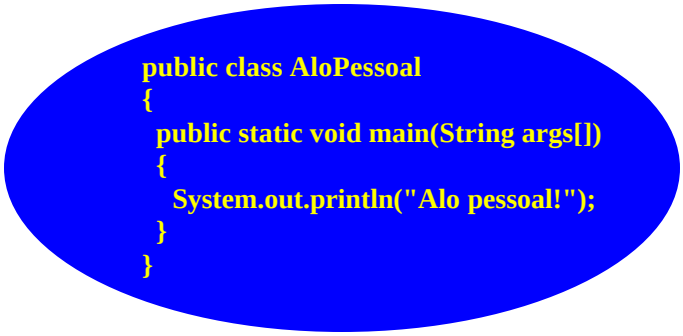
Fundação Universidade Regional de Blumenau
Centro de Ciências Exatas e Naturais
Departamento de Sistemas e Computação



Apostila da Linguagem de Programação Java

Prof. Francisco Adell Péricas
Prof. Marcel Hugo
{pericas,marcel}@furb.br

Fevereiro/2002

1 Índice

2 Introdução

Tendo sido originalmente concebida para o desenvolvimento de pequenos aplicativos e programas de controle de aparelhos eletrodomésticos e eletroeletrônicos, a linguagem de programação Java mostrou-se ideal para ser usada na rede Internet. O que a torna tão atraente é o fato de programas escritos em Java poderem ser executados virtualmente em qualquer plataforma, mas principalmente em Windows, Unix e Mac. Soma-se a isso o fato de programas Java poderem ser embutidos em documentos HTML, podendo assim ser divulgados pela rede. Diferente da linguagem C, não é apenas o código fonte que pode ser compartilhado pela rede, mas o próprio código executável compilado, chamado *bytecode*.

Java foi desenvolvida por um grupo de pesquisadores da SUN Microsystems por volta de 1990, pouco antes da explosão da Internet. Essa linguagem possui estrutura muito semelhante à da linguagem C, da qual descende imediatamente. O Java tem em comum com a linguagem C++ o fato de ser orientada a objetos e mantém com esta um alto grau de semelhança. O paradigma de programação orientada a objetos consiste de um grau a mais na abstração da programação, em comparação com a programação estruturada, e tem se mostrado extremamente útil na produção de programas cada vez mais sofisticados, em menor tempo e com maior qualidade. A programação orientada a objetos (POO) é hoje universalmente adotada como padrão de mercado.

Há uma certa curiosidade por detrás do nome dado a essa linguagem de programação. Java é o nome de uma ilha do Pacífico, onde se produz uma certa variedade de café homônimo. A inspiração bateu à equipe de desenvolvimento ao saborear esse café em uma lanchonete local. Deram-se conta de como era extremamente apreciado por profissionais da área de software (ao menos nos Estados Unidos), de modo que não foi menos justo fazer-lhe homenagem ao batizar uma nova linguagem de programação.

Atualmente, o site JavaSoft mantém informações atualizadas sobre o desenvolvimento da linguagem Java e suas relações com o mercado, assim como utilitários e ferramentas disponíveis para serem baixados gratuitamente.

2.1 Execução de Programas Java

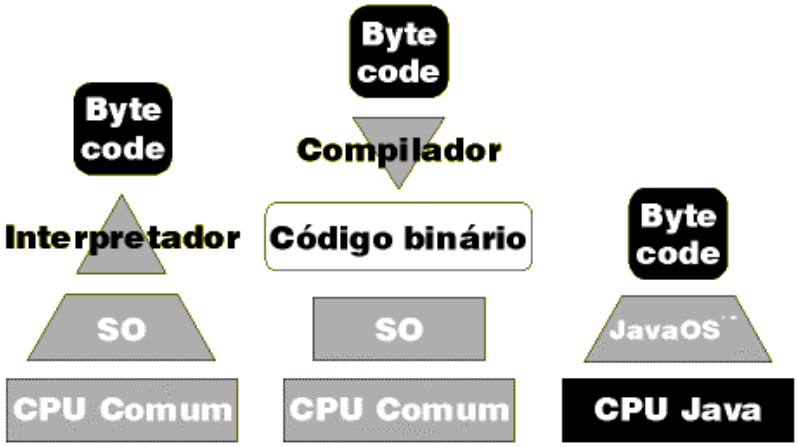
Um programa fonte escrito em Java é traduzido pelo compilador para os *bytecodes*, isto é, o código de máquina de um processador virtual, chamado *Java Virtual Machine (JVM).* A JVM é um programa capaz de interpretar os bytecodes produzidos pelo compilador, executando o programa cerca de 20 vezes mais lento do que C. Pode parecer ruim, mas é perfeitamente adequado para a maioria das aplicações. Com isto, um programa Java pode ser executado em qualquer plataforma, desde que esteja dotada de uma JVM. É o caso dos programas navegadores mais populares, como o Netscape Navigator e o Internet Explorer, que já vêm com uma JVM. A vantagem desta técnica é evidente: garantir uma maior portabilidade para os programas Java em código-fonte e compilados. Porém, as JVM tendem a ser

programas extensos que consomem muitos recursos, restringindo assim o tamanho das aplicações escritas em Java.

Atualmente, já existem compiladores capazes de traduzir bytecodes para instruções de máquina nativas, como o *Just In Time compiler* (ou JIT), tornando os programas ainda mais rápidos. Este compilador requer uma versão específica para cada plataforma onde se pretende que o programa Java seja executado. Em contrapartida à maior velocidade de execução está também uma maior necessidade de memória, pois os bytecodes compilados, em geral, ficam três vezes maiores do que o original. Uma alternativa bem mais interessante, e talvez muito mais viável, é a implementação da JVM em hardware na forma de uma placa ou microchip.

A primeira iniciativa neste sentido é da Sun Microelectronics, que está produzindo os chips picoJava™, microJava™ e UltraJava™. Estes são capazes executar diretamente bytecodes, acelerando em milhares de vezes a velocidade de execução. Isto permitirá o desenvolvimento viável de aplicativos cada vez mais complexos, abrangentes e funcionais.

Espera-se que estas soluções sejam brevemente empregadas na fabricação de telefones celulares, Pager, jogos, organizadores pessoais digitais, impressoras e eletrodomésticos de consumo, além aplicações mais sérias como estações de trabalho dotadas do sistema operacional JavaOS™. Trata-se certamente do futuro das soluções para aplicações de rede.



3 Programação Básica

3.1 Primeiros Exemplos

Já se tornou clássica a idéia de que para aprender uma nova linguagem de programação não se deve ir direto à sua descrição formal. Ao invés disso, é melhor examinar cuidadosamente um pequeno programa escrito nessa linguagem, o mais simples possível, mas que permita "quebrar o gelo". Ao compreender as diversas partes componentes do exemplo, já teremos dado um grande passo para podermos escrever qualquer programa.

Seguindo essa linha, apresentamos nosso primeiro programa, o clássico "Alô pessoal!". O objetivo deste programa é simplesmente escrever na tela a frase "Alô pessoal!". Vejamos como é o código fonte:

```
public class AloPessoal
{
  public static void main(String args[])
  {
    System.out.println("Alo pessoal!");
  }
}
```

3.1.1 Digitando o Programa

Para digitar este programa, é recomendável utilizar um editor de texto simples como o Notepad ou Bloco de Notas do Windows. O nome do arquivo deve ser exatamente igual ao nome que aparece após a palavra **class** na primeira linha do programa e dever ter **.java** como sufixo. Assim sendo, o nome deverá ser "AloPessoal.java". Cuidado para digitar corretamente as maiúsculas e minúsculas, pois a linguagem Java é sensível ao tipo de caixa das letras (*case sensitive*).

3.1.2 Compilando o Código Fonte

Para criar o código binário, chamamos o compilador Java através da linha de comando, do seguinte modo:

```
javac AloPessoal.java
```

Com isso, será criado um arquivo binário (desde que tudo corra bem) com o mesmo nome do arquivo original, mas com sufixo **.class** no lugar de **.java**. No nosso caso, teríamos um arquivo **AloPessoal.class**. Entre as coisas que poderiam dar errado nesse momento, o código fonte pode conter erros. Esse não deverá ser o caso, se tiver digitado o programa exatamente como aparece acima. Se, porém, o compilador emitir mensagens de erro, será preciso identificar as linhas que estão incorretas, corrigí-las no editor, e chamar o compilador novamente. As linhas que contém algum tipo de incorreção são listadas pelo compilador juntamente com seu número, facilitando sua localização no editor.

3.1.3 Executando o Código

Para podermos executar o programa é necessário chamar o interpretador Java, pois, como vimos, os *bytecodes* foram feitos para rodar em uma *Java Virtual Machine*. Podemos fazê-lo do seguinte modo:

```
java [nome da classe]
```

O campo [nome da classe] é o nome do arquivo sem o sufixo **.class**. Em nosso caso, este será **AloPessoal**. Ao executar o programa, ele deverá exibir na tela a frase:

```
Alo pessoal!
```

Este é um programa muito simples. Porém, a boa compreensão das estruturas presentes nele deverá permitir a programação fluente de Java em pouco tempo.

3.1.4 Entendendo a Estrutura do Programa

Todo programa Java, deve conter ao menos uma declaração da forma:

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

O campo [nome] é o nome da classe e a parte "..." é um código Java válido a ser executado no programa. O nome de uma classe é um identificador, como qualquer outro presente no programa, por isso não deve conter espaços ou outros caracteres gráficos, isto é, deve ser um nome composto de uma seqüência de caracteres que seja válida para um identificador. Outros exemplos de identificadores são nomes de variáveis, nomes de comandos, etc.

Vamos adiar um pouco a complicação sobre o que vem a ser uma classe, pois isso depende de alguns conceitos da programação orientada a objetos. Por hora, vamos apenas aceitar que todo programa Java deve conter ao menos uma classe e que é dentro de uma classe que vão os dados e os procedimentos. Notemos ainda que todo programa Java (mas não as applets) deve ter uma classe dotada de um procedimento chamado **main**. Os procedimentos em Java são chamados métodos. Os métodos encerram um conjunto de declarações de dados e de comandos que são executados mediante a chamada do método por seu nome. Vamos estudar os métodos em detalhe mais adiante. O método **main** é o ponto onde se dá o início da execução do programa, isto é, um método chamado automaticamente pela JVM.

Voltando ao nosso programa **AloPessoal**, o código a ser executado é

```
System.out.println("Alo pessoal!");
```

System.out.println é o nome de uma função que serve para escrever informações textuais na tela. Os dados a serem escritos, devem estar delimitados entre parênteses. *"Alo pessoal!"* é uma frase. Em computação, uma palavra ou uma frase que tenha função literal é denominada *string.* Em Java, a representação de uma *string* constante se dá colocando os caracteres entre aspas, por exemplo: "Imagem", "Rio de Janeiro", "Brasília", etc.

Note que existe um **;** (ponto e vírgula) no final da linha, logo após o ")". Em Java é obrigatório colocar um ponto e vírgula após cada comando. Isso porque um comando pode ser quebrado em múltiplas linhas, sendo necessário sinalizar de algum modo onde é que o comando termina.

Os espaços em branco são permitidos entre os elementos do código-fonte, sem qualquer restrição. Os espaços em branco incluem espaços, tabulações e novas linhas. São usados para melhorar a aparência visual e a legibilidade do código-fonte.

O Java é sensível ao tipo de caixa, isto é, distingue caixa alta (maiúsculo) da caixa baixa (minúsculo).

3.1.5 Mais Exemplos

Podemos escrever "Alo pessoal!" escrevendo primeiro, "Alo " e depois, "pessoal!". Para isso, o programa deve ser alterado da seguinte forma:

```
public class Alo
{
  public static void main(String args[])
  {
    System.out.print("Alo ");
    System.out.println("pessoal!");
  }
}
```

Para escrever dados genéricos na tela, existe o comando **System.out.print** que escreve o dado e mantém a mesma linha. Há também o **System.out.println** que escreve dados e muda para a próxima linha.

Podemos concatenar dois ou mais strings usando o operador "+". Por exemplo:

```
"Alo " + "pessoal!"
```

É o mesmo que "Alo pessoal!". Para escrever um número, basta escrever **[string]+n** onde **[string]** é uma *string* qualquer e **n** é um número. Por exemplo:

```
public class Numero
{
  public static void main(String args[])
  {
    System.out.println("O valor é " + 29);
  }
}
```

Como o lado esquerdo da expressão é uma *string*, 29 é convertido para a *string* "29" e é concatenado com "O valor é ". Compilando e executando esse exemplo como fizemos anteriormente, devemos obter na tela:

```
O valor é 29
```

Observemos que os comandos **System.out.print** e **System.out.println** escrevem uma informação de cada vez. Desta forma, precisamos usar "+" para unir "O valor e " com 29 para formar uma única string. No exemplo acima, nada impede que os dados sejam escritos em duas etapas: primeiro, "O valor e " e depois, 29. Neste caso, teríamos:

```
public class Numero
{
  public static void main(String args[])
  {
    System.out.print("O valor e ");
    System.out.println(29);
  }
}
```

Entretanto, a primeira abordagem parece mais razoável pois torna a programação mais clara e concisa, além de economizar comandos.

3.1.6 Introduzindo Comentários no Código

Um comentário é uma porção de texto que não tem função para o compilador Java, mas é útil ao leitor humano. Assim sendo, um comentário é identificado mas ignorado completamente pelo compilador Java. A utilidade dos comentários é óbvia: deve conter explicações sobre um particular desenvolvimento do código, permitindo ao leitor compreender claramente o que se deseja realizar.

Os comentários são introduzidos no código Java de três formas distintas:

- Única linha: Escrevendo "//" antes do comentário, que se estenderá até o final da linha;
- Várias linhas: Colocado em qualquer parte do programa e delimitado entre /* e */ ;
- De documentação: Colocado em qualquer parte do programa e delimitado entre /** e */ . Este tipo Indica que o comentário deve ser inserido em qualquer documentação gerada automaticamente, por exemplo pelo programa javadoc .

Por exemplo, o código:

```
//
// Este é um exemplo de como somar dois numeros
//
public class Numero
{
  public static void main(String args[]) /* Método principal */
  {
    double x, y; // estes sao numeros reais de dupla precisao
    // System.out.print("x = 2.0"); /* inicializando o "x" */
    x = 2;
    y = 3.0; /* iniciando o y, e fazendo y = y + x; */ y = y + x;
    // escrevendo a soma
    System.out.println("x + y = " + (x + y));
  }
} /* fim de Numero */
```

É equivalente ao código:

```
public class Numero
{
  public static void main(String args[])
  {
    double x, y;
    x = 2;
    y = 3.0;
    y = y + x;
    System.out.println("x + y = " + (x + y));
  }
}
```

3.1.7 Usando Variáveis

Uma variável é simplesmente um espaço vago, reservado e rotulado para armazenar dados. Toda variável tem um nome que a identifica univocamente e um valor, que corresponde àinformação a ela atribuída. Por exemplo:

```
int n;
```

Especifica que **n** é o nome de uma variável que pode armazenar um número inteiro como valor. Em geral, num contexto onde aparece o nome de uma variável ocorre a

substituição por seu valor. O valor de uma variável pode mudar muitas vezes durante a execução de um programa por meio de atribuições de valor.

Há diversos tipos de variáveis em Java, correspondendo aos vários tipos de dados aceitos. Vamos fazer um pequeno programa que declara uma variável inteira, atribui a ela uma constante, e imprime seu valor na tela:

```
public class Numero
{
  public static void main(String args[])
  {
    int n;
    n = 17 + 21;
    System.out.println("O valor numérico é " + n);
  }
}
```

O local onde uma variável está declarada é extremamente importante. Uma variável é conhecida apenas dentro de algum escopo. Por exemplo, uma variável declarada no escopo de uma classe (fora de um método) é conhecida por qualquer método que esteja declarado dentro dessa mesma classe, enquanto uma variável declarada no escopo de um procedimento é conhecida apenas por esse procedimento. Há ainda outros tipos de escopo, como veremos mais adiante.

O sinal "=" é um operador, utilizado para atribuir um valor a uma variável. Por exemplo, **n = 1;** faz com que o valor **1** seja armazenado na variável **n**. Há também os operadores usuais de adição, subtração, multiplicação e divisão de números. Estes são representados pelos símbolos "**+**", "**-**", "*****" e "**/**", respectivamente.

Ao executar o programa acima (claro, depois de compilá-lo), ele escreve:

```
O valor numérico é 38
```

3.1.8 Identificadores

Um nome de variável, assim como nome de um método, classe, rótulo e dezenas de outros itens lexicográficos, constitui o que é chamado um identificador. Uma vez criado, um identificador representa sempre o mesmo objeto a ele associado, em qualquer contexto em que seja empregado.

As seguintes regras regem a criação de identificadores:

- O primeiro caractere de um identificador deve ser uma letra. Os demais caracteres podem ser quaisquer seqüências de numerais e letras;
- Não apenas os numerais e letras latinas podem ser empregadas, como também letras de quaisquer outro alfabeto;
- Devido a razões históricas, o underscore "_" e o sinal de dólar "$" são considerados letras e podem ser usados nos identificadores;
- Os identificadores distinguem o tipo de caixa das letras, isto é, as maiúsculas são consideradas distintas das minúsculas;
- Os identificadores não podem ser palavras reservadas, como: **class**, **for**, **while**, **public**, etc.

Há um padrão de codificação definido pela Sun e seguido pela maioria dos programadores Java, que determina que:

- A primeira letra do nome de uma classe é maiúscula. Se este nome é composto por várias palavras, elas são escritas juntas (sem usar algum separador) e a primeira letra de cada palavra é maiúscula. Exemplo: class AloPessoal;
- Para o restante: métodos, atributos e referências a objetos, o estilo é similar ao da classe, porém sendo a primeira letra do identificador minúscula. Exemplo: boolean luzAcesa; int qtdeParafusos .

3.2 Tipos de Dados

O trabalho com computadores, desde os mais simples como escrever mensagens na tela, até os mais complexos como resolver equações ou desenhar imagens tridimensionais em animação, consiste essencialmente em manipulação de dados. Os dados representados em um computador podem ser números, caracteres ou simples valores.

A linguagem Java oferece alguns tipos de dados com os quais podemos trabalhar. Na verdade há basicamente duas categorias em que se encaixam os tipos de dados: tipos primitivos e tipos de referências. Os tipos primitivos correspondem a dados mais simples ou escalares e serão abordados em detalhe no que segue, enquanto os tipos de referências consistem em *arrays*, classes e interfaces. Estes serão vistos em seções subseqüentes.

Eis uma visão geral dos tipos que serão abordados nesta seção:

Tipo	Descrição
boolean	Tipo lógico que pode assumir o valor true ou o valor false.
char	Caractere em notação Unicode de 16 bits. Serve para a armazenagem de dados alfanuméricos.
byte	Inteiro de 8 bits em notação de complemento de dois. Variáveis deste tipo podem assumir valores entre -2^7=-128 e 2^7-1=127.
short	Inteiro de 16 bits em notação de complemento de dois. Os valores possíveis cobrem a faixa de -2^{15}=-32.768 a 2^{15}-1=32.767.
int	Inteiro de 32 bits em notação de complemento de dois. Pode assumir valores entre -2^{31}=-2.147.483.648 e 2^{31}-1=2.147.483.647.
long	Inteiro de 64 bits em notação de complemento de dois. Pode assumir valores entre -2^{63} e 2^{63}-1.
float	Representa números em notação de ponto flutuante normalizada em precisão simples de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo que pode ser representado por esse tipo é 1.40239846e-46 e o maior é 3.40282347e+38.
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo que pode ser representado é 4.94065645841246544e-324 e o maior valor positivo é 1.7976931348623157e+308.

Ao contrário do que acontece com outras linguagens de programação, as características dos tipos de dados listados acima independem da plataforma em que

o programa será executado. Dessa forma, os tipos de dados primitivos são realmente únicos e garantem a capacidade de intercâmbio de informações entre diversos tipos de computadores.

3.2.1 Tipo de Dado Lógico

O tipo **boolean** é o tipo de dado mais simples encontrado em Java. Uma variável lógica pode assumir apenas um entre dois valores: *true* ou *false*. Algumas operações possíveis em Java como **a<=b**, **x>y**, etc., têm como resultado um valor lógico, que pode ser armazenado para uso futuro em variáveis lógicas. Estas operações são chamadas operações lógicas. As variáveis lógicas são tipicamente empregadas para sinalizar alguma condição ou a ocorrência de algum evento em um programa Java. Por exemplo:

```
boolean fim_do_arquivo = false;
```

É a declaração de uma variável do tipo **boolean**, cujo nome é **fim_do_arquivo**. O valor **false** à direita do sinal "=" indica que a variável recebe esse valor como valor inicial. Sem essa especificação o valor de uma variável é imprevisível, podendo ser qualquer um dos valores possíveis para seu tipo (neste caso *true* ou *false*).

Aproveitando o ensejo, há nessa linha a essência da declaração de qualquer variável em Java:

1. Informar o tipo de dado que deseja declarar (**boolean**);
2. Informar o nome que será usado para batizar a variável (**fim_do_arquivo**);
3. Atribuir à variável um valor inicial (**= false**);
4. Terminar a declaração com um ponto-e-vírgula "**;**".

3.2.1.1 Operações com booleanos

Podemos realizar uma série de operações com os dados do tipo *boolean*. A tabela seguinte mostra uma lista completa.

!	Operador lógico de negação
==, !=	Operadores de igualdade e diferença
&&, ||	Operadores lógicos E e OU .
&=, |=, ^=	Operadores de atribuição com operação lógica E, OU e OU-exclusivo

3.2.2 Tipos de Dados Inteiros

Os tipos de dados primitivos **byte**, **int**, **short** e **long** constituem tipos de dados inteiros. Isso porque variáveis desses tipos podem conter um valor numérico inteiro dentro da faixa estabelecida para cada tipo individual.

Há diversas razões para se utilizar um ou outro dos tipos inteiros em uma aplicação. Em geral, não é sensato declarar todas as variáveis inteiras do programa como **long**. Raramente os programas necessitam trabalhar com dados inteiros que

permitam fazer uso da máxima capacidade de armazenagem de um **long**. Além disso, variáveis grandes consomem mais memória do que variáveis menores, como short.

Se alguma operação aritmética cria um resultado que excede um dos limites estabelecidos para o tipo inteiro empregado, não há qualquer indicação de erro para avisar sobre essa ocorrência. Ao invés disso, um complemento de dois do valor obtido será o resultado. Por exemplo, se a variável for do tipo **byte**, ocorrem os seguintes resultados: 127+1 = -128, 127+9=-120 e 127+127=-2. Entretanto, uma exceção do tipo **ArithmeticException** é levantada caso ocorra uma divisão por zero. As exceções e seus mecanismos serão abordados mais tarde. Vejamos o seguinte código:

```
public class Arith
{
  public static void main(String args[])
  {
    byte a = 127;
    short b = 32767;
    int c = 2147483647;
    long d = 9223372036854775807L;
    int e = 0;
    a += 1;
    b += 1;
    c += 1;
    d += 1;
    System.out.println("Valor de a = " + a);
    System.out.println("Valor de b = " + b);
    System.out.println("Valor de c = " + c);
    System.out.println("Valor de d = " + d);
    d /= e;   // Vai dar erro porque e = 0
  }
}
```

O seu respectivo resultado de execução é:

```
C:\>java Arith
Valor de a = -128
Valor de b = -32768
Valor de c = -2147483648
Valor de d = -9223372036854775808
java.lang.ArithmeticException: / by zero
        at Arith.main(Arith.java:18)

C:\>
```

Seguem abaixo alguns exemplos de declarações de variáveis de tipo inteiro:

```
byte contador = 1;
int anguloEmGraus = -45;
short indice = 6;
```

A diferença entre essas declarações e a declaração de dados lógicos vista acima está no tipo de dado especificado e no valor atribuído a cada variável.

3.2.2.1 Operações com inteiros

Podemos realizar uma série de operações com os dados do tipo inteiro. A tabela seguinte mostra uma lista completa.

Operação	Descrição
=	Operador de atribuição
==, !=	Operadores de igualdade e diferença
<, <=, >, >=	Operadores de desigualdade (relacionais)
+, -	Operadores unários
+, -, *, /, %	Adição, subtração, multiplicação, divisão e módulo
+=, -=, *=, /=, %=	Operadores de atribuição com adição, subtração, multiplicação, divisão e módulo
++, --	Incremento e decremento
<<, >>, >>>	Operadores de deslocamento de bits
<<=, >>=, >>>=	Operadores de atribuição com deslocamento de bits

Muitas das operações que aparecem na lista acima são familiares e praticamente não requerem explicação. Há outros, porém, que pode ser um tanto quanto ambíguos. É o caso dos operadores de atribuição aritméticos. Estes consistem de atalhos para atribuir um novo valor a uma variável onde esse novo valor depende do valor anterior lá armazenado. Por exemplo: += adiciona um valor ao valor antigo de uma variável e a soma passa a ser o novo valor. Esse padrão também é obedecido para as operações -=, *=, /= e %=. Temos assim as seguintes correspondências:

x += 5	**x = x + 5**
x -= y	**x = x - y**
x *= 2	**x = x * 2**
z /= 4	**z = z / 4**

Os operadores de incremento e decremento referem-se a apenas uma variável (logo são chamados de unários). Por exemplo, o operador de incremento soma um ao operando conforme o exemplo:

```
x++;
```

É a maneira muito mais concisa de se escrever x = x + 1. Esses operadores se comportam de modo diferente quando seguem ou precedem o nome de uma variável. Se o operador precede o nome da variável, então o incremento (ou decremento) ocorre antes que o valor da variável seja tomado para a expressão aritmética. No seguinte exemplo, o valor das variáveis x e y será 6:

```
int x = 5;
int y = ++x;
```

Porém, no próximo exemplo o valor de x será 6 enquanto que o valor de y será 5:

```
int x = 5;
int y = x++;
```

Vejamos alguns exemplos de declarações e de utilizações de operações envolvendo tipos inteiros:

```
byte j = 60;
short k = 24;
int l = 30;
long m = 12L;
long resultado = 0L;
```

```
resultado += j;         // resultado = 60 (0 mais 60)
resultado += k;         // resultado = 84 (60 mais 24)
resultado /= m;         // resultado = 7  (84 dividido por 12)
resultado -= l;         // resultado = -23(7 menos 30)
resultado = -resultado; // resultado = 23 ( -(-23) )
resultado %= m;         // resultado = 11 (resto de 23 div. 12)
```

3.2.3 Tipo de Dado Caractere

Uma variável do tipo **char** armazena um caractere Unicode. Um caractere Unicode é um caractere de 16 bits, sendo que de 0 a 225 correspondem aos caracteres do código ASCII (a tabela ASCII é uma tabela padronizada internacionalmente de associações entre caractere e a sua representação numérica no computador). Uma constante do tipo caractere é representada colocando-se entre apóstrofos, ou pelo valor numérico correspondente na tabela Unicode (ou ASCII), ou ainda, pela seqüência '\x' onde x especifica o caractere a ser referido. Por exemplo: 'a', 'f', '\n', etc, são constantes do tipo **char**.

3.2.4 Tipos de Dados Ponto Flutuante

Em Java existem duas categorias de variáveis de ponto flutuante*:* **float** armazena valores numéricos em ponto flutuante de precisão simples e **double** de precisão dupla. Ambas seguem norma: IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE, New York). O fato de obedecer a essa norma é que torna os tipos de dados aceitos pela linguagem Java tão portáveis. Esses dados serão aceitos por qualquer plataforma, independendo do tipo de sistema operacional e do fabricante do computador. A representação dos valores em ponto flutuante pode ser feita usando a notação decimal (exemplo: -24.321) ou a notação científica (exemplo: 2.52E-31).

Além dos possíveis valores numéricos que uma variável de ponto flutuante pode assumir há também os seguintes:

- menos infinito;
- mais infinito ;
- zero;
- NAN - not a number.

Estes são requeridos pela norma. O mais infinito é usado, por exemplo, ao somarmos 1 ao maior valor possivelmente representável por esse sistema.
Muitas das operações realizáveis com inteiros (porém não todas) têm análogas para números de ponto flutuante. Eis um resumo:

Operação	Descrição
=	Operador de atribuição
==, !=	Operadores de igualdade e diferença
<, <=, >, >=	Operadores de desigualdade
+, -	Sinais unários
+, -, *, /	Adição, subtração, multiplicação e divisão
+=, -=, *=, /=	Operadores de atribuição com adição, subtração, multiplicação e divisão
++, --	Operadores unários de incremento e decremento

3.2.5 Literais

Um literal é a forma como deve ser escrito um valor de tipo de dado no programa.

Os literais inteiros (*int*) podem ser escritos em:

- Decimais: 1 , 2, 45 ,...
- Octais: 07, 010 (a presença do 0 no início)
- Hexadecimais: 0xff (=255, a presença de 0x no início)

Os literais de ponto flutuante (*double*) podem ser escritos em:

- 2.0 , 3.14159 , ... (notação decimal)
- 314159e-05 , 314159E-05 (notação científica)
- 0.0f (f indica que o literal é do tipo *float*)

Os literais booleanos (*boolean*) são:

- *true* e *false*

Os literais de caracteres (*char*) podem ser escritos como:

- Valores de 16 bits que podem ser manipulados como inteiros
- Par de apóstrofos (‘ ‘) ‘a’
- Seqüência de escape (\ ) ‘ \141’ octal = ‘ \u0061’ hex = ‘a’

3.3 Expressões

Expressões são combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenar *strings*, comparar valores, realizar operações lógicas e manipular objetos. O resultado da avaliação de uma expressão é em geral um valor compatível com os tipos dos dados que foram operados.

A maneira como uma expressão é avaliada (ou calculada) obedece as mesmas regras familiares da Matemática. A regra mais simples é a da associatividade. Quando um mesmo operador aparece mais de uma vez em uma expressão, como em **a+b+c**, então o operador mais à esquerda é avaliado primeiro, em seguida o da direita, e assim por diante. Esta seria então equivalente a **((a+b)+c)**.

3.3.1 Precedência

Há situações em que é necessário juntar operadores diferentes numa mesma expressão. Nesse caso a associatividade não se aplica mais trivialmente. A precedência é um conjunto de regras que permitem determinar quais são os operandos de um dado operador.

Como a linguagem Java é rica em operadores, alguns pouco familiares, precisamos conhecer a precedência desses operadores.

Na tabela a seguir, está indicada a precedência dos operadores comumente usados em Java. A tabela apresenta os operadores ordenados da maior precedência para a menor. Observe que existem alguns operadores que naturalmente não requerem preocupação quanto à sua precedência, devido à forma como são empregados.

Operador	Descrição
. [] () (*tipo*)	Máxima precedência: separador, indexação, parâmetros, conversão de tipo
+ - ~ ! ++ --	Operadores unários: positivo, negativo, negação (inversão bit a bit), não (lógico), incremento, decremento
* / %	Multiplicação, divisão e módulo (inteiros)
+ -	Adição, subtração
<< >> >>>	Translação (bit a bit) àesquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0)
< <= >= <	Operador relacional: menor, menor ou igual, maior ou igual, maior
== !=	Igualdade: igual, diferente
&	Operador lógico **e** bit a bit
^	Operador lógico **ou exclusivo** (**xor**) bit a bit
|	Operador lógico **ou** bit a bit
&&	Operador lógico **e** condicional
||	Operador lógico **ou** condicional
?:	Condicional: **if-then-else** compacto
= += -= *= /= %=	Atribuição

3.3.2 Conversão entre Tipos de Dados

Ao trabalhar com expressões, salvo quando todos os operandos são do mesmo tipo, é inevitável ter que considerar conversões entre um tipo de dado e outro. Há basicamente dois tipos de conversões de dados. O primeiro se refere a conversão implícita na qual os dados são convertidos automaticamente, praticamente sem a preocupação do programador. Isto ocorre no caso de conversão de dados de tipo inteiro para real e de números para strings. Por exemplo:

```
double x;
int i = 20;
x = i;
```

Neste caso o valor do inteiro **i** é convertido automaticamente para um **double** antes de ser armazenado na variável **x**. As regras de conversão implícita empregadas pela linguagem Java são as seguintes:

- os operadores unários ++ e -- convertem um tipo **byte** e **short** para um **int**, e os demais tipos não são afetados;
- para os operadores binários, as regras são um pouco mais complicadas. Para operações envolvendo apenas inteiros, se um dos operandos for **long**, o outro será convertido para um **long** antes de realizar a operação, a qual resultará num **long**. Caso contrário, ambos os operandos são convertidos para um **int** e o resultado será também um **int**, a menos que o resultado da operação seja grande demais para caber num **int**. Nesse caso, o resultado será convertido

para um **long**. Para operações envolvendo números de ponto flutuante, se um dos operadores for **double**, o outro será convertido para **double** antes de realizar a operação e o resultado será um **double**. Do contrário, ambos os operando são convertidos para **float**, e o resultado será um **float**.

Algumas vezes, porém, as conversões implícitas não são suficientes para garantir um resultado esperado em uma expressão. Nesses casos, é importante podermos controlar precisamente a ocorrência de uma conversão de tipo. Isto pode ser feito por meio de um operador unário de conversão. Por exemplo:

```
float eventos = 25.7;
float dias = 7.2;
x = (int) (eventos / dias);
```

O resultado da expressão acima será precisamente 3, isto é a parte inteira de 25.7 dividido por 7.2. A diferença é que o operador de conversão de tipo (int) transformou o valor do quociente 25.7/7.2 em um inteiro, truncando as casas decimais. Note que o operador de conversão é um operador unário de maior precedência, logo, tivemos de colocar a divisão entre parênteses para garantir que o resultado dessa divisão formasse um único operando. A conversão entre quaisquer inteiros e entre inteiros e números de ponto flutuante é permitida. Porém, não é permitido converter dados do tipo **boolean** para nenhum outro tipo, enquanto a conversão entre objetos somente é permitida para classes pais.

Vejamos um outro exemplo de utilização do operador de conversão de tipo:

```
int a = 3;
int b = 2;
double x;
x = a / b;
```

Neste exemplo, desejamos que a variável x tenha o valor do quociente entre a e b. Como x é um dado de ponto flutuante, presume-se que o resultado desejado neste exemplo seja, digamos, **x = 3 / 2 = 1.5**. Como porém os operandos da divisão são ambos inteiros, o resultado será também um inteiro, isto é, teremos **x = 1.0**. Para contornar essa situação, podemos converter explicitamente um dos operandos para um dado em ponto flutuante, fazendo:

```
x = (double) a / b;
```

Observamos que a conversão entre inteiros de maior comprimento para inteiros de menor comprimento pode causar perda de informação na representação dos valores. Caso o valor a ser convertido ocupe mais bits do que o tipo da variável recebendo esse valor, então o valor é truncado.

3.4 Controle de Fluxo

Controle de fluxo é a habilidade de ajustar a maneira como um programa realiza suas tarefas. Por meio de instruções especiais, chamadas comandos, essas tarefas podem ser executadas seletivamente, repetidamente ou excepcionalmente. Não fosse o controle de fluxo, um programa poderia executar apenas uma única seqüência de tarefas, perdendo completamente uma das características mais interessantes da programação: a dinâmica.

Podemos classificar os comandos aceitos pela linguagem Java em basicamente quatro categorias:

Comando	Palavras-chave
Tomada de decisões	**if**-**else**, **switch**-**case**
Laços ou repetições	**for**, **while**, **do**-**while**
Apontamento e tratamento de exceções	**try**-**catch**-**finally**, **throw**
Outros	**break**, **continue**, **return**

3.4.1 Execução Condicional

A forma mais simples de controle de fluxo é o comando **if-else**. Ele é empregado para executar seletivamente ou condicionalmente um outro comando mediante um critério de seleção. Esse critério é dado por uma expressão, cujo valor resultante deve ser um dado do tipo **boolean**, isto é, *true* ou *false*. Se esse valor for *true*, então o outro comando é executado; se for *false*, a execução do programa segue adiante. A sintaxe para esse comando é:

```
if ([condição])
  [comando]    // Executado se a condição for true
```

Uma variação do comando **if-else** permite escolher alternadamente entre dois outros comandos a executar. Nesse caso, se o valor da expressão condicional que define o critério de seleção for *true*, então o primeiro dos dois comandos é executado, do contrário, o segundo.

```
if([condição])
  [comando 1]   // Executado se a condição for true
else
  [comando 2]   // Executado se a condição for false
```

Por exemplo:

```
import java.io*;
public class Sqrt
{
  public static void main(String args[]) throws IOException
  {
    double x;
    BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
    // Lê um dado double a partir do teclado
    System.out.print("x = ");
    x = Double.parseDouble(teclado.readLine());
    // Decide se é possível calcular a raiz quadrada do
    // número dado. Sendo possível, calcula-a. Do contrário
    // emite uma mensagem de aviso
    if(x >= 0)
      System.out.println("raiz quadrada de x e " + sqrt(x));
    else
      System.out.println("x e negativo");
  }
}
```

Quando toda uma lista de comandos deve ser executada seletivamente, há uma maneira bastante conveniente de agrupar longas seqüências de comandos formando uma unidade lógica: trata-se da noção de bloco de comandos. Este consiste basicamente de uma lista de comandos delimitados por chaves { }. Para efeitos de programação, um bloco de comandos é interpretado como se fosse um único comando. Eis alguns exemplos de blocos de comandos:

```
{
  System.out.print("x = ");
  System.out.println(x);
}

{ temp = y; y = x; x = temp; }
```

Desse modo, podemos fazer também a execução seletiva de blocos de comandos, conforme ilustra o seguinte programa para discutir sobre as raízes de uma equação do segundo grau $ax^2+bx+c=0$:

```
import java.io.*;
public class Baskhara
{
  public static void main(String args[])
  {
    double a, b, c, delta;
    BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
    // Requisitando do usuário os dados sobre a equação a ser
    // analisada.
    //
    try
    {
       System.out.println("Digite os coeficientes da equação ax^2+bx+c = 0");
       System.out.print("a = "); a = Double.parseDouble(teclado.readLine());
       System.out.print("b = "); b = Double.parseDouble(teclado.readLine());
      System.out.print("c = ");  c = Double.parseDouble(teclado.readLine());
    }
    catch(java.io.IOException e)
    {
      System.out.println("Falha na entrada dos dados.");
      a = 0;
      b = 0;
      c = 0;
    }
    // Calculando o discriminante da equação
    //
    delta = b * b – 4 * a * c;
    System.out.println( "Equação: ("+a+")x^2+("+b+")x+("+c+")" );
    // Decidindo sobre o número de raízes da equação mediante o
    // valor do discriminante calculado acima
    //
    if ( delta > 0 )    // A eq. tem duas raízes reais
    {
      double r, d, x1, x2;
      r = Math.sqrt(delta);
      d = 2*a;
      x1 = ( -b - r ) / d;
      x2 = ( -b + r ) / d;
      System.out.println("A equação tem duas soluções reais: ");
      System.out.println("   x1 = " + x1);
      System.out.println("   x2 = " + x2);
    }
    else  if ( delta < 0 )         // A eq. não tem raízes reais
              System.out.println("A equação não tem raízes reais.");
          else               // A eq. tem uma raiz real
            {
              double x;
              x = -b / (2*a);
              System.out.println("A equação tem uma única raiz real:");
              System.out.println("   x1 = x2 = " + x);
            }
    System.out.println("Fim da discussão.");
  }
}
```

3.4.1.1 Execução Condicional Compacta

O **if-else** compacto não é propriamente um comando, mas um operador que realiza avaliação seletiva de seus operandos, mediante o valor de uma expressão lógica

semelhante à do comando **if-else**. Se essa expressão for *true*, então um primeiro operando é avaliado; se for *false* então o segundo operando é avaliado. A sua sintaxe é:

```
[expressão condicional]? [expressão 1] : [expressão 2]
```

O campo **[expressão condicional]** deve resultar em *true* ou *false*, **[expressão 1]** e **[expressão 2]** são os operandos, que podem ser expressões quaisquer. O valor resultante da aplicação do operador **if-else** compacto é obviamente igual ao valor do operando que tiver sido avaliado. O tipo desse valor obedece às regras de conversão entre tipos de dados discutida anteriormente.

Para melhor ilustrar o if-else compacto, consideremos o seguinte comando:

```
y = (x < 1)? x * x : 2 - x;
```

Este é logicamente equivalente à seguinte seqüência de comandos:

```
if (x < 1) then
  y = x * x;
else
  y = 2 - x;
```

É importante notar que o **if-else** compacto é um operador de baixa precedência, logo o uso de parênteses para separar seus operandos não é necessário (a menos que mais de um desses operadores esteja presente na mesma expressão). Porém há casos em que o uso de parênteses para definir claramente os operandos é essencial. Por exemplo, y = |x|*sin(x), pode ser codificado como:

```
y = ((x < 0)? -x : x ) * Math.sin(x); // aqui os parenteses são essenciais.
```

3.4.2 Execução Seletiva de Múltiplos Comandos

Freqüentemente, desejamos que um único comando (ou único bloco de comandos) de uma lista seja executado mediante um dado critério. Isso pode ser feito através do alinhamento ou acoplamento de vários comandos **if-else**, do seguinte modo:

```
if ([condição 1])
  [comando 1]
else if ([condição 2])
  [comandos 2])
else if ([condição 3])
  [comando 3]
....
else
  [comando n]
```

Neste código, o **[comando 1]** será executado (e os demais saltados) caso a primeira condição seja *true*, o **[comando 2]** será executado (e os demais saltados) caso a primeira condição seja *false* e a segunda condição seja *true*, e assim sucessivamente. O **[comando n]** (se houver) somente será executado (e os demais saltados) caso todas as condições sejam *false*.

3.4.2.1 Execução Seletiva por Valores

Assim como no caso execução seletiva de múltiplos comandos, há situações em que se sabe de antemão que as condições assumem o valor *true* de forma mutuamente exclusiva, isto é, apenas uma entre as condições sendo testadas assume o valor *true* num mesmo momento. Nesses casos, a linguagem Java provê um comando de controle de fluxo bastante poderoso. Trata-se do comando **switch**, cuja sintaxe é a seguinte:

```
switch([expressão])
{
  case [constante 1]:
    [comando 1]
    break;
  case [constante 2]:
    [comando 2]
    break;
    …
    …
    ...
  case [constante n]:
    [de comando n]
    break;
  default:
    [comando]
}
```

A **[expressão]** pode ser qualquer expressão válida. Esta é avaliada e o seu valor resultante é comparado com as constantes distintas **[constante 1]**, **[constante 2]**, ..., **[constante n]**. Caso esse valor seja igual a uma dessas constantes, o respectivo comando é executado (e todos os demais são saltados). Se o valor for diferente de todas essas constantes, então o comando presente sob o rótulo **default:** é executado (e todos os demais são saltados), caso este esteja presente. Por exemplo:

```
public class test
{
  public static void main(String args[])
  {
    int op;
    op = 2;
    System.out.print("valor de op eh: " + op)
    switch(op)
    {
      case 1:
        System.out.println("case 1: op=" + op);
        break;
      case 2:
        System.out.println("case 2: op=" + op);
        break;
      case 3:
        System.out.println("case 3" + op);
        break;
      default:
        System.out.println("default: op nao esta no limite 1..3");
    }
  }
}
```

3.4.3 Laço de Iteração Enquanto/Faça

Freqüentemente, desejamos que uma tarefa seja executada repetidamente por um programa enquanto uma dada condição seja verdadeira. Isso é possível pela utilização do comando **while**. Este comando avalia uma expressão condicional, que

deve resultar no valor *true* ou *false*. Se o valor for *true*, então o comando subjacente é executado; se a expressão for *false*, então o comando é saltado e a execução prossegue adiante. A diferença é que após executar o comando subjacente, a expressão condicional é novamente avaliada e seu resultado novamente considerado. Desse modo a execução do comando subjacente se repetirá, até que o valor da expressão condicional seja *false*. Observe, porém, que a expressão é avaliada antes de uma possível execução do comando subjacente, o que significa que esse comando pode jamais ser executado.

```
while([condição])
  [comando subjacente]
```

Por exemplo:

```
// Achar o raiz quadrada de 2 pelo metodo de bissecção
public class sqrt2
{
  public static void main(String args[])
  {
    double a, b, x=1.5, erro = 0.05;
    a = 1;
    b = 2; // 1 < (raiz de 2) < 2
    while((b - a) > erro)
    {
      x = (a + b) / 2;
      if (x * x < 2) // x < raiz de 2
        a = x;
      else // x >= raiz de 2
        b = x;
    }
    System.out.println("Valor aproximado de raiz quadrada de 2: " + x);
  }
}
```

Uma das observações importantes é sempre certificar que não ocorra o laço infinito (um laço que nunca para, pois a condição sempre é verdadeira). Caso tenha alguma chance de entrar no laço infinito, coloque um contador ou use o laço **for**.

3.4.4 Laço de Iteração Faça/Enquanto

Um outro tipo de laço de repetição, similar ao enquanto/faça, é o laço faça/enquanto. Este é introduzido por um par de comandos **do**/**while** que tem a seguinte sintaxe:

```
do
  [comando]
while ([condição]);
```

Diferente do laço enquanto/faça, este tipo de laço de repetição executa o comando e em seguida avalia a expressão condicional. A repetição ocorre se o valor dessa expressão for *true*. Se esse valor for *false*, a execução prossegue adiante do **while**. Vejamos o seguinte exemplo:

```
public class Menu
{
  public static void main(String args[])
  {
    char op;
    int i = 0;
    double x = 0;
    do
    {
      System.out.println("\nOpcoes:");
      System.out.println("p - Atribuir: x = 0.5, i = 2");
```

```
      System.out.println("n - atribuir: x = -0.2, i = -1");
      System.out.println("x - ver o valor de x");
      System.out.println("i - ver o valor de i");
      System.out.println("f - fim");
      System.out.print("Sua escolha: "); System.out.flush();
      try
      {
        op = (char) System.in.read();
        System.in.read(); // Para pegar o 'enter'
      }
      catch(java.io.IOException e)
      {
        op = 'q';
      }
      switch(op)
      {
        case 'p':
          x = 0.5;
          i = 2;
          break;
        case 'n':
          x = -0.2;
          i = -1;
          break;
        case 'x':
          System.out.println("\n--> x = " + x);
          break;
        case 'i':
          System.out.println("\n--> i = " + i);
          break;
        case 'f':
          break;
        default:
          System.out.println("\n--> Opcao invalida");
      }
    } while(op != 'f');
    System.out.println("\nAte logo.");
  }
}
```

3.4.5 Laço de Iteração com Contagem

Em certas situações, precisamos de laços de repetições nos quais alguma variável é usada para contar o número de iterações. Para essa finalidade, temos o laço **for**. Este é o tipo de laço mais geral e mais complicado disponível na linguagem Java. Sua sintaxe é a seguinte:

```
for ([expressão 1]; [condição]; [expressão 2])
  [comando]
```

Onde **[expressão 1]** é chamada *expressão de* inicialização, **[condição]** é uma expressão condicional e **[expressão 2]** é uma expressão qualquer a ser executado no final de cada iteração. O laço **for** acima é equivalente a:

```
[expressão 1]
while ([condição])
{
  [comando]
  [expressão 2]
}
```

Isto que dizer que o laço **for** avalia inicialmente a expressão de inicialização. Em seguida, avalia a expressão condicional. Se o valor desta for *true*, então o comando é executado, a segunda expressão é avaliada em seguida, e finalmente o laço volta a avaliar novamente a expressão condicional. Do contrário, se o valor da expressão for *false*, a execução prossegue adiante do laço **for**. Este arranjo é muito

conveniente para manter variáveis de contagem, conforme ilustra o seguinte exemplo:

```
for (i = 0; i < n; i++)
  System.out.println("V[" + i + "]=" + v[i]);
```

Este código imprime os valores de cada elemento **v[i]** de um vetor **v**, para **i** variando de **0** até **n-1**. O operador **++** foi utilizado na última expressão do laço **for** para somar um ao valor da variável de contagem. Caso necessitássemos de um incremento (ou decremento) maior do que um, poderíamos usar os operadores **+=** ou **-=**. Por exemplo, para imprimir todos os números pares entre de **0** até **10**, poderíamos fazer:

```
for(i = 0; i <= 10; i += 2)
  System.out.println(" " + i);
```

Tanto a **[expressão 1]** quanto a **[expressão 2]** do laço **for** permitem acomodar múltiplas expressões, bastando separá-las por vírgula. Por exemplo, a soma de {1/n} n=1,2, .., N pode ser obtida por:

```
for (soma = 0, n = 1; n <= N; n++)
  soma += 1 / n;
```

ou ainda por:

```
for(soma = 0, n = 1; n <= N; soma += 1 / n, n++);
```

3.4.6 Comandos **Break** e **Continue**

O comando **break** é usado para interromper a execução de um dos laços de iteração vistos acima ou de um comando **switch**. Este comando é comumente utilizado para produzir a parada de um laço mediante a ocorrência de alguma condição específica, antes da chegada do final natural do laço. Exemplo:

```
// Achar i tal que v[i] é negativo
for(i = 0; i < n; i++)
  if(v[i] < 0)
    break;
if(i == n)
  System.out.println("elemento negativo não encontrado.");
```

Se a interrupção se der dentro de um laço duplo, o comando **break** provocará a interrupção apenas do laço em que o comando é imediatamente subjacente. Os outros laços continuam normalmente. Exemplo:

```
// Notificando a existencia de elemento nulo em cada linha do matriz A
for(i = 0; i < m; i++)        // Para cada linha i da matriz faça
  for(j = 0; j < n; j++)      // Para cada coluna j da matriz faça
    if(A[i][j] == 0)
    {
      System.out.println("A possui elemento nulo na linha " + i);
      break;
    }
```

O comando **continue** tem a função de pular direto para final do laço. Porém, em vez de interromper o laço como no **break**, ele continua executando o próximo passo do laço.

3.5 Vetores e Matrizes

A linguagem Java dá suporte a vetores e matrizes (*arrays*) de diversas formas. Os vetores constituem uma forma muito conveniente de organizar informações em fileira. Por exemplo, podemos formar um vetor com as notas de cinco alunos de uma sala de aula do seguinte modo:

```
float nota[] = { 7.8, 8.4, 4.2, 1.8, 6.4 };
```

Neste caso nota[0] é a nota do primeiro aluno, isto é, 7.8, nota[1] é a nota do segundo, ou seja, 8.4, e assim por diante.

A utilização de vetores e matrizes em Java envolve três etapas:

1. **Declarar o vetor ou matriz.** Para isto, basta acrescentar um par de colchetes antes ou depois do nome da variável. Por exemplo:

   ```
   int ind[];
   double a[][];
   int[] nota;
   ```

2. **Reservar espaço de memória e definir o tamanho**. É preciso definir o tamanho do vetor, isto é, a quantidade total de elementos que terá de armazenar. Em seguida é necessário reservar espaço de memória para armazenar os elementos. Isto é feito de maneira simples pelo operador **new**:

   ```
   ind = new int[10];
   nota = new int[70];
   a = new double[10][20];
   ```

3. **Armazenar elementos no vetor ou matriz**. Para armazenar uma informação em um dos elementos de um vetor ou matriz, é necessário fornecer um índice que indique a posição desse elemento. Por exemplo, para armazenar um valor na quarta posição do vetor **nota**, fazemos o seguinte:

   ```
   nota[3] = 5.2;
   ```

   Como podemos observar, os índices começam em zero e vão até o número de posições reservadas, menos um. No vetor **nota** criado acima, os índices válidos vão de 0 até 69. Caso haja a tentativa de atribuir um valor a um elemento cujo índice esteja fora desse intervalo, ocorrerá um erro que impedirá a execução do programa. Por isso, é necessário um certo cuidado ao manejar com esses índices, garantindo o perfeito funcionamento do programa.

Existe um atalho para esses três passos quando desejamos criar um vetor com valores atribuídos de modo estático. Foi o que fizemos no primeiro exemplo acima, declarando o vetor **nota** com as notas de cinco alunos. Nesse caso o espaço suficiente para as notas de cinco alunos foi reservado e as notas foram guardadas em respectivas posições do vetor.

Entretanto, nem sempre é tão fácil assim. Em geral, estaremos interessados em trabalhar com vetores muito maiores, e cujos elementos sejam provenientes de outras fontes, que variam com o tempo. Assim, seremos obrigados a seguir os passos acima.

Eis mais alguns exemplos de vetores e matrizes:

```
// 12 primeiros termos da seqüência de Fibonacci:
long Fibonacci[] = {1,1,2,3,5,8,13,34,55,89,144};
// Tabela de sen(n*pi/6), n = 0,1,2,...5
float seno[] = {0.0000,0.5000,0.8660,1.0000,0.8660,0.5000};
// Tabela de log(1+n), n = 0,2...99:
double tlog[] = new double[100];
for(int n = 0; n < 100; n++) tlog[i] = Math.log(1+n);
// Matriz dos coeficientes
double a[][] = {{1,2,3}, {0,1,3}, {0,0,-1}};
```

3.6 Métodos

Em contraste com a estática dos dados, os métodos definem as ações a serem tomadas em diversos momentos da execução de um programa. Os métodos correspondem aos conceitos comuns de funções, procedimentos ou sub-rotinas. Estes são apenas conjuntos ordenados de declarações de dados, comandos e expressões.

3.6.1 Classes

Os métodos, assim como os dados, têm um local de residência, as classes. Mais adiante, vamos estudar as classes em detalhes. Por hora, precisamos apenas de alguns poucos conceitos para poder entender os métodos. Pensemos numa classe como sendo um conjunto de dados (variáveis) e métodos (funções) da forma:

```
class [nome]
{
  [dados e métodos]
}
```

O campo **[nome]** é um identificador que define o nome da classe e o par de chaves delimita uma região para declaração de variáveis e métodos. A declaração de variáveis já foi vista anteriormente sobre tipos de dados. Uma classe pode ser privativa ou pública. Uma classe privativa é declarada como no exemplo acima e é conhecida apenas no escopo delimitado pelo arquivo que a contém.

Como um programa Java pode ser quebrado em múltiplos arquivos de código fonte distintos, pode ser necessário que as diversas partes integrantes do programa interajam, trocando dados entre si e chamando métodos umas das outras. Isso torna-se possível através das classes públicas, as quais são conhecidas por qualquer arquivo fonte que componha o programa. Para tornar uma classe pública, basta preceder sua declaração pela palavra-chave **public** como no seguinte exemplo:

```
public class [nome da classe]
{
  [dados e métodos]
}
```

Há uma convenção em Java que estabelece que deve haver exatamente uma classe pública para cada arquivo fonte de que consiste um programa Java e seu nome deve ser precisamente o nome do arquivo, sem o sufixo **.java**. Desse modo, existe

uma correspondência biunívoca entre as classes públicas e os arquivos fonte que as contém.

3.6.2 Chamando Métodos

Um método entra em ação no momento em que é chamado. Isto pode ocorrer explicitamente ou implicitamente. A chamada explícita se dá por ordem do programador através da execução de um comando ou expressão contendo o nome do método. Em nosso programa **AloPessoal** fizemos uma chamada explícita do método **System.out.println** para mostrar um texto na tela do computador. As chamadas implícitas ocorrem quando o interpretador Java chama um método por sua própria deliberação. A chamada do método **main** é um exemplo de chamada implícita. O interpretador Java chama esse método para dar início à execução do programa.

3.6.3 Declarando Métodos

A declaração mais simples que podemos fazer de um método (lembrando que isso deve ser feito dentro de uma classe) é a seguinte:

```
void [nome do método] ()
{
  [corpo do método]
}
```

O campo **[nome do método]** é um identificador que define o nome pelo qual o método é conhecido, e **[corpo do método]** consiste de uma lista ordenada de declaração de variáveis, de expressões e de comandos. A primeira palavra-chave, **void**, define o valor retornado pelo método, que, neste caso, não é nenhum. Podemos usar qualquer tipo de dado válido como valor de retorno de um método. Nesse caso, ao terminar, o método seria obrigado a devolver um dado do tipo especificado. Por exemplo:

```
class Numero
{
  double x = 1;
  void print()
  {
    System.out.println("O valor e " + x);
  }
}
```

define uma classe chamada Numero, a qual contém uma variável **x**, inicializada com 1, e um método sem valor de retorno, **print**, que apenas escreve um texto e o valor de **x**, através da chamada do método **System.out.println**.

O par de parênteses adiante do nome do método introduz uma lista (vazia, neste caso) de argumentos. A chamada de um método pode ser acrescida de parâmetros, os quais são associados aos seus respectivos argumentos.

Um exemplo de métodos que retornam valores é o seguinte:

```
class Calculador
{
```

```
  int Soma(int a, int b)
  {
    return a + b;
  }
  double Produto(double a, double b)
  {
    return a * b;
  }
}
```

O primeiro método, **Soma**, realiza a adição de dois números inteiros fornecidos pelos argumentos **a** e **b**, devolve a soma no valor de retorno. O segundo método realiza a multiplicação de dois números de ponto-flutuante **a** e **b** devolvendo seu produto.

A sintaxe completa para a declaração de um método é a seguinte:

```
[moderadores de acesso] [modificador] tipo_do_valor_de_retorno  nome  ([argumentos])
[throws lista_de_exceções]
{
  [corpo]
}
```

onde os termos entre colchetes são opcionais (ou acessórios). Nesta seção vamos analisar detalhadamente cada um dos termos **[moderadores de acesso]**, **[modificador]**, **[tipo do valor de retorno]**, e **[argumentos]**. Vamos, porém, adiar um pouco as explicações sobre **[lista de exceções]** até o capítulo sobre exceções. Uma exceção à esta sintaxe é a que se aplica aos métodos especiais, chamados construtores, que serão vistos adiante na seção sobre classes.

3.6.3.1 Moderadores de Acesso

Os moderadores de acesso são empregados para restringir o acesso a um método. Entretanto, independentemente do moderador escolhido, um método é sempre acessível, isto é, pode ser chamado, a partir de qualquer outro método contido na mesma classe. Os moderadores de acesso existentes em Java são os seguintes:

- **public**: o método declarado com este moderador é público e pode ser chamado a partir de métodos contidos em qualquer outra classe. Esta é a condição de menor restrição possível;
- **protected**: o método é protegido e pode ser chamado por todas as classes que compõe um conjunto maior chamado **package**, assim como dentro de qualquer classe que tenha sido derivada desta classe, ainda que esteja fora do *package*. Veremos adiante que os *packages* são um modo conveniente de organizar diversas classes que estão em estreito relacionamento;
- **friendly**: a chamada a este método é permitida dentro da classe que o contém e dentro de outras classe do *package*. Este é o moderador padrão (default), isto é, aquele que é assumido se nenhum moderador for explicitamente especificado;
- **private**: o método é privativo da classe que o contém e seu uso é vedado a qualquer outra classe.

Exemplo:

```
// classe de numero
class Numero
{
  double x=1;
```

```
  public void print1()
  {
    System.out.println("O valor e "+x);
  }
  private void print2()
  {
    System.out.println("O valor e "+x);
  }
}

// classe principal
public class PrintNum
{
  public static void main(String args[])
  {
    Numero num = new Numero();
    num.print1(); // correto
    num.print2(); // ilegal
  }
}
```

O exemplo acima dará um erro, pois não pode acessar o **print2**. O método **print1** é definido como **public** e, portanto, está disponível a qualquer classe. Mas o método **print2** foi especificado como **private** e, portanto, a classe principal **PrintNum** não pode acessá-lo.

3.6.3.2 Modificador do Método

O modificador do método permite especificar algumas propriedades de um método, determinando como classes derivadas podem ou não redefinir ou alterar o método e de que forma esse método será visível:

- **static:** é um método de Classe. O método é compartilhado por todos os objetos instanciados a partir da mesma classe. Um método **static** não pode acessar qualquer variável declarada dentro de uma classe (salvo se a variável estiver declarada também como **static**, o que veremos mais adiante), pois não é capaz de discernir entre os diferentes objetos que compartilham esse método;
- **abstract**: Podemos declarar um método sem contudo especificar seu corpo, dizendo-o abstrato. Isto funciona como uma espécie de lembrete para que alguma classe derivada complete a declaração fornecendo um corpo. Assim sendo, uma classe que contenha um método abstrato ou que seja derivada de alguma classe que contenha um método abstrato mas não complete sua declaração, não pode ser instanciada.
- **final**: especifica que nenhuma classe derivada pode alterar ou redefinir este método. Um método declarado como final deve ser obrigatoriamente seguido de um corpo;
- **native**: declara apenas o cabeçalho sem corpo, como no caso de **abstract**. Porém esta especificação designa um método implementado em outra linguagem, como C++ por exemplo;
- **synchronized**: esta declaração é usado para desenvolver o programa de processamento concorrente. Seu propósito é impedir que dois métodos executando concorrentemente acessem os dados de uma classe ao mesmo tempo. **Synchronized** especifica que se um método estiver acessando o dado, outros que também desejem acessá-lo têm que esperar.

3.6.3.3 Tipo de Valor de Retorno

O tipo de valor de retorno é especificado por uma palavra chave ou nome de classe na declaração do método, estabelecendo o valor que ele pode devolver:

```
// Classe de números complexos
class Complexo
{
  double x, y;       // parte real e complexo, respectivamnte
  public double Re()  // retorna a parte real
  {
    return x;
  }
  public double Im()  //retorna a parte imaginaria
  {
    return y;
  }
  public Complexo Vezes(Complexo c)
  {
    Complexo resultado;
    resultado.x = x * c.x - y * c.y;
    resultado.y = x * c.y + y * c.x;
    return resultado;
  }
  public void print()
  {
    System.out.println("(" + x + " + " + y + "i)");
  }
}

public class Teste
{
  public static void main(String args[])
  {
    Complexo z, w;
    z.x = 1;
    z.y = 2;
    System.out.print("O valor de z é ");
    z.print();
    System.out.println("A parte real de z é = " + z.Re());
    System.out.println("A parte imaginária de z é = ", z.Im());
    System.out.print("O valor de z ao quadrado é ");
    w = z.Vezes(z);
    w.print();
  }
}
```

Ao executar esse programa teríamos a resposta:

```
O valor de z é (1 + 2i)
A parte real de z é = 1
A parte imaginária de z é = 2
O valor de z ao quardado é (-3 + 4i)
```

Um método que retorna valor, isto é, não declarado como **void**, deve conter a linha **return ...;** a qual especifica o valor a ser retornado. Por exemplo, **return x;** especifica que o valor da variável **x** será retornado.

3.6.3.4 Lista de Argumentos

A lista de argumentos é a lista de valores que o método vai precisar, obedecendo a sintaxe:

```
[tipo 1] [nome 1], [tipo 2] [nome 2], ...
```

O campo **[tipo ?]** é a especificação do tipo de dado e o campo **[nome ?]** é um identificador pelo qual o parâmetro é conhecido:

Exemplo:

```
// Classe de numeros complexos
class Complexo
{
  double x=1, y=1; // parte real e complexo, respectivamnte
  public double Re() // retorna a parte real
  {
    return x;
  }
  public double Im()  //retorna a parte imaginaria
  {
    return y;
  }
  public set(double a, double b)
  {
     x = a;
     y = b;
  }
}

public class Teste
{
  public static void main(String args[])
  {
    Complexo c = new Complexo();
    c.set(3,2);
    System.out.println("z = (" + c.Re() + " + " + c.Im() + "i)");
  }
}
```

Uma observação importante é que Java trata o objeto por referência e por isso, se o método modificar o conteúdo do objeto recebido como parâmetro, o objeto será modificado externamente. No entanto, se o parâmetro recebido for tipo primitivo, o método pode alterar o parâmetro a vontade que não influencia no valor externo. Ou seja, em Java a passagem de parâmetros ocorre apenas por valor.

Para ilustrar, veja o exemplo a seguir:

```
// classe de inteiro
class inteiro
{
  public int i;  // não deve ser utilizado desta forma. Mas neste momento, vamos empregar assim pois não
                 // vimos todos os conceitos necessários ainda.
}

// classe principal
public class TestPar
{
  // método que altera o valor do parametro int
  // este metodo deve ser static, pois sera chamado
  // pelo metodo main() que e static
  static void MudaInt(int k)
  {
    System.out.println("MudaInt: valor de k e " + k);
    k += 2;
    System.out.println("MudaInt: valor de k e apos incremento e " + k);
  }
  // método que altera o valor do parametro inteiro
  // este metodo deve ser static, pois sera chamado
  // pelo metodo main() que e static
  static void MudaInteiro(inteiro k)
  {
    System.out.println("MudaInteiro: valor de k e " + k.i);
    k.i += 2;
    System.out.println("MudaInteiro: valor de k e apos incremento e " + k.i);
  }
  // main() do TestPar
```

```
  public static void main(String args[])
  {
    int i;
    inteiro n = new inteiro();
    i = 2;
    n.i = 3;
    System.out.println("Valores originais:");
    System.out.println("Valor de i e " + i);
    System.out.println("Valor de n e " + n.i);
    MudaInt(i);
    MudaInteiro(n);
    System.out.println("Valores apos a chamada dos metodos:");
    System.out.println("Valor de i e " + i);
    System.out.println("Valor de n e " + n.i);
  }
}
```

A especificação public de uma variável dentro da classe, faz com que este variável seja acessado de qualquer lugar. Para especificar a variável **i** do objeto **K**, basta escrever **k.i**. Em geral, a variável **[var]** dentro do objeto **[obj]** do tipo classe e referenciado por **[obj].[var]**. Note o uso de especificação **static** para os métodos **MudaInt()** e **MudaInteiro()**. Esta especificação é necessária, pois o método principal é **static**, o qual não pode chamar outros métodos da mesma classe que não sejam **static**.

O exemplo acima fornece a saída:

```
Valores originais:
Valor de i e 2
Valor de n e 3
Valores apos a chamada dos metodos:
Valor de i e 2
Valor de n e 5
```

3.7 Classes

3.7.1 Introdução

Na seção sobre métodos, antevimos alguns conceitos básicos sobre classes. Agora, vamos aprofundar mais esses conceitos, permitindo-nos elaborar classes mais sofisticadas, com toda a funcionalidade que elas permitem.

Usamos as classes para construir objetos, o que é chamado de instanciação. E os objetos consistem a essência da programação orientada a objetos (ou *POO*, do inglês *Object-Oriented Programming*). Falando intuitivamente, as classes são uma maneira de organizar um conjunto de dados e designar todos os métodos necessários para usar ou alterar esses dados.

O conjunto de todos os dados contidos em uma classe define o estado de um objeto. Por exemplo, se tivéssemos uma classe **Semaforo** contendo uma única variável chamada **luzAcesa**, então o estado de **Semaforo** é determinado pelo valor de **luzAcesa**:

```
public class Semaforo
{
  int luzAcesa = 0; // 0=vermelho,1=verde,2=amarelo
  void Alternar()
  {
    luzAcesa = ++luzAcesa % 3;
  }
}
```

Os métodos de uma classe, por sua vez, determinam a utilidade que uma classe terá. No caso da classe **Semaforo**, seu único método **Alternar** tem como função provocar a mudança da luz de vermelho a verde, de verde a amarelo e de amarelo a vermelho, respectivamente, em cada nova chamada. Assim, se o método **Alternar** for chamado em intervalos de tempo regulares, poderemos utilizar o estado da classe **Semaforo** para controlar um semáforo com luzes reais.

Para distinguir entre variáveis declaradas em classes daquelas declaradas localmente dentro de métodos, comumente nos referimos àquelas como atributos. Assim, dizemos que **luzAcesa** é um atributo da classe **Semaforo**.

3.7.2 Atributos e Variáveis Locais

Chegou o momento de discutirmos sobre a forma como as variáveis são declaradas dentro de um programa. Podemos declarar variáveis no corpo de uma classe, chamadas atributos, e variáveis no corpo de métodos, chamadas variáveis locais.

A capacidade de acessar uma variável de uma classe depende fundamentalmente de duas coisas: moderadores de acesso e localização da variável dentro da classe. As variáveis locais somente são acessíveis pelo método que as declara, enquanto que os atributos dependem dos moderadores. Apesar de ser possível deixar todos os atributos de uma classe publicamente acessíveis, isto não é recomendável. Do contrário estaríamos desperdiçando o sofisticado mecanismo de proteção de dados fornecido pela POO (encapsulamento), o qual permite escrever programas mais robustos e livres de erros.

Os possíveis moderadores empregados na declaração de atributos são os seguintes:

- **friendly**: todos os campos são **friendly** por **default**. Isto significa que são acessíveis por outras classes integrantes do mesmo **package** e não são acessíveis por nenhuma classe ou subclasse exterior ao **package**;
- **public**: idêntico ao moderador de acesso dos métodos. O campo é acessível a partir de qualquer outra classe, independentemente do **package**;
- **protected**: os campos **protected** podem ser acessados a partir de qualquer classe derivada da classe atual, mas não são acessíveis de fora do package;
- **private**: é o maior grau de proteção. Um campo **private** é acessível unicamente pela classe atual;
- **static**: um campo **static** é compartilhado por todas as instâncias de uma classe, isto é, há um único valor para esse campo, independentemente da quantidade de instâncias existentes, mesmo que não haja nenhuma;
- **final**: um modificador **final** precedendo um campo declara esse campo como uma constante. Seu valor não pode mudar durante a execução do programa. Por isso, é necessário que haja uma inicialização de campo. Por exemplo:

  ```
  final int MaxDimen = 10;
  ```

  O uso de constantes dentro de um programa torna-o mais facilmente legível e fácil de seguir. Para economizar memória, é recomendável também declarar constantes como **static**.

3.7.3 Declarando uma Classe

A forma geral da declaração de uma classe é a seguinte:

```
[modificadores] class nome_classe [extends nome_super] [implements nome_interface]
```

As partes que aparecem entre colchetes são opcionais. Como podemos notar, há quatro diferentes propriedades de uma classe definidas por essa declaração:

- Modificadores
- Nome de classe
- Super classes
- Interfaces

Vamos ver em detalhes cada uma destas propriedades a seguir.

3.7.3.1 Modificadores

Os modificadores de uma classe determinam como uma classe será manipulada mais tarde no decorrer do desenvolvimento do programa. Estes são muito parecidos com os moderadores de acesso introduzidos anteriormente na seção sobre métodos.

Ao declarar uma nova classe, é possível especificar um dos seguintes modificadores:

- **public**: permite definir classes públicas. Estas classes são acessíveis a partir de qualquer objeto, independentemente do package. Uma classe pública deve ser a única classe desse tipo no arquivo em que está declarada e o nome do arquivo deve ser igual ao da classe;
- **friendly**: se nenhum modificador de classe for especificado, então a classe será considerada friendly. Apenas os objetos integrantes do mesmo package podem utilizar uma classe friendly;
- **final**: uma classe final pode ser instanciada, mas não pode ser derivada, isto é, não pode ser superclasse de nenhuma subclasse;
- **abstract**: classes abstratas são aquelas que contém ao menos um método incompleto. Desse modo uma classe abstrata não pode ser instanciada, mas pode ser derivada. Neste caso, a subclasse deve prover o corpo do método para que possa ser instanciada. Isto é muito útil quando desejamos definir em uma classe regras gerais para o comportamento de uma parte do programa, para que, mais tarde, as regras mais específicas sejam introduzidas por subclasses.

3.7.3.2 Nome de Classe

Como qualquer identificador em Java, o nome de uma classe deve obedecer às seguintes regras:

- Iniciar com uma letra, ou um dos caracteres: '$', '_';
- Conter somente caracteres Unicode considerados letras, dígitos ou um dos dois caracteres acima;
- Não pode ser igual a uma palavra-chave reservada pela linguagem Java, tal como **void**, **int**, **for**, **while**, **etc**.

Lembre-se: as letras maiúsculas e as minúsculas são consideradas diferentes.

3.7.3.3 Super Classes (extends)

Um dos aspectos mais importantes da POO é a capacidade de usar atributos e métodos de uma classe previamente construída. Por meio da extensão de classes simples podemos construir classes maiores, acrescentando àquelas mais atributos e métodos, obtendo com isto mais funcionalidades. Neste processo, há uma grande economia no esforço de codificação. Sem esse recurso, freqüentemente seria necessário re-codificar grande parte dos programas para acrescentar-lhes funcionalidade ou fazer modificações significativas.

Ao derivar uma classe, estamos primeiramente fazendo “uma cópia da classe mãe”. É exatamente isto que obtemos se deixarmos vazio o corpo da subclasse. Tal classe se comportaria exatamente como sua superclasse. Entretanto, podemos acrescentar novos atributos e métodos à subclasse, além de sobrepor métodos existentes na superclasse, declarando-os exatamente como na superclasse, exceto por dar um corpo diferente.

Não existe herança múltipla em Java, ou seja, somente é possível derivar uma classe a partir de uma outra e não de várias.

3.7.4 Instanciando uma Classe

Uma classe define um tipo de dado. Ela não pode ser usada a não ser que seja instanciada. Uma instância é um objeto do tipo definido pela classe. Qualquer classe (desde que não seja **abstract**) pode ser instanciada como qualquer outro tipo de dado da linguagem Java. O trecho de código abaixo exibe uma classe chamada **Geometria** criando um a instância da classe **Vértice**:

```
public class Geometria
{
  Vértice v = new Vértice(1.2, 3.5);
  ...
}
```

A diferença mais evidente entre a declaração de um objeto de uma classe e a declaração de um dado primitivo reside na necessidade de reservar memória para o objeto através do uso do operador **new**. Na verdade, esse operador realiza uma série de tarefas:

- Reserva espaço para a instância da classe **Vértice**, o qual deve ser suficiente para conter seu estado, isto é, os valores dos seus campos;
- Realiza a chamada do método construtor;
- Retorna uma referência para o novo objeto, o qual é atribuído àvariável **v**.

Outra importante diferença entre objetos e dados de tipo primitivo é que estes são sempre referenciados por valor, enquanto aqueles são sempre referenciados por meio de sua referência. Isto tem impacto significativo na maneira como os objetos são passados como parâmetros na chamada de métodos. Se o método realizar internamente alguma modificação no objeto que foi passado, essa modificação

refletirá no objeto original. Isto não ocorre com a passagem de dados de tipo primitivo.

Após instanciar uma classe é desejável podermos acessar algum de seus atributos ou então algum de seus métodos. Dentro de uma classe os atributos e métodos são acessíveis imediatamente pelo nome. Repare como na classe **Computador** acima o método **Desligar** acessa diretamente o atributo **ligado**: simplesmente por meio do seu nome.

Entretanto, considere a seguinte classe chamada **CPD** a qual contém várias instâncias da classe **Computador**:

```
public class CPD
{
  Computador Gauss = new Computador();
  Computador Davinci = new Computador();
  Computador Fermat = new Computador();
  ...
  public void Fechar()
  {
    Gauss.Desligar();
    Davinci.Desligar();
    Fermat.Desligar();
  }
  ...
}
```

O método **Fechar** realiza o desligamento de cada particular instância da classe **Computador** chamando seu método **Desligar**. Para indicar a qual objeto o método se refere, devemos precedê-lo do nome do objeto seguido de um operador ponto '.'. A notação geral é:

```
[nome da instância].[nome do método ou variável]
```

Uma exceção a essa regra aplica-se àreferência de campos ou métodos declarados como **static**. Tais declarações são compartilhadas por todas as instâncias de uma classe, por isso não fornecemos o nome de uma particular instância, mas o nome da própria classe ao referenciá-los.

3.7.5 Objetos

Uma particular instância de uma classe é chamada objeto. Comparamos as classes às fábricas e os objetos aos produtos feitos por elas. A grosso modo, podemos dizer que as classes não ocupam espaço na memória por serem abstrações, enquanto que os objetos ocupam espaço de memória por serem concretizações dessas abstrações.

Nas declarações acima, introduzimos algumas classes que permitem representar polígonos. Porém, não instanciamos nenhuma das classes criando particulares objetos a partir dessas classes. Por exemplo, a partir da classe **Quadrado**, podemos fazer objetos representando quadrados de diversos comprimentos laterais, ou retângulos de diferentes dimensões:

```
Quadrado A, B, C;
Retângulo D;
A.lado = 1;    // O quadrado A terá os lados de comprimento 1
B.lado = 2;    // O quadrado B terá os lados de comprimento 2
C.lado = 7;    // O quadrado C terá os lados de comprimento 7
D.base = 3;    // O retangulo D terá base 3 e ...
D.alt = 4;     // altura 4, e centrado na origem
```

```
D.cx = 0;
D.cy = 0;
```

As declarações acima são semelhantes às que vimos anteriormente, com exceção de que no lugar do nome que identifica o tipo de dado estamos usando o nome de uma classe. Neste exemplo, as classes **Quadrado** e **Retângulo** foram empregadas para declarar os objetos (ou variáveis) A, B, C e D.

Em certo sentido as classes complementam os tipos de dados nativos da linguagem Java, com tipos de dados complexos criados pelo programador. Esse fato, aliado à possibilidade de derivar classes, tornam as linguagens orientadas a objetos extremamente producentes.

3.7.6 Construtores

Os construtores são métodos muito especiais, a começar pela sua sintaxe declarativa, e também por suas propriedades e finalidade únicas. Por exemplo, o construtor da classe Vértice vista acima é o seguinte:

```
Vértice(double x, double y)
{
  this.x = x;
  this.y = y;
}
```

Sua única finalidade é inicializar o objeto com um par de coordenadas fornecidas no momento da instanciação. Aliás, esta é a principal finalidade dos construtores: atribuir a um objeto um estado inicial, apropriado ao processamento subseqüente.

Os construtores são métodos facilmente identificáveis pois têm o mesmo nome da classe. Além disso, os construtores não especificam nenhum valor de retorno, mesmo que este seja **void**, uma vez que não são chamados como os outros métodos. Os construtores somente podem ser chamados no momento da instanciação. Por exemplo:

```
Vértice v = new Vértice(1.0, 2.0);
```

Temos neste trecho de código a instanciação da classe **Vértice**, que ocorre no momento em que reservamos espaço para conter um novo objeto dessa classe. Nesse momento o construtor **Vértice** é chamado com os argumentos **1.0** e **2.0**.

É usual declarar os construtores como públicos. Isto porque, se eles tiverem um nível de acesso inferior ao da classe propriamente dita, outra classe será capaz de declarar uma instância dessa classe, mas não será capaz de realizar ela mesma a instanciação, isto é, não poderá usar o operador **new** para essa classe. Há situações, porém, em que essa característica é desejável. Deixando seus construtores como privativos, permite a outras classes usar métodos estáticos, sem permitir que elas criem instâncias dessa classe.

Uma classe pode ter múltiplos construtores declarados, desde que cada um tenha lista de argumentos distinta dos demais. Isto é muito útil, pois em determinados contextos do programa um objeto deve ser inicializado de uma maneira particular em relação a outros contextos possíveis.

Quando nenhum construtor é declarado explicitamente, um construtor vazio é provido implicitamente. Por exemplo, se não tivéssemos especificado um construtor na classe Vértice, este seria o construtor padrão:

```
Vértice()
{
}
```

Os construtores não podem ser declarados com os modificadores: **native**, **abstract**, **static**, **synchronized** ou **final**.

3.7.7 Herança

Uma das maiores vantagens da POO reside na possibilidade de haver herança entre classes. Herança consiste na capacidade de construir novas classes a partir de outras existentes. Nesse processo, os dados e métodos de uma classe existente, chamada de classe base ou superclasse, são herdados pela nova classe, chamada classe derivada ou subclasse.

Este é um recurso muito útil, pois permite aproveitar um esforço de elaboração anterior – reutilização - aumentando significativamente a produtividade da programação e diminuindo o tamanho do código objeto. Suponhamos por exemplo, que tenhamos declarado previamente a seguinte classe:

```
class Polígono
{
  int cx, cy;   // Coordenadas do centro do polígono
}
```

Esta classe define em linhas gerais o que é um polígono, guardando uma única característica comum a qualquer polígono, isto é, as coordenadas de seu centro. Agora, suponhamos que desejamos criar uma classe para guardar informações sobre um quadrado. Neste caso, não precisamos criar uma classe que dê as coordenadas do centro do quadrado assim como as suas dimensões. Basta fazer simplesmente:

```
class Quadrado extends Polígono
{
  int lado;    // Comprimento de um lado do quadrado
}
```

A classe **Quadrado** declarada desse modo se diz uma classe derivada da classe **Poligono**, da qual herda os dados (e os métodos) nela contidos. A declaração anterior é equivalente a:

```
class Quadrado
{
  int cx, cy;    // Coordenadas do centro do polígono
  int lado;      // Comprimento de um lado do quadrado
}
```

porém esta não se beneficia das vantagens da herança.

Desejando fazer uma classe para representar um retângulo, bastaria fazer então

```
class Retângulo extends Poligono
{
  int base, alt;  // Base e altura do retângulo
}
```

3.7.8 Encapsulamento e Polimorfismo

Outro benefício importante da POO reside no encapsulamento. Este consiste na habilidade de efetivamente isolar informações do restante do programa. Outro benefício desta prática é garantir que a informação não será corrompida acidentalmente pelo resto do programa. Criamos, assim, programas mais robustos e confiáveis. Para garantir o encapsulamento, devemos declarar os atributos sempre como privados (*private*) ou protegidos (*protected*).

Uma característica importante das classes reside no fato de que as subclasses de uma dada classe são consideradas do mesmo tipo de seu pai. Isto é chamado polimorfismo. Este permite a realização de uma mesma operação sobre diferentes tipos de classes, desde que mantenham algo em comum. Por exemplo, considere a classe **Polígono** e suas derivadas **Retângulo** e **Triângulo** declaradas abaixo. Apesar de retângulos e triângulos serem diferentes, eles ainda são considerados polígonos. Assim, qualquer coisa que fosse permitido fazer com um objeto da classe **Polígono**, seria também permitida para objetos das classes **Retângulo** e **Triângulo**. O seguinte exemplo ilustra o polimorfismo entre essas classes permitindo que se desenhe um polígono, independentemente do prévio conhecimento de que se trata de um retângulo ou de um triângulo.

```
class Vértice
{
  public double x, y;
  public Vértice(double x, double y)
  {
    this.x = x;
    this.y = y;
  }
}

abstract class Polígono
{
  int numVértices;   // Quantidade de vértices do polígono
  Vértice v[];       // Coordenadas de seus vértices
  abstract void desenhar(Gráfico onde);
}

class Gráfico
{
  // Saída gráfica hipotética para desenhos, com os métodos de desenho necessários...
}

class Retângulo extends Polígono
{
  // construtor
  Retângulo(Vértice param[])
  {
    this.v = new Vértice[4];
    numVértices = 4;
    for(int i = 0; i < 4; i++)
      this.v[i] = param[i];
  }
  void desenhar(Gráfico onde)
  {
              onde.retangulo(this.v);
              onde.pintar(this.v[0], this.v[2]);
  }
}

class Triângulo extends Polígono
{
  Triângulo(Vértice param[])
  {
    this.v = new Vértice[3];
```

```
    numVértices=3;
    for(int i = 0; i < 3; i++)
      this.v[i] = param[i];
  }
  void desenhar(Gráfico onde)
  {
            onde.triangulo(this.v);
            onde.pintar(this.v[0]);
  }
}

public class Polimorfismo
{
  // Coordenadas dos vértices de um retângulo - valores pré-determinados para simplificar a instanciação
  static Vértice v[] =
  {
    new Vértice(0.0,0.0),
    new Vértice(2.0,0.0),
    new Vértice(2.0,1.0),
    new Vértice(0.0,1.0)
  };
  // Coordenadas dos vértices de um triângulo - valores pré-determinados para simplificar a instanciação
  static Vértice w[] =
  {
    new Vértice(-1.0,0.0),
    new Vértice(1.0,0.0),
    new Vértice(0.0,1.0)
  };

  // Exemplo
  public static void main(String args[])
  {
    Polígono r, t;
    Gráfico g = new Gráfico();
    r = new Retângulo(v);    // Isto é válido, pois Retângulo é um Polígono
    t = new Triângulo(w);    // Isto é válido, pois Triângulo é um Polígono
    // Desenha o retângulo na área gráfica g - polimorfismo
    r.desenhar(g);
    // Desenha o triângulo na área gráfica g - polimorfismo
    t.desenhar(g);
  }
}
```

3.7.9 Sobreposição

Não é permitido declarar em uma mesma classe dois métodos com o mesmo nome e mesma lista de argumentos. De fato, isto parece não fazer nenhum sentido, pois os métodos são unicamente identificados pelo nome e pela lista de argumentos que os acompanha.

Entretanto, uma das finalidades de permitir a derivação de classes é atribuir a elas novas funcionalidades. Isto é possível acrescentando-se novos métodos às subclasses. Mas também é possível sobrepor qualquer dos métodos existentes na superclasse, declarando o novo método na subclasse exatamente com o mesmo nome e lista de argumentos, como consta na superclasse. Por exemplo, considere a classe Computador abaixo:

```
class Computador
{
  private boolean ligado = true;
  public void Desligar()
  {
    ligado = false;
  }
}
```

Esta classe permite que o computador seja desligado, através da chamada do método **Desligar**. Porém, isto pode não ser muito seguro, pois poderíamos desligar

o computador mesmo quando ele estiver executando algum programa. Nesse caso, podemos evitar uma catástrofe derivando a classe computador do seguinte modo:

```
class ComputadorSeguro extends Computador
{
  private boolean executando = true;
  public void Desligar()
  {
    if (executando)
      System.out.println("Há programas rodando. Não desligue!");
    else
      ligado = false;
  }
}
```

Agora, um objeto da classe ComputadorSeguro somente será desligado quando não tiver programas rodando.

A sobreposição somente acontece quando o novo método é declarado com exatamente o mesmo nome e lista de argumentos que o método existente na superclasse. Além disso, a sobreposição não permite que o novo método tenha mais proteções do que o método original. No exemplo acima, como o método **Desligar** foi declarado como **public** na superclasse, este não pode ser declarado **private** na subclasse.

3.7.10 A Especificação **this**

Vimos acima como fazer referências a partes de classes. Mas, e se desejássemos fazer referência a partes da própria classe? Veja o exemplo da classe Vértice. Nessa classe o método construtor declara dois argumentos **x** e **y**, os quais têm o mesmo nome dos campos **x** e **y** da classe. Esse método distingue os argumentos dos campos pelo uso da especificação **this**. Assim **this.x** e **this.y** referem-se aos campos **x** e **y** declarados na classe, enquanto **x** e **y** propriamente ditos referem-se aos argumentos do construtor. A palavra **this** substitui uma referência à própria classe.

Há basicamente duas situações em que devemos empregar a palavra **this**:

- Quando houver duas variáveis com mesmo nome numa mesma classe - uma pertencendo à classe e outra pertencendo a algum dos métodos da classe. Nesse caso, apenas esse método específico requer o uso do **this** se quiser fazer referência ao campo da classe;
- Quando uma classe precisa passar uma referência de si própria a um método. Vamos ter a oportunidade de explorar este aspecto quando estivermos estudando os applets.

3.7.11 A Especificação **super**

A palavra **super** provê acesso a partes de uma superclasse a partir de uma subclasse. Isto é muito útil quando estamos sobrepondo um método. Poderíamos reescrever o método Desligar da classe ComputadorSeguro do seguinte modo:

```
class ComputadorSeguro extends Computador {
  private boolean executando = true;
  public void Desligar()
```

```
  {
    if ( executando )
      System.out.println("Há programas rodando. Não desligue!");
    else
      super.Desligar();
  }
}
```

Note a chamada **super.Desligar()**. Esta corresponde a chamada do método Desligar declarado na superclasse **Computador**, o qual vai efetivamente ajustar o campo **ligado** para o valor *false*. Imaginando que o método **Desligar** fosse muito mais complicado, não precisaríamos recodificá-lo completamente na subclasse para acrescentar a funcionalidade que permite o desligamento apenas quando o computador estiver desocupado.

No caso de o método que desejamos chamar ser um construtor, a chamada usando a palavra **super** é bem particular. Examinemos o seguinte exemplo de uma classe chamada **VérticeNumerado** que estende a classe **Vértice**, acrescentando às coordenadas do vértice um rótulo numérico que o identifica visualmente:

```
class VérticeNumerado extends Vértice
{
  int numero;
  VérticeNumerado(int numero, int x, int y)
  {
    this.numero = numero;
    super(x, y);
  }
}
```

Note que a chamada **super(x, y)** se traduz na chamada do construtor **Vértice(x,y)** da superclasse. Com isto, evitamos de ter que recodificar no novo construtor as tarefas contidas no construtor da superclasse: basta chamá-lo. Vale observar que esse tipo de chamada também só é permitida de dentro de um construtor.

4 Um aprofundamento em Herança e Polimorfismo

4.1 Herança

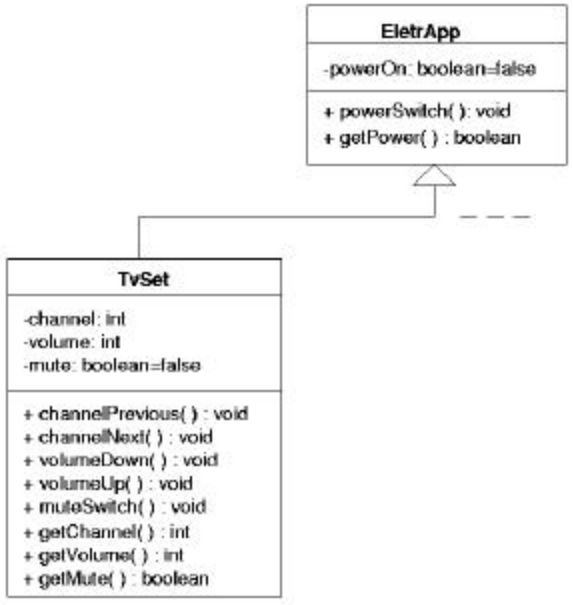
Herança é sempre utilizada em Java, mesmo que não explicitamente. Qualquer classe que se cria está herdando da classe raiz, chamada Object. Esta classe contém as funcionalidades básicas e comuns a todos os objetos, tais como:

- o método equals() para comparar conteúdos de objetos. Se o objeto de uma nova classe contém referências a outros objetos, este método pode ser redefinido para realizar a comparação recursivamente para os componentes;
- o método toString() para converter o estado do objeto para uma representação String, que possa ser textualmente apresentada. Todas as novas classes devem redefinir este método se a conversão automática para string for necessária -- por exemplo, em uma concatenação associada a uma invocação de println().
- o método finalize(), invocado pelo *garbage collector* antes de liberar a memória usada por um objeto não mais referenciado. Se o objeto fizer uso de outros recursos que não a memória alocada por Java, este método deve ser redefinido para liberar tais recursos.

4.1.1 Sintaxe de especificação de herança

Considere uma aplicação de automação residencial, com alguns aparelhos eletrônicos sendo controlados por um software de controle. Uma constatação é que todos os aparelhos têm pelo menos um comportamento comum, que é a opção de ligar ou desligar o aparelho.

Assim, uma possível especificação desta aplicação poderia ser:



Em Java, a palavra chave para especificar herança entre classes é **extends**. A situação acima poderia ser representada pela seguinte estrutura Java:

```
class EletrApp {
   private boolean powerOn;

   public void powerSwitch()    { powerOn = ! powerOn;  }

   public boolean getPower()    { return powerOn; }
}

public class TvSet extends EletrApp {
   // constantes para a classe
   private static final int MINCHANNEL = 2;
   private static final int MAXCHANNEL = 181;
   private static final int MAXVOLUME = 100;
   // atributos da classe
   private int channel = MINCHANNEL;
   private int volume;
   private boolean mute;

// definição de métodos de TvSet
public void channelPrevious() {
  if (getPower()) {
    channel--;
    if (channel < MINCHANNEL)
      channel = MAXCHANNEL;
  }
}
```

```
  public void channelNext() {
    if (getPower()) {
      channel++;
      if (channel > MAXCHANNEL)
        channel = MINCHANNEL;
    }
  }

  public void volumeDown() {
    if (getPower() && volume > 0)
        volume--;
  }

  public void volumeUp() {
    if (getPower() && volume < MAXVOLUME)
        volume++;
  }

  public void muteSwitch() {
    if (getPower())
      mute = !mute;
  }

  public int getChannel()       {       return(channel);}
  public int getVolume()  {     return(volume); }
  public boolean getMute()      {       return(mute);           }
}
```

A aplicação poderá conter invocações como:

```
TvSet aTv = new TvSet();
...
aTv.powerSwitch();
```

4.1.2 Construção de objetos derivados

Objetos derivados invocam implicitamente, a partir de seu construtor, o método construtor de sua superclasse antes de executar o corpo de seu construtor.

Assim, no exemplo acima, ao se construir um objeto da classe TvSet (usando o construtor default, já que nenhum outro foi definido), o método construtor da classe EletrApp será inicialmente invocado. Este, por sua vez, invocará o construtor de sua superclasse, Object. Como Object não tem uma superclasse, seu construtor é executado; então executa-se o restante do construtor de EletrApp e finalmente o restante do construtor de TvSet.

Construtores da superclasse podem ser diretamente invocados usando o método **super()**. Implicitamente, é isto que o compilador faz para cada construtor. Entretanto, a invocação direta pode ser interessante quando se deseja invocar algum construtor que não o *default*.

O prefixo super. pode também ser utilizado para referenciar métodos da superclasse que tenham sido redefinidos pela classe derivada.

4.1.3 final

A palavra chave **final** pode ser utilizada como uma indicação de algo que não deve ser modificado ao longo do restante da hierarquia de descendentes de uma classe. Pode ser associada a atributos, a métodos e a classes.

Um atributo final pode indicar um valor constante, que não deve ser alterado pela aplicação. 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 pode ser modificado -- apenas a referência é fixa.

Um método que é definido como *final* em uma classe não pode ser redefinido (sobrescrito ou sobreposto) em classes derivadas. Por exemplo, todos os métodos da classe java.util.Vector são definidos como *final*. Assim, a tentativa de redefinir algum método em uma classe derivada de Vector, como em

```
public class MyVector extends java.util.Vector {
  //...
  void addElement(Object obj) {
    // minha implementacao de addElement...
  }
}
```

causaria um erro:

```
C:\> javac MyVector.java
MyVector.java:3: The method void addElement(java.lang.Object) declared
  in class MyVector cannot override the method of the same signature
  declared in class java.util.Vector.  The access modifier is made
  more restrictive.
  void addElement(Object obj) {
       ^
1 error
```

Uma classe definida como *final* não pode ser estendida. Assim, a compilação do arquivo Reeleicao.java

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

ocasionaria um erro de compilação:

```
C:\> javac Reeleicao.java
Reeleicao.java:4: Can't subclass final classes: class Mandato
public class Reeleicao extends Mandato {
                               ^
1 error
```

4.1.4 Várias formas de relacionamentos de herança

1. Extensão: 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.
2. 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.
3. 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.

4.1.5 Usando heranças

Como decidir qual o tipo de relacionamento -- composição, herança estrita, herança polimórfica -- que existe entre as classes de sua aplicação?

Em geral, o procedimento requer iterações entre especializações (*top-down*) e generalizações (*bottom-up*).

Especialização: quando uma classe é-um-tipo-de (*is-a-kind-of*) uma outra classe. Se a nova classe puder ser utilizada em qualquer contexto onde a classe original podia ser utilizada, então diz-se que a nova classe é um subtipo da superclasse. Se esta substituição é possível, a subclasse obedece ao Princípio de Substituição de Liskov (por Barbara Liskov). Uma das restrições em subtipos é que o significado dos métodos não deve ser alterado.
Na subtipagem estrita, métodos não podem ser redefinidos, apenas acrescentados.

Generalização: nesse processo, classes com características comuns são reorganizadas como subclasses de uma classe que agrega apenas a parte comum entre as classes originais. É o "outro lado da moeda" da especialização.

Especificação: usada quando, no projeto, você sabe o que uma classe deve fazer, embora não saiba exatamente como fazê-lo. Neste caso, é possível especificar métodos sem implementação, os quais deverão ser efetivamente implementados pelas subclasses.
Na especificação pura, nenhum método é implementado; apenas a especificação da interface é definida.

Contração: seria uma outra forma 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, pois a subclasse já não pode mais 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 reanalisada para detecção de inconsistências (problema pássaros-pinguins). De modo geral, o mecanismo de contração deve ser evitado.

Como saber qual o mecanismo correto para o meu caso?
Prática, prática, prática, ...

4.1.6 Mecanismos de herança em Java

- Extensão simples: palavra-chave *extends*; para herança estrita, marcar todos os métodos da superclasse como *final*.
- Especificação: em Java, uma especificação é implementada como uma *interface*; subclasses da especificação usam a palavra-chave *implements* para indicar este tipo de herança.
- Herança polimórfica: como a extensão simples, usa a palavra-chave *extends*. Usando na superclasse as palavras-chave *final* e *abstract*, é possível indicar que partes da superclasse não podem ser modificadas ou devem ser modificadas.

4.2 Polimorfismo

Mecanismo para expressar comportamentos diferenciados para objetos derivados de uma mesma superclasse em tempo de execução.
Termos correspondentes: *dynamic binding* , *late binding*

Princípio:
1. superclasse expressa métodos que subclasses devem suportar
2. subclasses podem (ou tem que) definir métodos que podem ser diferenciados. No entanto, recomenda-se que métodos de mesmo nome tenham comportamentos consistentes;
3. na invocação do método, pode-se utilizar no programa referências à superclasse; o sistema verifica a qual objeto a invocação refere-se e aplica o método mais adequado.

4.2.1 Exemplo

No exemplo a seguir, a superclasse (Raiz) e as duas classes derivadas (Filho1 e Filho2) tem um método show().
A aplicação (método main da classe Exemplo1) invoca o método relativo ao objeto da classe Raiz, sem saber durante a compilação se o objeto será da classe Raiz, Filho1 ou Filho2.

```
class Raiz {
  public void show() {
    System.out.println("Raiz");
  }
}
```

```
class Filho1 extends Raiz {
  public void show() {
    System.out.println("Filho1");
  }
}

class Filho2 extends Raiz {
  public void show() {
    System.out.println("Filho2");
  }
}

public class Exemplo1 {
  // Metodo recebe um argumento "1" ou "2"
  // para indicar que tipo de filho deve ser
  // associado ao objeto raiz.  Sem argumentos,
  // um objeto raiz (nao filho) e' criado.
  static public void main(String[] args) {
    Raiz r;

    // Algum objeto e' criado
    if (args.length > 0)
      if (args[0].charAt(0) == '1')
        r = new Filho1();
      else {
        if (args[0].charAt(0) == '2')
          r = new Filho2();
        else
          r = new Raiz();
      }
    else
      r = new Raiz();

    // Invocacao do metodo
    r.show();
  }
}
```

Se a referência a objeto r vai estar referenciando um objeto da classe Raiz, Filho1 ou Filho2 só será definido no momento de execução. Mesmo assim, o sistema consegue verificar a que classe a referência está associada e invocar o método que aplica:

```
C:\> javac Exemplo1.java
C:\> java Exemplo1
Raiz
C:\> java Exemplo1 1
Filho1
C:\> java Exemplo1 2
Filho2
C:\> java Exemplo1 3
Raiz
```

Observe neste código que referências a objetos das classes derivadas podem ser atribuídas diretamente a referências à superclasse, sem necessidade de *cast*. Este mecanismo é algumas vezes referenciado como ***upcasting***.

Referências a uma superclasse podem ser atribuídas a referências a objetos de classes derivadas desde que o objeto seja de fato da classe indicada. Por exemplo,

supondo que o programa acima fosse modificado para incluir a seguinte atribuição ao final:

```
    ...
    r.show();
    Filho1 f = (Filho1) r;
```

O *cast* é aqui necessário pois não se caracteriza um *upcast* (na verdade, ocorre um "*downcast*"). Se a aplicação for executada com o argumento 1, tudo está bem:

```
C:\> java Exemplo1a 1
Filho1
```

No entanto, com o argumento 2, o interpretador indica uma condição de exceção:

```
C:\> java Exemplo1a 2
Filho2
Exception in thread "main" java.lang.ClassCastException: Filho2
        at Exemplo1a.main(Compiled Code)
```

4.2.2 Classes e métodos abstratos

No exemplo acima, a superclasse tinha um comportamento válido associado ao método show(). No entanto, nem sempre este é o caso. Em grande parte das situações, uma superclasse especifica apenas uma interface comum que todas suas classes derivadas deverão implementar, mas não como a implementação deve ocorrer.

Um exemplo clássico do uso de polimorfismo envolve classes de objetos geométricos. Uma superclasse Shape indica a funcionalidade que deve ser suportada por formas geométricas em geral; por exemplo, desenhar (draw) ou remover (erase) a forma da tela. No entanto, a implementação destes métodos só faz sentido quando se conhece a forma que se deseja desenhar ou remover, como um retângulo ou círculo.

Seguindo a estrutura do Exemplo1 acima, seria possível criar a superclasse Shape com métodos com corpo vazio (ou com mensagens de erro) e especificar o corpo do método apenas nas classes derivadas:

```
class Shape {
  public void draw() {
    System.out.println("???");
  }
  public void erase() {
    System.out.println("???");
  }
}

class Square extends Shape {
  public void draw() {
    System.out.println("Desenhando quadrado");
  }
  public void erase() {
    System.out.println("Apagando quadrado");
  }
}
```

```
class Circle extends Shape {
  public void draw() {
    System.out.println("Desenhando circulo");
  }
  public void erase() {
    System.out.println("Apagando circulo");
  }
}

public class Exemplo2 {
  // Metodo recebe um argumento "Q" ou "C"
  // para indicar que tipo de filho deve ser
  // associado ao objeto raiz.  Sem argumentos,
  // um objeto Shape e' criado.
  static public void main(String[] args) {
    Shape r;

    // Criacao do objeto
    if (args.length > 0)
      if (args[0].toUpperCase().charAt(0) == 'Q')
        r = new Square();
      else {
        if (args[0].toUpperCase().charAt(0) == 'C')
          r = new Circle();
        else
          r = new Shape();
      }
    else
      r = new Shape();

    // Desenho e remocao da forma
    r.draw();
    r.erase();
  }
}
```

A execução desta classe apresentaria como resultados:

```
C:\> javac Exemplo2.java
C:\> java Exemplo2
???
???
C:\> java Exemplo2 q
Desenhando quadrado
Apagando quadrado
C:\> java Exemplo2 c
Desenhando circulo
Apagando circulo
C:\> java Exemplo2 x
???
???
```

Um dos problemas associados a este exemplo é que não se sabe como lidar com "formas genéricas" -- na verdade, isto nem faz sentido para a aplicação. Neste tipo de situação, seria mais adequado sinalizar que os métodos da classe Shape não têm implementação -- mais ainda, que nem faz sentido criar objetos desta classe.

O mecanismo que Java e outras linguagens orientadas a objetos oferecem para suportar este conceito é a definição de métodos abstratos. Um método abstrato

define apenas uma interface, uma assinatura, sem especificação do corpo. A sintaxe de definição de um método abstrato é:

```
  public abstract void draw();
  public abstract void erase();
```

Uma classe que contenha métodos abstratos deve ser uma classe abstrata. Assim, a tentativa de definir uma classe não abstrata contendo métodos abstratos, como em

```
// Exemplo3.java
class Shape {
  public abstract void draw();
  public abstract void erase();
}

// ...
```

acarretaria em um erro de compilação:

```
C:\> javac Exemplo3.java
Exemplo3.java:1: class Shape must be declared abstract.
It does not define void draw() from class Shape.
class Shape {
      ^
Exemplo3.java:1: class Shape must be declared abstract.
It does not define void erase() from class Shape.
class Shape {
      ^
```

Ainda mais, a tentativa de se instanciar objetos de uma classe abstrata também gera erros de compilação:

```
Exemplo3.java:40: class Shape is an abstract class.
It can't be instantiated.
          r = new Shape();
              ^
Exemplo3.java:43: class Shape is an abstract class.
It can't be instantiated.
      r = new Shape();
          ^
```

Assim, a definição de Shape como classe abstrata requer o uso da palavra chave ***abstract***, e objetos desta classe não podem ser criados:

```
abstract class Shape {
  public abstract void draw();
  public abstract void erase();
}

class Square extends Shape {
  public void draw() {
    System.out.println("Desenhando quadrado");
  }
  public void erase() {
    System.out.println("Apagando quadrado");
  }
}

class Circle extends Shape {
```

```
  public void draw() {
    System.out.println("Desenhando circulo");
  }
  public void erase() {
    System.out.println("Apagando circulo");
  }
}

public class Exemplo3 {
  // Metodo recebe um argumento "Q" ou "C"
  // para indicar que tipo de filho deve ser
  // associado ao objeto raiz.  Sem argumentos,
  // uma mensagem indica forma de uso.
  static public void main(String[] args) {
    Shape r = null;

    // Criacao do objeto
    if (args.length > 0)
      if (args[0].toUpperCase().charAt(0) == 'Q')
        r = new Square();
      else {
        if (args[0].toUpperCase().charAt(0) == 'C')
          r = new Circle();
        else
          System.out.println("Uso: Exemplo3 Q ou Exemplo3 C");
      }
    else
      System.out.println("Uso: Exemplo3 Q ou Exemplo3 C");

    // Desenho e remocao da forma
    if (r != null) {
      r.draw();
      r.erase();
    }
  }
}
```

Resultado:

```
C:\> javac Exemplo3.java
C:\> java Exemplo3
Uso: Exemplo3 Q ou Exemplo3 C
C:\> java Exemplo3 q
Desenhando quadrado
Apagando quadrado
C:\> java Exemplo3 c
Desenhando circulo
Apagando circulo
C:\> java Exemplo3 x
Uso: Exemplo3 Q ou Exemplo3 C
C:\>
```

Toda a classe que seja derivada de uma classe abstrata deve implementar (redefinir) todos os métodos abstratos ou ser também declarada como abstrata. Por exemplo, se a classe Square implementasse apenas o método draw() mas não o método erase(),

```
class Square extends Shape {
  public void draw() {
    System.out.println("Desenhando quadrado");   }
}
```

a seguinte mensagem seria apresentada durante a compilação:

```
C:\>Polimorf[39] javac Exemplo4.java
Exemplo4.java:6: class Square must be declared abstract.
It does not define void erase() from class Shape.
class Square extends Shape {
      ^
```

Classes podem ser declaradas como abstratas mesmo sem ter nenhum método abstrato. Neste caso, indica-se que nenhum objeto pode ser criado para a classe.

4.3 Interface

Uma interface em Java é uma classe abstrata para a qual todos os métodos são implicitamente *abstract* e *public*, e todos os atributos são *static* e *final*.
Em outros termos, uma interface Java implementa uma "classe abstrata pura".

A definição de uma interface Java equivale à definição de uma classe, apenas usando a palavra chave *interface* ao invés da palavra chave *class*:

```
interface Eleccion {
  void vota(String candidato);
  int contagem();
  String vencedor();
}
```

Apesar de não estar explicíto, os dois métodos desta interface são *public* e *abstract*. Se desejado, as duas palavras chaves poderiam ser utilizadas na declaração dos métodos sem alteração na definição da interface.

A diferença entre uma classe abstrata e uma interface Java é que a interface obrigatoriamente não tem um "corpo" associado, apenas forma. Enquanto uma classe abstrata é "estendida" (palavra chave *extends*) por classes derivadas, uma interface Java é "implementada" (palavra chave *implements*) por outras classes:

```
class ConeSurDemocracia implements Eleccion {
  int votos = 0;
  public void vota(String qualquerUm) {
    ++votos;
  }
  public int contagem() {
    return votos > 2 ? votos-1 : 1;
  }
  public String vencedor() {
    return "El Senor Presidente";
  }
}
```

A utilização de uma interface em um programa ocorre da mesma forma que para classes abstratas. Por exemplo, na aplicação a seguir, cria-se uma referência à interface, br, à qual é atribuída uma referência a um objeto que implementa a interface (*upcasting*):

```
public class Exemplo5 {
```

```
  public static void main(String[] args) {
    Eleccion br = new ConeSurDemocracia();
    int i = 0;

    // votacao
    while (i < args.length)
      br.vota(args[i++]);

    // and the winner is...

    System.out.println(br.vencedor() + " foi eleito com " +
                       br.contagem() + " votos!");
  }
}
```

Outra diferença essencial entre classes e interfaces Java é que uma classe pode implementar múltiplas interfaces, mas pode estender apenas uma superclasse. Neste caso, as interfaces implementadas são separadas por vírgulas:

```
  class Derivada extends SuperClasse
       implements Interface1, Interface2 {
       ...
  }
```

Neste exemplo, a classe Derivada deve implementar todos os métodos da Interface1 e todos os métodos da Interface2. Adicionalmente, se SuperClasse também tiver métodos abstratos, esses também deverão ser implementados por Derivada.

Assim como ocorre com classes, interfaces Java também podem ser derivadas de outras interfaces. A diferença é que diversas interfaces podem ser estendidas:

```
interface S1 {
  ...
}

interface S2 {
  ...
}

interface D extends S1, S2 {
  ...
}
```

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 redefinir nenhum método:

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


class CokeMachine implements Moedas {
```

```
  int price = 2*QUARTER;
  // …
}
```

5 Bibliografia

ECKEL, Bruce. Thinking in Java. New Jersey: Prentice-Hall, 2000.

NIEMEYER, P., KNUDSEN, J. Aprendendo Java. Rio de Janeiro: Campus, 2000.

NEWMAN, Alexander. Usando Java : o guia de referência mais completo. Rio de Janeiro: Campus, 1997.

RICARTE, Ivan Luiz Marques. Programação Orientada a Objetos. Página web em http://www.dca.fee.unicamp.br/courses/PooJava/. Acessada em Fevereiro/2001.