uma mesma máquina virtual;
- a possibilidade de reiniciar a execução de objetos remotos que tenham encerrado sua execução em decorrência de alguma falha do sistema.

Para tornar um objeto que implementa um serviço como remotamente ativável, é preciso satisfazer três requisitos. Primeiro, é preciso implementar o serviço como uma subclasse de `Activatable`, do pacote `java.rmi.activation`. Segundo, é preciso criar construtores de ativação na implementação do serviço. Finalmente, é preciso registrar o objeto e seu método de ativação no serviço de ativação.

A classe `Activatable` oferece construtores com argumentos específicos para o registro (no serviço de ativação) e a ativação de objetos (incluindo o URL onde o *bytecode* para o objeto pode ser localizado, um objeto da classe `MarshaledObject` representando os argumentos de inicialização do objeto e um *flag* booleano indicando se o objeto deve ser reiniciado com seu grupo) e para a reativação de objetos (incluindo como argumento um objeto `ActivationID`, previamente designado pelo serviço de ativação). Esses construtores deverão ser invocados nos construtores do objeto que implementam o serviço. Particularmente, o serviço de ativação busca um construtor com dois argumentos dos tipos `ActivationID` e `MarshaledObject`.

A classe `ActivationDesc` permite registrar a informação de ativação de um objeto sem criar uma instância deste objeto, usando para tanto o método estático `register()` da classe `Activatable`.

Antes de criar um serviço ativável, é preciso criar ou especificar o grupo de ativação ao qual ele pertence. Isto é realizado através dos métodos das classes `ActivationGroup`, `ActivationGroupID` e `ActivationGroupDesc`, todas do pacote `java.rmi.activation`. Um grupo define um conjunto de objetos ativáveis que devem compartilhar o mesmo espaço de endereçamento, executando na mesma máquina virtual.

O serviço de ativação é implementado em uma máquina virtual com um objeto da classe `Activator` do pacote `java.rmi.activation`. A aplicação `rimd`, distribuída juntamente com o pacote básico do JDK 1.2, é um *daemon* que implementa esse serviço.

Coleta de lixo distribuída

O processo de remoção de objetos remotamente não-referenciados ocorre de maneira automática. Cada servidor com objetos exportados mantém uma lista de referências remotas aos objetos que ele oferece. Através de comunicação com o cliente, ele é notificado quando a referência é liberada na aplicação remota.

Cada referência remota recebe também um período de validade; quando esse período expira, a referência é eliminada e o cliente é notificado. Esse mecanismo oferece uma alternativa para a liberação de objetos que tenham sido referenciados por clientes que eventualmente tenham falhado, ficando impedidos de sinalizar que a referência havia sido liberada.

Embora não sejam utilizadas normalmente por programadores, as funcionalidades do *distributed garbage collector* estão especificadas através das classes do pacote `java.rmi.dgc`.

Callback

Nas aplicações em uma arquitetura de objetos distribuídos, nem sempre a comunicação no estilo cliente-servidor é suficiente para atender aos requisitos da aplicação. É usual que o servidor RMI aja algumas vezes como cliente, invertendo os papéis com o cliente RMI original.

Considere o exemplo do *applet* cliente RMI. Nesse *applet*, não há como saber se outro cliente do mesmo objeto remoto realizou alguma atualização no valor do contador a não ser pressionando o botão *Get* e verificando se houve mudança. Essa é uma situação típica em muitas aplicações, sendo clara a necessidade de realizar tais notificações de forma automática.

O mecanismo para atingir esse objetivo é utilizar a estratégia de *callback*. Esta técnica é tipicamente utilizada quando a aplicação cliente requer um retorno do servidor mas não quer permanecer bloqueado aguardando a resposta. Através dessa técnica, o servidor obtém uma referência para o cliente de forma que pode invocar remotamente um método do objeto cliente. Assim, quando a execução do serviço solicitado é concluída, o servidor pode notificar o cliente através da invocação do método disponibilizado pelo cliente para uso remoto.

Basicamente, assim como para o objeto de serviço RMI, deve-se oferecer uma interface remota para o cliente a fim de permitir que o servidor tenha acesso ao “serviço” de atualização do cliente. Esse exemplo ilustra tal situação, com um método que será invocado pelo servidor quando seu valor for alterado, quando passará um valor inteiro para o cliente com a valor atualizado:

```
1 import java.rmi.*;
2 public interface CountClientInterface extends Remote {
3     void update(int val) throws RemoteException;
4 }
```

Observe que neste exemplo a interface remota do serviço também foi atualizada de forma a permitir o cadastro dos clientes interessados na atualização:

```
1 import java.rmi.*;
2 public interface Count extends Remote {
3     void set(int val) throws RemoteException;
4     void reset() throws RemoteException;
5     int get() throws RemoteException;
6     int increment() throws RemoteException;
7     void addClient(CountClientInterface c) throws RemoteException;
8     void remClient(CountClientInterface c) throws RemoteException;
9 }
```

Com *callback*, ambos cliente e servidor deverão implementar o serviço remoto especificado. Considere o código para o servidor:

```
1 import java.util.*;
2 import java.rmi.*;
3 import java.rmi.server.UnicastRemoteObject;
4 public class CountImpl extends UnicastRemoteObject
5     implements Count {
6     private int sum;
```

```
7    // Lista de clientes registrados
8    private Vector clientes = new Vector();
9    public CountImpl() throws RemoteException {
10   }
11   public void set(int val) throws RemoteException {
12       sum = val;
13   }
14   public void reset() throws RemoteException {
15       sum = 0;
16   }
17   public int get() throws RemoteException {
18       return sum;
19   }
20   public int increment() throws RemoteException {
21       sum++;
22       if (sum%100 == 0)
23           update();
24       return sum;
25   }
26   public void addClient(CountClientInterface client)
27       throws RemoteException {
28       clientes.add(client);
29   }
30   public void remClient(CountClientInterface client)
31       throws RemoteException {
32       clientes.remove(client);
33   }
34   public void update() throws RemoteException {
35       CountClientInterface cci;
36       for (int i=0; i<clientes.size(); ++i) {
37           cci = (CountClientInterface) clientes.elementAt(i);
38           cci.update(sum);
39       }
40   }
41 }
```

Similarmente, para o código do cliente:

```
1 import java.rmi.*;
2 import java.awt.*;
3 import java.awt.event.*;
4 import java.applet.*;
5 import java.rmi.server.*;
6 public class AppletClient extends Applet
7     implements ActionListener, CountClientInterface {
```



```
8      Count remCount;
9      TextField tfCnt;
10     Button bStart;
11     String bslabel = "Start";
12     public void init() {
13         try {
14             setLayout(new GridLayout(3,1));
15             add(new Label("Count:"));
16             tfCnt = new TextField(7);
17             tfCnt.setEditable(false);
18             add(tfCnt);
19             bStart = new Button(bslabel);
20             bStart.addActionListener(this);
21             add(bStart);
22             UnicastRemoteObject.exportObject(this);
23             showStatus("Binding remote object");
24             remCount = (Count) Naming.lookup("Count001");
25             showStatus("Registering with remote object");
26             remCount.addClient(this);
27             tfCnt.setText(Integer.toString(remCount.get()));
28         }
29         catch (Exception e) {
30             e.printStackTrace();
31         }
32     }
33     public void paint() {
34         try {
35             tfCnt.setText(Integer.toString(remCount.get()));
36         }
37         catch (Exception e) {
38             e.printStackTrace();
39         }
40     }
41     public void actionPerformed (ActionEvent ev) {
42         try {
43             showStatus("Incrementing...");
44             for (int i = 0 ; i < 1000 ; i++ )
45                 remCount.increment();
46             showStatus("Done");
47         }
48         catch (Exception e) {
49             e.printStackTrace();
50         }
51     }
```

```
52     public void update(int val) throws RemoteException {
53         showStatus("Update");
54         tfCnt.setText(Integer.toString(val));
55     }
56 }
```

Como anteriormente, devem ser criados os stubs e skeletons para ambos os serviços. O código do servidor não sofre alteração em relação ao exemplo anterior, assim como a forma de execução da aplicação.

5.4.3 Java IDL

A API Java IDL, presente na plataforma Java desde a versão 1.2, permite a integração entre objetos Java e outros objetos, eventualmente desenvolvidos em outras linguagens de programação, através da arquitetura CORBA. Os principais pacotes que compõem essa API são `org.omg.CORBA` e `org.omg.CosNaming`.

A partir da versão 1.3 da plataforma Java, é possível gerar interfaces IDL para classes Java usando o compilador `rmic` com a opção `-idl`. Outra opção, `-iiop`, indica que o protocolo de comunicação de CORBA, IIOP, será utilizado em *stubs* e *ties* (correspondentes aos *skeletons*) de RMI.

Arquitetura CORBA

CORBA (*Common Object Request Broker Architecture*) é um padrão definido pelo consórcio OMG (*Object Management Group*) que define uma arquitetura de objetos, com uma linguagem para descrição de interfaces com mapeamentos padronizados para diversas linguagens e um conjunto de serviços básicos.

Como o padrão CORBA visa atender a diversas linguagens de programação, sua especificação é ampla e relativamente complexa. De forma extremamente simplificada, os componentes básicos dessa arquitetura são:

- a linguagem de descrição de interfaces;
- o intermediário para repassar requisições a objetos remotos;
- o serviço para localizar objetos remotos; e
- o protocolo de comunicação.

IDL é a *Interface Description Language*, uma linguagem que permite especificar interfaces de forma independente da linguagem de programação na qual a especificação é implementada. CORBA determina uma série de mapeamentos padronizados entre IDL e outras linguagens, tais como C, C++, COBOL e Java.

ORB é o *Object Request Broker*, o núcleo da arquitetura CORBA. É um programa que deve estar executando em cada máquina envolvida em uma aplicação CORBA, sendo o responsável pela conexão entre clientes e serviços através dos correspondentes *stubs* e *skeletons*.

O Serviço de Nomes de CORBA define uma estrutura para associar nomes a objetos remotos definidos na arquitetura. A estrutura definida é uma hierarquia (ou árvore), onde cada ramo define um contexto distinto e cujas folhas são os nomes dos serviços disponibilizados. Assim, a referência completa para o nome de um serviço é dada pelo contexto (os nomes dos nós intermediários) e pelo nome do serviço.

O protocolo de comunicação de CORBA especifica o padrão para que as requisições de objetos transmitidas entre ORBs, independentemente de como ou em qual linguagem esses ORBs foram implementados, possam ser reconhecidas. O protocolo de comunicação CORBA mais comum é o IIOP, o *Internet Inter-ORB Protocol*, em função da disseminação da Internet, mas outros protocolos podem ser obtidos para outras plataformas.

CORBA e Java

Uma vez definida ou obtida a interface IDL para um serviço, as classes auxiliares para acessar o objeto remoto que implementa o serviço são obtidas pela compilação da interface, usando o aplicativo `idlj` (ou `idltojava` ou ainda `idl2java` em versões anteriores à Java 1.3). Além de classes para *stubs* e *skeletons*, são geradas classes auxiliares (*helpers* e *holders*) para permitir a comunicação entre objetos Java e dados estabelecidos em outras linguagens.

Na plataforma Java há uma implementação para o serviço de nomes de CORBA, oferecida pelo aplicativo `tnameserv`. Esse serviço está mapeado por default para a porta 900, podendo esta ser modificada pela opção `-ORBInitialPort`.

A interação entre um ORB e um programa Java dá-se através de métodos da classe ORB. Para inicializar a referência ao ORB, utiliza-se o método estático `init()` dessa classe. Para obter uma referência para o serviço de nomes utiliza-se o método `resolve_initial_references()`, tendo a `NameService` como argumento.

O exemplo a seguir é a implementação usando CORBA do clássico programa “Hello, world”. É composto por três arquivos: a interface IDL, o cliente e o servidor.

A Interface IDL descreve um serviço com um único método, sendo aqui definida usando as construções da linguagem IDL:

```
1 module HelloApp {
2     interface Hello {
3         string sayHello();
4     }
5 }
```

Usando-se o aplicativo `idlj`, gera-se a interface Java correspondente, com a tradução das construções IDL para as primitivas Java segundo o padrão estabelecido em CORBA, além de outros arquivos auxiliares (*stub*, *skeleton*, *helper*, *holder*), não apresentados aqui:

```
1 package HelloApp;
2 public interface Hello
3     extends org.omg.CORBA.Object {
4     String sayHello();
5 }
```

O código cliente ativa o ORB, obtém uma referência para o serviço de nomes e, a partir deste serviço, obtém uma referência remota para o objeto com o serviço *Hello*. Obtida a referência, o método é invocado normalmente:

```
1  import HelloApp.*;
2  import org.omg.CosNaming.*;
3  import org.omg.CORBA.*;
4  public class HelloClient {
5      public static void main (String args[]) {
6          try {
7              ORB meuOrb = ORB.init(args,null);
8              org.omg.CORBA.Object objRef =
9                  meuOrb.resolve_initial_references("NameService");
10             NamingContext ncRef = NamingContextHelper.narrow(objRef);
11             NameComponent nc = new NameComponent("Hello","");
12             NameComponent path[] = {nc};
13             Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
14             String hi = helloRef.sayHello();
15             System.out.println(hi);
16         }
17         catch(Exception e) {
18             System.out.println(e);
19             e.printStackTrace(System.out);
20         }
21     }
22 }
```

Nesse exemplo, combina-se a implementação do serviço e o correspondente servidor. A classe *HelloServer* é um servidor que ativa o ORB, cria o objeto que implementa o serviço, obtém uma referência para o serviço de nomes e registra o objeto neste diretório associado ao nome *Hello*. A classe *HelloServant* é uma implementação do serviço especificado; observe que essa classe é uma extensão de *_HelloImplBase*, o *skeleton* definido pelo aplicativo *idl j*:

```
1  import HelloApp.*;
2  import org.omg.CosNaming.*;
3  import org.omg.CosNaming.NamingContextPackage.*;
4  import org.omg.CORBA.*;
5  public class HelloServer {
6      public static void main(string args[]) {
7          try {
8              // Create the ORB
9              ORB orb = ORB.init(args,null);
10             // Instantiate the servant object
11             HelloServant helloRef = new HelloServant();
12             // Connect servant to the ORB
13             orb.connect(helloRef);
```

```
14         //Registering the servant
15         org.omg.CORBA.Object objRef =
16             orb.resolve_initial_references("NameService");
17         NamingContext ncRef = NamingContextHelper.narrow(objRef);
18         NameComponent nc = new NameComponent("Hello", "");
19         NameComponent path[] = {nc};
20         ncRef.rebind(path, helloRef);
21         // Wait for invocation
22         java.lang.Object sync = new java.Lang.Object();
23         synchronized(sync) {
24             sync.wait();
25         }
26     }
27     catch(Exception e) {
28         System.out.println(e);
29         e.printStackTrace(System.out);
30     }
31 }
32 }
33 class HelloServant extends _HelloImplBase {
34     public String sayHello() {
35         return "\nHelloWorld!\n";
36     }
37 }
```

Apêndice A

Palavras chaves de Java

As palavras a seguir são de uso reservado em Java e não podem ser utilizadas como nomes de identificadores:

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