APOSTILA DE APOIO

Desenvolvimento de Software

SUMÁRIO

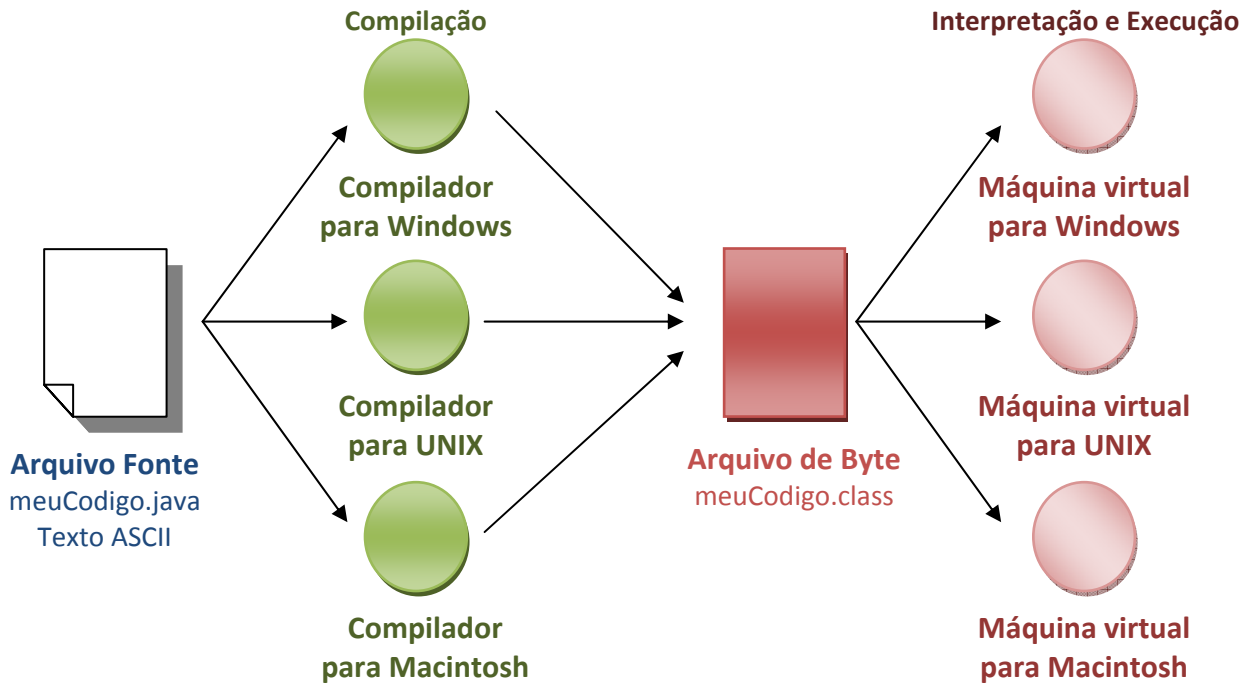
Características gerais da linguagem 1
Particularidades e vantagens 1
Limitações e desvantagens 1
Aplicativos independentes e applets 2
Aplicativo independente 2
Applet 3
Dicas 4
Prática 4
Olá Mundo! - Aplicativos que saúdam o mundo 4
Elementos básicos da síntaxe 7
Comandos 7
Variáveis 7
Comentários 7
Tipos de dados primitivos 8
Números inteiros 8
Números em ponto flutuante 8
Caracteres 9
Booleanos 9
Expressões e operadores 10
Operadores aritméticos 10
Operadores de atribuição 10
Operadores de incremento e decremento 10
Operadores de comparação 11
Operadores lógicos 11
Precedência e associatividade dos operadores 11
Estruturas de controle de fluxo 12
Estruturas de seleção 12
Estruturas de repetição (laços) 13
Prática 14
Laço 14
Execução condicional 14
Cadeias de caracteres 15
Construção 15
Comprimento 15
Comparação 15
Concatenação 16

Conversão de dados numéricos 16
Formatação 17
Decomposição em palavras 17
Listas 18
Construção 18
Comprimento 19
Referência e valor 19
Índices múltiplos 19
Prática 19
Produto escalar de dois vetores 19
Objetos e classes 22
Classes 22
Objetos 22
Variáveis 23
Variáveis de instância e variáveis de classe 23
Restrições de acesso 23
Como um objeto acessa as variáveis de outro objeto 24
Variáveis globais e variáveis locais 25
Variáveis constantes 26
Métodos 27
Valor de retorno e argumentos 27
Métodos estáticos 27
Restrições de acesso 27
Como um objeto chama um método de outro objeto 28
Como criar um objeto - método construtor 28
Prática 29
Classe representando um vetor tridimensional 29
Máquina para cálculo vetorial 29
Utilização no programa da prática anterior 29
Programação orientada a objetos: Aspectos fundamentais 30
Sobrecarga de métodos 30
Herança 30
Superação de métodos 31
Polimorfismo 32
Recursos adicionais 34
Interfaces 34
Classes abstratas 35

Características gerais da linguagem

Particularidades e vantagens

- Programas compilados independentes de plataforma.



Os arquivos de byte são **pequenos** -> carregados rapidamente pela Internet.

- Ampla biblioteca de componentes e recursos - Gratuita!
  - Componentes gráficas
  - Tratamento de eventos
  - Formatação de texto
  - Funções matemáticas
  - Entrada e saída
  - Computação em rede
  - Bancos de dados
  - . . .
- Programação (relativamente) simples e natural.
  - Programação "orientada a objeto" - facilita a reutilização de código.
  - Permite linhas múltiplas de execução.
  - Comparação com C++:
    - Possui herança simples. Implementa uma forma segura de herança múltipla através de "interfaces".
    - Não permite aritmética de ponteiros.
    - Possui desalocação automática de memória ("coletor de lixo").

Limitações e desvantagens

- Velocidade de execução relativamente baixa
  - Pode ser melhorada usando compiladores JIT ("just in time") que transformam o bytecode em código de máquina antes da execução.
- Evolução rápida -> multiplicidade de versões
  - A versão mais recente, batizada Java 2 Platform (SDK1.3) ainda não está implementada nos navegadores. Requer o Java Plug-in http://java.sun.com/products/plugin/

Aplicativos independentes e applets

Aplicativo independente

- Programa que é executado fora de um navegador. Utilize um editor (Bloco de Notas) para digitar e editar o arquivo fonte. Cuidado ao digitar: Java distingue maiúsculas e minúsculas. O esquema mínimo para o programa de um aplicativo independente é:

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

  onde . . . indica comandos que definem o que o programa vai fazer. O programa deve ser salvo num arquivo chamado: MeuAplicativo.java . Vê-se que para construir um programa em Java, precisamos definir uma classe. O corpo da classe, contido entre as chaves mais externas, contem em geral definições de variáveis e métodos, que especificam aquilo que a classe pode fazer. No exemplo acima, o único método presente é o método principal, que deve estar presente em qualquer aplicativo independente e é executado automaticamente quando o aplicativo é executado. Parâmetros podem ser passados ao programa no comando de execução, na forma de uma lista de cadeias de caracteres (tipo String) chamada args[] no exemplo acima. Os colchetes indicam que trata-se de uma lista.

- Não detalharemos agora o significado das demais palavras chaves presentes no código acima. Por enquanto, estamos apenas "dando a receita" para construir o aplicativo o mais simples possível.
- Para compilar o programa, digite na linha de comando (do DOS se você está usando Windows):

```
javac MeuAplicativo.java
```

  Se a compilação for bem sucedida, isto gera uma arquivo MeuAplicativo.class na mesma pasta.

- Para executar o programa, digite na linha de comando:

```
java MeuAplicativo
```

  Se o seu programa requer parâmetros de entrada (digamos a1 e a2), estes devem ser acrescentados ao comando:

```
java MeuAplicativo a1 a2
```

Applet

- Programa que é executado dentro de um navegador. Utilize um editor para digitar e editar o arquivo fonte. O esquema mínimo para o programa de um applet é:

```
import java.applet.*;

public class MeuApplet extends Applet
{
   public void init( )
   {
     . . .
   }
}
```

onde . . . indica comandos que definem o que o programa vai fazer. O programa deve ser salvo num arquivo chamado MeuApplet.java . O método init() é chamado automaticamente pelo navegador quando ele carrega o applet. Note que o pacote java.applet deve ser importado para ter acesso à classe Applet.

- Para compilar o programa, digite na linha de comando:

```
javac MeuApplet.java
```

Se a compilação for bem sucedida, isto gera uma arquivo MeuApplet.class na mesma pasta.

- O applet é carregado por um navegador através de um arquivo HTML. O arquivo mínimo é:

```
<HTML>
<HEAD>
<TITLE>Meu Primeiro Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="MeuApplet.class" WIDTH=400 HEIGHT=160>
</APPLET>
</BODY>
</HTML>
```

Digamos que o nome deste arquivo seja MeuApplet.htm. Para testar o seu applet sem ter que abrir um navegador, digite na linha de comando:

```
appletviewer MeuApplet.htm
```

Dicas

- Console Java
  Procure este item no menu do seu navegador para abrir uma janela que dá informações sobre a máquina virtual e mostra eventuais mensagens de erro.
- Documentação:
  A pasta /docs do JDK contem uma documentação completa em formato HTML. Em especial, a pasta docs/api documenta as bibliotecas.
- Demonstrações
  A pasta /demo do JDK contem vários applets demonstrativos, com arquivos fontes incluídos.
- Caminho
  Para que você possa executar o compilador (javac), o interpretador (java) e o visualizador de applet (appletviewer), é necessário que a pasta /bin do JDK esteja incluída no seu PATH. No Windows, isto é feito editando o arquivo Autoexec.bat.

Prática

Você pode agora construir os seus primeiros aplicativos e applets, seguindo o roteiro apresentado a seguir:

Olá Mundo! - Aplicativos que saúdam o mundo

Seguindo a tradição do ensino da programação, começamos a prática da programação Java com programas que meramente saúdam o mundo. Consultaremos a documentação docs/api para descobrir quais os recursos necessários.

Aplicativo independente não-gráfico

No caso de um aplicativo indendente não-gráfico, a saudação somente pode ser apresentada na linha de comando. Para tanto:

- Consulte a documentação do pacote **java.lang** (para "language") que contém recursos básicos de Java. Olhe na classe **System** e note que ela possui a ***variável*** **out**, que representa a saída padrão, ou seja, no caso de um aplicativo independente, a linha de comando.
- Olhando a descrição deste objeto **out**, você encontra o comando que você precisa usar. Note que ele utiliza o ***método*** **println**, associado ao objeto **out**. Método é o nome utilizado em Java para uma ação que um objeto pode realizar.
- Olhe a documentação a respeito do método **println**. Você vê que ele admite vários tipos de argumento. No caso, o argumento que você vai passar ao método é a **String** (cadeia de caracteres) "Olá Mundo!" (com as aspas, que indicam tratar-se de uma **String**).
- Agora você deve estar pronto para inserir o comando adequado no método **main** do seu aplicativo. Feito isto, compile e execute o programa.
- Reflita sobre a sintaxe do comando, em especial sobre o papel do ***ponto***, que é fundamental em Java. Procure expressar em palavras o significado do comando. Você deve chegar a algo parecido com: Mando o objeto **out** (a saida padrão) da *classe* **System** (o sistema) executar o seu método **println** (imprimir uma linha) com o argumento **"Olá Mundo!"** (o texto a ser impresso).
- Note que, apesar de estar usando recursos do pacote **java.lang**, você não precisou importar este pacote. É o único pacote que está automaticamente à disposição de qualquer programa

O código fonte:

```
public class OlaAplicativo{
   public static void main( String args[] ){
     System.out.println( "Olá, Mundo!" );
   }
}
```

Applet

- No caso de um applet, você pode inserir no método **init** o comando utilizado no caso anterior. Porém, a saída padrão não é a janela do navegador na qual o arquivo HTML contendo o applet foi carregado. Para um applet **System.out**, assim como **System.err** (o dispositivo padrão para mostrar mensagens de erro) é a ***console Java***, que você pode abrir no menu do navegador e que é utilizada principalmente para informar erros e exceções.
- Consultando a documentação a respeito da classe **Applet** do pacote **java.applet**, você descobre que um applet é um **Panel** (***painel***, classe do pacote **java.awt** que contém as classes úteis para montar uma interface gráfica) sobre o qual você pode afixar componentes gráficas utilizando o método **add**.
- A componente gráfica adequada para meramente mostrar poucas palavras de texto é o **Label** (rótulo). Você precisa, portanto, criar um novo Label, o que é feito usando a palavra chave **new**. Você atribui a este rótulo o texto **"Olá, Mundo!"** e manda o applet afixar o resultado a si mesmo. A sintaxe desta operação toda é então:

```
add( new Label( "Olá, Mundo!" ) );
```

- Há uma terceira maneira de um applet apresentar algumas palavras de texto: usando a ***barra de status*** do navegador. Consultando a documentação sobre a classe **Applet**, você descobre que ela oferece o método **showStatus**. Utilize este método para saudar o mundo na barra de status. Note que esta barra é normalmente utilizada pelo navegador para informar o usuário dos percalços da navegação, de maneira que apoderar-se dela para proferir futilidades não é realmente uma boa idéia. Rodando o exemplo abaixo, você provavelmente não enxergará a mensagem de saudação na barra de status, pois ela é instantaneamente sobre-escrita por uma mensagem indicando que o carregamento da página foi concluído.
- Note que você precisa ***importar os pacotes*** **java.applet** e **java.awt** para ter acesso a todos os recursos utilizados acima.

O código fonte

```
import java.applet.*;
import java.awt.*;
public class OlaApplet extends Applet{
   public void init(){
     System.out.println( "Olá, Mundo!" );
     add( new Label( "Olá, Mundo!" ) );
     showStatus( "Olá, Mundo!" );
   }
}
```

O código HTML

```
<HTML>
<HEAD>
<TITLE>Meu Primeiro Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="OlaApplet.class" WIDTH=200 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

Aplicativo independente gráfico

- Se quisermos um aplicativo gráfico que execute independentemente de um navegador, precisamos criar uma janela, o que pode ser feito utilizando a classe **Frame** do pacote **java.awt**. Uma maneira elegante de proceder é fazer com que a própra classe que estamos construindo seja ela mesma um **Frame**. Isto é feito utilizando a palavra chave extends. A declaração do nossa classe fica então:

```
public class OlaAplGrafico extends Frame
```

- Porém, para ter acesso aos recursos da classe Frame, precisamos ainda criar uma instância desta classe. O procedimento é semelhante àquele que foi utilizado acima para criar o rótulo, com um aspecto adicional: precisamos dar um nome ao nosso objeto para podermos nos referir a ele mais adiante. Declaração e criação podem ser realizadas simultaneamente pelo comando:

```
OlaAplGrafico olaAplicativo = new OlaAplGrafico();
```

- Podemos agora afixar um rótulo ao nosso Frame, como foi feito para o applet. Precisamos especificar que o objeto que deve executar o seu método add é aquele que acabamos de criar. A sintaxe adequada é:

```
olaAplicativo.add( new Label( "Olá, Mundo!" ) );
```

- Para que a janela apareça quando o programa for executado, você precisa definir as suas dimensões e declará-la visível. Procure os métodos adequados na documentação sobre a classe Frame. Verifique que os comandos seguintes são adequados:

```
olaAplicativo.setSize( 200, 100 );
olaAplicativo.setVisible( true );
```

- Você precisa importar o pacote java.awt para ter acesso às classes gráficas.
- Compile e execute o seu programa. Deve aparecer uma pequena janela mostrando a saudação desejada. Note que se você clicar no x no canto superior direito, a janela não desaparece. A resposta adequada do programa ao clique do mouse deve ser implementada pelo programador, como aprenderemos mais adiante neste curso.

O código fonte

```
import java.awt.*;
public class OlaAplGrafico extends Frame{
   public static void main( String args[] ){
     OlaAplGrafico olaAplicativo = new OlaAplGrafico();
     olaAplicativo.add( new Label( "Olá, Mundo!" ) );
     olaAplicativo.setSize( 200, 100 );
     olaAplicativo.setVisible( true );
   }
}
```

Elementos básicos da síntaxe

Comandos

- Comandos em Java são separados por um ***ponto-vírgula***. É permitido colocar vários comandos sobre a mesma linha.
- Comandos são organizados em ***blocos*** definidos por ***chaves*** **{...}**. Em especial, classes e métodos são blocos. Blocos podem conter outros blocos.

Variáveis

- Nomes de variáveis devem começar com uma letra, um caractere de sublinhado _ ou um cifrão $. Não podem começar por um número.
- É convencional (mas não obrigatório) usar uma ***letra minúscula*** para a ***primeira letra*** do nome de uma variável.
- Todas as variáveis de ***tipos primitivos*** precisam ser ***declaradas*** e ***inicializadas***, através de uma instrução da forma

```
<tipo> nomeDaVariahvel = <valor>;
```

onde **<tipo>** representa um dos tipos primitivos (discutidos na próxima seção) e **<valor>** um valor adequado para este tipo. Repare a notação **<...>** utilizada nestas notas para indicar um elemento de código que ***deve*** ser substituído por uma palavra chave, ou um valor concreto. Existem valores "default" para os tipos primitivos.

- As variáveis associadas a ***objetos***, que são ***referências*** aos mesmos, precisam ser ***declaradas***. Os objetos por sua vez precisam ser ***criados***. Declaração e criação podem sem realizadas por comandos separados:

```
NomeDaClasse nomeDoObjeto;
nomeDoObjeto = new NomeDaClasse();
```

A primeira linha informa que a variável **nomeDoObjeto** vai designar um objeto que é uma ***instância*** da classe **NomeDaClasse**. A segunda linha cria um novo objeto pertencente àquela classe e atribui a este objeto a referência **nomeDoObjeto**. A parte **NomeDaClasse()** do comando é na verdade uma chamada ao ***método construtor*** da classe. Em muitos casos, este método pode receber parâmetros que são incluídos nos parênteses e servem para especificar a ***inicialização*** do objeto. As duas linhas acima podem ser combinadas numa só:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

- O ***escopo*** de uma variável, ou seja a região do programa na qual ela está definida, é limitado ao bloco no qual ela foi declarada.

Comentários

- Um comentário curto relativo a uma linha de código pode ser incluído no fim da linha, precedido de **//** :

```
... ; // Comentário sobre o comando ...
```

- Um comentário precedido de **/*** e seguido de ***/** pode ocupar várias linhas:

```
/* Comentário maior,
que pode ocupar várias linhas. */
```

Obviamente, isto serve também para forçar o compilador a ignorar temporariamente um trecho de código.

- O JDK oferece um recurso para gerar automaticamente um ***arquivo HTML*** documentando uma classe. Comentários que forem precedidos de **/**** e seguidos de ***/** serão incluídos neste arquivo:

```
/** Este comentário será incluído no arquivo HTML
documentando a classe. */
```

Para gerar a documentação relativa à(s) classe(s) cuja(s) fonte(s) estão no arquivo MeuPrograma.java, digita-se na linha de comando:

```
javadoc MeuPrograma.java
```

Tipos de dados primitivos

Números inteiros

- Há quatro tipos de inteiros em Java:

Tipo	Tamanho	Valor
byte	8 bits	-128 a 127
short	16 bits	-32.768 a 32.767
int	32 bits	-2.147.483.648 a 2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

- Um número inteiro pode sempre ser atribuído a outro de maior precisão:

```
int a = 274;
long b = a;
```

A operação inversa requer coerção explicita ("casting"):

```
long a = 274;
int b = ( int ) a;
```

Sem esta coerção, haverá erro de compilação. Note que, mesmo com a coerção, ainda pode haver problema na execução, caso o valor do número atribuído ultrapassar o maior valor possível para o tipo em questão.

Números em ponto flutuante

- Há dois tipos, de precisão simples e de precisão dupla:

Tipo	Tamanho	Valor
float	32 bits	-3.40292347E+38 a +3.40292347E+38
double	64 bits	-1.79769313486231570E+308 a +1.79769313486231570E+308

- Um número de precisão simples pode sempre ser atribuído a outro de precisão dupla:

```
float a = 2.74F; // F (ou f) após o literal indica precisão simples
double b = a;
```

A operação inversa requer coerção explicita:

```
double a = 2.74e12;
float b = ( float ) a;
```

Sem esta coerção, haverá erro de compilação. De novo, mesmo com a coerção, ainda pode haver problema na execução, caso o valor do número de precisão dupla ultrapassar o maior valor possível para um número de precisão simples.

- Note que Java não possui tipos primitivos representando números complexos. Não há recursos para lidar com números complexos nas bibliotecas fornecidas com o JDK, mas vários pacotes avulsos são disponíveis. Não sendo tipos primitivos, números complexos devem ser objetos em Java.

Caracteres

- Há um tipo primitivo que representa um caractere:

Tipo	Tamanho	Valor
char	16 bits	'\u0000' a '\uFFFF'

- Java utiliza o padrão de caracteres Unicode, que abrange os conjuntos de caracteres de muitas línguas.
- Um literal de caractere é especificado por um único caractere entre apóstrofes simples:

```
char exemplo = 'a';
```

- Para caracteres de escape, usa-se a barra invertida:

Escape	Significado
\n	nova linha
\t	tabulação
\b	passo para trás
\r	retorno do carro
\\	barra invertida
\'	apóstrofe
\"	aspas

Booleanos

- São variáveis lógicas que podem assumir os valores verdadeiro e falso:

Tipo	Tamanho	Valor
boolean	1 bit	true ou false

- Note que, em Java, estas variáveis não podem ser interpretadas como os números inteiros 0 e 1.

Expressões e operadores

Uma expressão é uma instrução de realizar uma operação que produz um valor (valor de retorno). Operadores são símbolos especiais utilizados para operações matemáticas, atribuições, comparações e operações lógicas.

Operadores aritméticos

- Os operadores aritméticos disponíveis em Java são:

Operador	Significado
+	adição
-	subtração
*	multiplicação
/	divisão
%	resto da divisão (módulo)

- Note que Java não possui um operador específico para potência (tal como ** em FORTRAN). Para calcular uma potência, você deve usar o método pow da classe Math do pacote lang, ou seja, para calcular x^y, você escreve Math.pow(x, y), onde x e y são de tipo double, e o resultado também.

Operadores de atribuição

- Estes operadores são simplesmente uma notação compacta para uma operação aritmética seguida da atribuição do valor de retorno à variável que continha o primeiro termo da operação.

Operador	Exemplo	Expressão equivalente
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Operadores de incremento e decremento

- São operadores que atuam sobre uma única variável numérica, aumentando ou diminuindo o seu valor de uma unidade:

Operador	Exemplo	Significado
++	++a	adicionar 1 à variável a e depois calcular a expressão na qual a reside
	a++	calcular a expressão na qual a reside e depois adicionar 1 à variável a
--	--a	subtrair 1 da variável a e depois calcular a expressão na qual a reside
	a--	calcular a expressão na qual a reside e depois subtrair 1 da variável a

- Exemplo:

```
int x = 5, y = 7; // x vale 5 e y vale 7
y += ++x; // x agora vale 5 + 1 = 6 e y vale 7 + 6 = 13
x -= y--; // x agora vale 6 - 13 = - 7 e y vale 13 - 1 = 12
```

Operadores de comparação

- Estes operadores atuam sobre valores numéricos e retornam valores booleanos, true (verdadeiro) ou false (falso):

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

- O operador == também serve para comparar outros tipos de dados, inclusive objetos.

Operadores lógicos

- Estes operadores atuam sobre valores booleanos e retornam valores booleanos, true ou false.

Operador	Significado	Exemplo	Explicação
&&	E ("logical AND")	a && b	retorna true se a e b forem ambos true. Senão retorna false. Se a for false, b não é avaliada.
&	E ("boolean logical AND")	a & b	retorna true se a e b forem ambos true. Senão retorna false. Ambas expressões a e b são sempre avaliadas.
||	OU ("logical OR")	a || b	retorna true se a ou b for true. Senão retorna false. Se a for true, b não é avaliada.
|	OU ("boolean logical inclusive OR")	a | b	retorna true se a ou b for true. Senão retorna false. Ambas expressões a e b são sempre avaliadas.
^	OU EXCLUSIVO ("boolean logical exclusive OR")	a ^ b	retorna true se a for true e b for false ou vice-versa. Senão retorna false
!	NÃO ("logical NOT")	!a	retorna true se a for false. Senão retorna false

Precedência e associatividade dos operadores

- Os operadores estão listados na tabela abaixo em ordem decrescente de prioridade.

Operador	Associatividade
()	da esquerda para a direita
++ -- + - !	unários; da direita para a esquerda
* / %	da esquerda para a direita
+ -	da esquerda para a direita
< <= > >=	da esquerda para a direita
== !=	da esquerda para a direita
&	da esquerda para a direita
^	da esquerda para a direita
|	da esquerda para a direita
&&	da esquerda para a direita
||	da esquerda para a direita
= += -= *= /= %=	da direita para a esquerda

Estruturas de controle de fluxo

Estruturas de seleção

- Para executar um comando sujeita a uma condição:

```
if( <condição> ) <comando>;
```

A variável ou expressão condição deve ser do tipo boolean. O comando é executado se condição possuir o valor true.

- Para executar vários comandos sujeitos a uma condição, coloca-se os comandos num bloco:

```
if( <condição> ){
  <comando 1>;
  <comando 2>;
  <...
}
```

- Para executar certos comandos caso uma condição for satisfeita, e outros comandos caso contrário:

```
if( <condição> ){
  <comando 1>;
  <comando 2>;
}
else{
  <comando 3>;
  <comando 4>;
}
```

No este exemplo, comando 1 e comando 2 são executados se condição for true. Caso contrário, comando 3 e comando 4 são executados.

- O operador ?: pode ser utilizado em certos casos simples como um atalho para a estrutura if...else:

```
<variável> = <condição> ? <valor 1> : <valor 2>;
```

Aqui, <variável> recebe <valor 1> se <condição> for true e <valor 2> se <condição> for false.

- Escolhas múltiplas de comandos a serem executados, dependendo do valor de uma variável, podem ser implementadas da seguinte maneira:

```
switch( <controle> ){
  case <valor 1>:
    <comando 1>;
    <comando 2>;
    break;
  case <valor 2>: case <valor 3>:
    <comando 3>;
    <comando 4>;
    break;
  case <valor 4>:
    ...
    break;
  default:
    <comando 5>;
}
```

O <controle> deve ser de tipo int ou char. Se ele assume o <valor 1>, <comando 1> e <comando 2> são executados. O comando break determina então a saida do bloco. Se o <controle> assume o <valor 2> ou o <valor 3>, <comando 3> e <comando 4> são executados. O comando break de novo determina a saida do bloco, etc. O <comando 5> é executado se o valor do <controle> é diferente de todos os valores listados.

Estruturas de repetição (laços)

- Para repetir um conjunto de instruções, controlando o número de repetições com um contador:

```
for( int i = <iinicial>; i <= <ifinal>; i++ ){
  <comando 1>;
  <comando 2>;
}
```

Note a declaração int i do contador. O teste i <= <ifinal> é realizado antes de cada entrada no laço. O incremento i++ é realizado no fim de cada passagem pelo laço.

- Para repetir um conjunto de instruções enquanto uma certa condição for satisfeita:

```
while( <condição> ){
  <comando 1>;
  <comando 2>;
}
```

Obviamente a <condição> deve ser do tipo boolean. Uma forma um pouco diferente é:

```
do{
  <comando 1>;
  <comando 2>;
}
while( <condição> )
```

A diferença é que a <condição> é testada após a execução do bloco de instruções. Portanto, este bloco é executado no mínimo uma vez.

- Para interromper um laço, usa-se o comando break:

```
while( <condição 1> ){
  <comando 1>;
  if( <condição 2> ) break;
  <comando 2>;
}
```

O laço é interrompido quando a <condição 2> assume o valor true.

- Para passar à iteração seguinte de um laço, usa-se o comando continue:

```
for( int i = <inicial>; i <= <ifinal>; i++ ){
  <comando 1>;
  if( <condição 2> ) continue;
  <comando 2>;
<}
```

Se <condição 2> for true, o programa "pula por cima" do <comando 2> e passa à iteração seguinte.

- Os comandos break e continue sem rótulos atuam sobre o laço mais interno no qual eles se encontram. No caso de laços aninhados, comandos rotulados podem ser utilizados para saltar para um laço mais externo:

```
rohtulo:{
  while( <condição 1> ){
    <comando 1>;
    for( int i = <iinicial>; i <= <ifinal>; i++ ){
      if( <condição 2> ) break rohtulo;
      <comando 2>;
    }
  }
  <comando 3>;
}
```

Se <condição 2> for true, o programa sai do bloco rotulado, pulando por cima de <comando 2> e <comando 3>.

```
rohtulo:{
  for( int k = <kinicial>; k <= <kfinal>; k++ ){
    <comando 1>;
    for( int i = <iinicial>; i <= <ifinal>; i++ ){
      if( <condição 2> ) continue rohtulo;
      <comando 2>;
    }
  }
}
```

Se <condição 2> for true, o programa sai de ambos os laços.

Prática

Laço

Programe um applet que lista na tela as fatoriais dos números inteiros n de 0 até 10. Para tanto, no método init:

- Faça um laço sobre o número inteiro n.
- Escreva o resultado na forma de um rótulo (Label) para cada valor de n. Note que você pode utilizar o operador + para concatenar cadeias de caracteres e números, que serão automaticamente expressos em caracteres. Por exemplo, se m=n!, você pode formar o argumento do Label como segue:

```
"Fatorial de " + n + " = " + m,
```

onde n e m são do tipo int.

Execução condicional

Programe um applet que lista na tela o número de dias em cada um dos 12 meses do ano (ignorando anos bissextos). Para tanto:

- Faça um laço sobre uma variável inteira associada ao mês (variada de 1 até 10).
- Utilize uma estrutura switch...case para atribuir a cada mês o número de dias adequado.
- Escreva o resultado na forma de um rótulo (Label) para cada mês.

Cadeias de caracteres

Construção

- Cadeias de caracteres são objetos, instâncias da classe String do pacote java.lang. Para declarar uma cadeia de caracteres e atribuir à mesma um valor literal, coloca-se este valor entre aspas:

```
String s = "Olá!";
```

- Pode-se também criar explicitamente um novo objeto, chamando-se o método construtor da classe:

```
String s = new String( "Olá!" );
```

  O método construtor pode receber outros tipos de argumentos, por exemplo, uma lista dos caracteres que vão constituir a cadeia. Veja os detalhes na documentação.

- Uma vez a cadeia criada, o seu conteúdo não pode mudar. Se precisarmos de uma cadeia de caracteres cujo conteúdo possa mudar, precisamos usar a classe StringBuffer do pacote java.lang, que não será discutida nestas notas.
- Obviamente, uma dada referência do tipo String não precisa apontar sempre para o mesmo objeto. Assim, se tivermos

```
String s = new String( "Olá!" );
s = "Olá Mundo!";
```

  a referência s

  deixa de referir-se ao primeiro objeto criado (cujo conteúdo será "Olá!") e passa a referir-se ao objeto cujo conteúdo é "Olá Mundo!"). Se não houver mais nenhuma referência apontando para o primeiro objeto, este estará marcado para ser varrido pelo coletor de lixo tão logo este resolver entrar em ação.

Comprimento

- O método length permite descobrir o número de caracteres contidos numa String

  . Por exemplo:
  String s = new String("Olá!");
  int nuhmero = s.length();
  resulta no valor 4 para o inteiro chamado nuhmero.

- Há outros métodos que permitem obter informação a respeito de uma String, por exemplo, descobrir qual o caractere numa dada posição. Veja a documentação.

Comparação

- Para determinar se duas cadeias de caracteres possuem o mesmo conteúdo, usa-se o método equals. Este método considera maiúsculas e minúsculas como diferentes. Se não desejamos fazer esta distinção, devemos usar o método equalsIgnoreCase.
- É importante não confundir estas comparações de conteúdo com o uso do operador ==, que, quando usado entre duas variáveis que são referências a objetos, serve para determinar se as duas variáveis apontam para o mesmo objeto.
- Como ilustração, considere o seguinte trecho de código:

```
String s1, s2, s3, s4, s5, s6;
s1 = "Olá!";
s2 = "Olá!";
s3 = "olá!":
s4 = new String( "Olá!" );
s5 = s4;
s6 = new String( "Olá!" );
boolean b1, b2, b3, b4, b5, b6, b7, b8;
b1 = s2.equals( s1 );
b2 = s3.equals( s1 );
b3 = s3.equalsIgnoreCase( s1 );
b4 = s4.equals( s1 );
b5 = ( s2 == s1 );
b6 = ( s4 == s1 );
b7 = ( s4 == s5 );
b8 = ( s4 == s6 );
b9 = s4.equals( s6 );
```

Convença-se de que, após execução deste código, os seguintes booleanos possuem o valor true: b1, b3, b4, b5, b7 e b9. Já b2, b6 e b8 possuem o valor false.

- Consulte a documentação para obter informação sobre métodos que permitem comparar pedaços de cadeias de caracteres e também determinar qual de duas cadeias vem primeira na ordem alfabética.

Concatenação

- Para concatenar duas String, usa-se o método concat. Note que, já que o tamanho de uma String não pode mudar, a concatenação resulta na criação de um novo objeto.
- Também pode-se usar o operador + para a concatenação. Por exemplo, no trecho de código abaixo, a mesma linha é impressa duas vezes:

```
String s1 = "Olá ";
String s2 = "Mundo!";
System.out.println( s1.concat( s2 ) );
System.out.println( s1 + s2 );
```

- Consulte a documentação para obter informação sobre métodos que permitem modificar cadeias de caracteres, por exemplo alterando um caractere ou passando de minúsculas para maiúsculas. Lembre-se de que, na verdade, estes métodos criam uma nova cadeia.

Conversão de dados numéricos

- O método estático valueOf da classe String permite obter uma cadeia de caractere que representa um dado tipo numérico. Um método estático está associado à classe, e não a uma instância particular da mesma. É fácil entender que o método em questão deve ser estático, pois quando resolvemos expressar um número na forma de caracteres, ainda não temos uma String. O método a ser chamado é que vai criá-la. Por exemplo:

```
double a = 3.1415927;
int i = 123;
System.out.println( "a = " + String.valueOf( a ) );
System.out.println( "i = " + String.valueOf( i ) );
```

Este código resulta na impressão de a = 3.1415927 seguido de i = 123 na saída padrão. Na verdade, o mesmo resultado pode ser obtido com uma sintaxe simplificada, pois numa concatenação, números

são transformados automaticamente em caracteres. Assim, as duas últimas linhas acima podem ser substituídas por

```
System.out.println( "a = " + a );
System.out.println( "i = " + i );
```

- Para realizar a operação inversa, ou seja transformar caracteres em tipos numéricos, é de se esperar que precisamos utilizar um método estático associado ao tipo numérico em questão. Porém, como um tipo primitivo em si não é um objeto, precisamos primeiro descobrir como associar uma classe a um tipo primitivo. As classes adequadas são chamadas de classes de embrulhamento (("wrapper classes") e podem ser encontradas no pacote java.lang. Por exemplo, a classe de embrulhamento de um tipo double é Double. Portanto, precisamos primeiro traduzir a nossa cadeia de caracterees num objeto Double. Para tanto, chamamos o método estático valueOf desta classe. Ainda precisaremos "desembrulhar" o nosso número, ou seja traduzir o nosso objeto Double num tipo primitivo double. Isto é feito chamando o método doubleValue da classe Double. O trecho de código abaixo ilustra este procedimento, assim como o semelhante para o caso de um tipo inteiro. (Outros tipos podem ser tratados também da mesma maneira.)

```
String atxt = "3.1415927";
String itxt = "123";
double a = Double.valueOf( atxt ).doubleValue();
int i = Integer.valueOf( itxt ).intValue();
```

Após execução, o valor da variável de precisão dupla a é 3.1415927 e o valor da variável inteira i é 123. Num exemplo mais realístico, os valores expressos na forma de caracteres poderiam ser lidos de um arquivo ou digitados pelo usuário.

- No caso de inteiros, há uma maneira mais simples de realizar a operação acima: o método parseInt da classe Integer transforma diretamente uma cadeia de caracteres num tipo int. Portanto, o comando

```
int i = Integer.parseInt( itxt );
```

pode ser usado em vez da última linha do trecho de código anterior.

Formatação

- A transformação de números em caracteres descrita na seção anterior resulta numa representação completa do número, sem arredondamento nem controle do formato pelo programador. Recursos para formatar texto, e em especial números, podem ser encontrados no pacote java.text. Como é de se esperar, formatos são objetos em Java. Como ilustração, o trecho de código abaixo resulta na impressão de a = 3.1416 na saida padrão:

```
double a = 3.1415927;
DecimaFormat meuFormato = new DecimalFormat( "0.####" );
System.out.println( "a = " + meuFormato.format( a ) );
```

O formato desejado é especificado pelo argumento do método construtor (veja os detalhes na documentação). O método format transforma o número (neste caso, de precisão dupla) numa cadeia de caracteres, respeitando este formato.

Decomposição em palavras

- Se uma cadeia de caracteres representa uma sucessão de palavras chaves, pode ser necessário decompô-la em palavras individuais. Semelhantemente, se dados numéricos foram armazenados

num arquivo na forma de uma tabela com várias colunas e estamos lendo linha por linha, precisamos decompor cada linha em números individuais. A classe StringTokenizer do pacote java.util permite realizar este tipo de operação. Cria-se um objeto StringTokenizer associado à cadeia de caracteres a ser decomposta. O método countTokens conta o número de palavras (ou outros elementos) da cadeia. As palavras podem ser extraídas sucessivamente com o método nextToken. O método hasMoreTokens testa a presença de palavras ainda não separadas. Por exemplo:

```
String linhaDeDados = "0.225 4.32 6.17";
StringTokenizer separador = new StringTokenizer( linhaDeDados );
int i = separador.countTokens();
String n1 = separador.nextToken();
String n2 = separador.nextToken();
boolean b = separador.hasMoreTokens();
```

Após execução, o inteiro i vale 3; as cadeias de caracteres n1 e n2 contém "0.225" e "4.32", respectivamente. O booleano b possui o valor true, pois há ainda um número que não foi separado.

Listas

Construção

- Listas ("Arrays") são utilizadas para agrupar variáveis do mesmo tipo. O tipo pode ser qualquer tipo primitivo ou qualquer classe de objetos. Para declarar que uma variável representa uma lista, acrescenta-se colchetes após o nome da variável. A lista é criada utilizando-se a palavra-chave new e indicando o número de elementos entre colchetes:

```
int c[];
c = new int[ 12 ];
double d[] = new double[ 30 ];
```

- No caso de uma lista de tipos primitivos, a lista pode ser criada e inicializada através de uma lista de valores entre chaves:

```
int b[] = { 3, 5, 7, 9 };
```

- O índice especificando um elemento de uma lista é contado a partir de 0. Por exemplo, o comando acima é equivalente ao seguinte:

```
int b[] = new int[ 4 ];
b[ 0 ] = 3;
b[ 1 ] = 5;
b[ 2 ] = 7;
b[ 3 ] = 9;
```

- No caso de uma lista de objetos, deve-se notar que a criação da lista não cria os objetos. Cria simplesmente uma lista de referências que ainda não apontam para nada (null). Os objetos ainda precisam ser criados e as suas referências atribuídas aos elementos da lista. Por exemplo, o código abaixo cria 3 rótulos organizados numa lista:

```
Label llb[] = new Label[ 3 ];
llb[ 0 ] = new Label( "Vetor a" );
llb[ 1 ] = new Label( "Vetor b" );
llb[ 2 ] = new Label( "Produto" );
```

Comprimento

- Qualquer lista possui uma variável inteira length que fornece o número de elementos da lista. Como exemplo de uso desta variável, podemos acrescentar ao trecho de código acima um laço que faz com que a cor de fundo dos três rótulos seja azul:

```
for( int i = 0; i < llb.length; i++){
  llb[ i ].setBackground( Color.blue );
}
```

Referência e valor

- Uma lista pode ser passada como argumento a um método. Deve-se notar que listas são objetos e, na linguagem Java, objetos são sempre passados aos métodos por referência, ao passo que tipos primitivos são sempre passados por valor. Isto quer dizer que uma modificação à lista realizada no método chamado implicará na mesma modificação à lista definida no programa que chamou o método em questão.

Índices múltiplos

- Listas de múltiplos índices podem ser definidas, sendo consideradas como listas de listas. Os seguintes exemplos criam objetos que são essencialmente matrizes:

```
int b[][];
b = new int[ 2 ][ 3 ];
int c[][] = { { 1, 2, 3 }, { 4, 5, 6} };
```

- Porém, listas de listas podem ser mais gerais de que matrizes, pois as "linhas" podem ter tamanhos diferentes:

```
int b[][];
b = int[ 2 ][ ];
b[ 0 ] = new int[ 2 ];
b[ 1 ] = new int[ 3 ];
int c[][] = { { 1, 2 }, { 3, 4, 5} };
```

Prática

Produto escalar de dois vetores

Para começar, escreva um método que calcula o produto escalar de dois vetores. Este método será incorporado ao aplicativo independente e aos applets desenvolvidos a seguir. Dicas:

- O método deve receber os dois vetores como argumentos. Os vetores devem ser listas de 3 elementos de tipo **double**.
- O método deve retornar o produto escalar, que é uma variável de tipo **double** também.

Aplicativo independente

A seguir, desenvolva um aplicativo independente que calcule o produto escalar de dois vetores cujas componentes são digitadas como argumentos no comando de execução do aplicativo. Dicas:

- Os argumentos são passados ao aplicativo na forma de uma lista de cadeias de caracteres (**String**), lista esta que é recebida como argumento do método **main** do aplicativo. Esta lista era chamada **args[]** no exemplo desenvolvido na 1ª Prática.
- Você precisa transformar as cadeias de caracteres em números de precisão dupla e organizar estes números em duas listas (os vetores).
- Os passos anteriores são realizados no método **main**. Se você está desenvolvendo um aplicativo gráfico, a sua classe deve estender a classe **Frame** e você deve criar no método **main** uma instância da sua classe, como visto na 1º Prática. O aspecto novo é que, para que a sua classe possa manipular os vetores, estes devem ser passados como argumentos do construtor da mesma. Ou seja, se você resolve chamar a sua classe de **ProdutoEscalarFrame**, você terá no método **main** o comando de criação

```
new ProdutoEscalarFrame( a, b );
```

  e a assinatura do seu método construtor será

```
public ProdutoEscalarFrame( double a[], double b[] );
```

- Dentro do método construtor, você colocará uma chamada ao método que calcula o produto escalar, método este que deve ser incluido na classe.
- Você precisa montar uma interface gráfica para apresentar os dados e o resultado. Uma possibilidade seria algo do tipo:

Vetor a	**Vetor b**
1.	4.
2.	5.
3.	6.
Produto:	**32.**

- Cada elemento desta tabela seria um rótulo (**Label**). Para organizar os elementos em tabela, você precisa utilizar um ***gerenciador de disposição*** (**LayoutManager**). Para uma organização em grade, o gerenciador adequado seria **GridLayout** (uma classe do pacote **java.awt**). Para atribuir este gerenciador ao seu **Frame**, você coloca no método construtor o comando

```
setLayout( new GridLayout( 5, 2 ) );
```

  onde os argumentos são o número de linhas e o número de colunas desejados. Feito isto, você afixa os rótulos sucessivamente, linha após linha, usando o método add.

- Note que para escrever as componentes dos vetores e o produto escalar nos rótulos, você precisa transformar os números em cadeias de caracterees, como visto nesta aula.
- Se tudo der certo, ao digitar na linha de comando

```
java ProdutoEscalarFrame 1 2 3 4 5 6
```

  você deve obter uma tabela parecida com aquela mostrada acima.

Applet

Faremos agora algo semelhante com um applet. As diferenças com o aplicativo são as seguintes.

- Você não precisa criar o applet, ele é criado automaticamente pelo navegador. Você pode fazer no método **init** aquilo que foi feito acima no método construtor.
- As componentes dos vetores podem ser fornecidas como parâmetros no arquivo HTML da seguinte maneira:

```
<HTML>
<HEAD>
<TITLE>Produto escalar de vetores</TITLE>
</HEAD>
<BODY>
<APPLET CODE="ProdutoEscalarApplet.class" WIDTH=400 HEIGHT=400>
<PARAM NAME="a1" VALUE="1.">
<PARAM NAME="a2" VALUE="2.">
...
<PARAM NAME="b1" VALUE="4.">
...
</APPLET>
</BODY>
</HTML>
```

- Para ler estes parâmetros, usa-se o método **getParameter** da classe **Applet**. No caso do exemplo acima, a chamada **getParameter("a1")** retorna a **String "1."**. Note que o nome do parâmetro é passado como argumento e o valor do parâmetro é fornecido pelo método, na forma de uma **String**. Ainda é necessário transformar esta num tipo numérico.

Objetos e classes

Classes

- Uma classe é um modelo usado para definir vários objetos com características semelhantes. Um programa é constituído de uma classe ou de um conjunto de classes. Os elementos básicos de uma classe são chamados membros da classe e podem ser divididos em duas categorias:
    - As variáveis, que especificam o estado da classe ou de um objeto instância desta classe.
    - Os métodos, que especificam os mecanismos pelos quais a classe ou um objeto instância desta classe podem operar.
- O esqueleto de uma classe apresenta-se da seguinte maneira:

```
class NomeDaClasse{
   ...
   TipoDaVariahvel1 variahvel1;
   TipoDaVariahvel2 variahvel2;
   ...

   TipoDeRetorno1 mehtodo1(){
     ...
   }

   TipoDeRetorno2 mehtodo2(){
     ...
   }
   ...
}
```

- É claro que, além das variáveis mencionadas acima, que estão definidas fora de qualquer método, haverá em geral também variáveis definidas dentro de um determinado método e cujo escopo será limitado a este método, ou possivelmente a um sub-bloco deste método. Estas variáveis são chamadas locais. Em contradistinção, as variáveis das quais tratamos aqui são chamadas globais. Veja adiante nesta apostila uma discussão um pouco mais completa desta distinção.
- Para que uma instância de uma classe possa ser criada por qualquer outra classe, a classe em questão deve ser declarada pública, o que é feito acrescentando-se a palavra chave public à declaração da classe:

```
public class NomeDaClasse{
   ...
}
```

Classes que constituem aplicativos independentes e applets devem ser públicas. Uma classe que não for declarada pública somente poderá ser acessada por outras classes do mesmo pacote. Deve-se notar que cada arquivo fonte pode conter somente uma classe pública. Caso o arquivo contenha uma classe pública, ele deve possuir o mesmo nome que esta classe, com a terminação .java.

Objetos

- Um objeto é uma instância de uma classe, ou seja uma realização concreta e particular da mesma. Um objeto precisa ser criado. Para que seja possível acessar as variáveis e os métodos de um objeto, é preciso atribuir uma referência ao objeto. O tipo de uma referência, ou seja, a classe à qual pertence o objeto ao qual ela vai referir-se precisa ser declarada.

- Declaração: a seguinte instrução declara que a variável nomeDoObjeto refere-se a um objeto instância da classe NomeDaClasse:

```
NomeDaClasse nomeDoObjeto;
```

- Criação: a seguinte instrução cria (em memória) um novo objeto instância da classe NomeDaClasse, que será referenciado pela variável nomeDoObjeto previamente declarada:

```
nomeDoObjeto = new NomeDaClasse();
```

- As duas instruções acima podem ser combinadas numa só:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

- Vale notar que a atribuição de uma referência a um objeto não é obrigatória. Há casos em que basta criar o objeto e passá-lo como argumento de um método, com um comando do tipo

```
algumMehtodo( new AlgumaClasse() );
```

Variáveis

Variáveis de instância e variáveis de classe

- Uma variável de instância é uma variável cujo valor é específico ao objeto e não à classe. Uma variável de instância em geral possui um valor diferente em cada objeto representante da classe.
- Uma variável de classe é uma variável cujo valor é comum a todos os objetos representantes da classe. Mudar o valor de uma variável de classe em um objeto automaticamente muda o valor para todos os objetos instâncias da mesma classe. Um exemplo óbvio de uma variável de classe seria o número de instâncias desta classe que já foram criadas.
- Uma variável é considerada como de instância por "default". Para declarar uma variável de classe, acrescenta-se a palavra-chave static. Aliás, outra expressão utilizada para indicar uma variável de classe é variável estática. Exemplo:

```
static int nuhmeroDeInstahnciasDestaClasse;
```

Restrições de acesso

- Algumas palavras-chaves são diponíveis para ampliar ou restringir o acesso a uma variável. Estas palavras-chaves são acrescentadas à declaração da variável.
- Uma variável que pode será acessada por qualquer outra classe é dita pública, e é declarada usando-se a palavra-chave public.
- Uma variável que pode será acessada somente por métodos da própria classe é dita privada, e é declarada usando-se a palavra-chave private.
- Uma variável que, além de poder será acessada por métodos da própria classe, também pode ser acessada pelas subclasses da classe na qual ela é declarada, é dita protegida e é declarada usando-se a palavra-chave protected. [O conceito de subclasse será desenvolvido no próximo capítulo.]
- Uma variável para a qual não foi especificada nenhuma destas palavras-chaves é dita amigável e pode ser acessada por todas as classes que pertencem ao mesmo pacote. Pacotes são agrupamentos de classes. Como já sabemos, as classes de biblioteca da SUN estão organizadas em pacotes. O programador também pode criar os seus próprios pacotes.
- Os vários tipos de declaração de acesso estão exemplificados abaixo:

```
/* a classe definida neste arquivo pertence ao pacote chamado algumPacote */
package algumPacote;
public class nomeDaClasse{
/* a variável w é acessível por qualquer classe *
  public int w;
/* a variável x é acessível pelos métodos da classe nomeDaClasse e das suas
subclasses */
  protected int x;
/* a variável y é acessível pelos métodos da classe nomeDaClasse e das outras
classes que pertencem ao pacote chamado algumPacote */
  int y;
/* a variável z é acessível somente pelos métodos da classe nomeDaClasse */
  private int z;
   ...
}
```

- As restrições de acesso desempenham um papel fundamental na programação orientada a objeto. Embora possa parecer mais simples e simpático permitir a qualquer objeto o acesso a todas as variáveis de qualquer outro objeto, isto resultaria em código muito pouco robusto. Recomenda-se, pelo contrário, limitar o acesso a qualquer variável o quanto for possível, dada a função da variável em questão. Mesmo no caso de variáveis que precisam será acessíveis a todos os objetos, há uma alternativa preferível ao acesso irrestrito outorgado pela palavra chave public, como veremos na próxima seção.

Como um objeto acessa as variáveis de outro objeto

- Supondo que as restrições de acesso discutidas acima o permitam, uma classe pode utilizar ou modificar uma variável pertencente a um objeto através do operador ponto.
- Por exemplo, após declarar e criar o objeto nomeDoObjeto, representante da classe nomeDaclasse, podemos ir buscar a variável variahvel deste objeto e atribui-la à variável var da classe que estamos desenvolvendo:

```
TipoDaVariahvel var;
var = nomeDoObjeto.variahvel;
```

Evidentemente, var e variahvel devem ser do mesmo tipo (chamado TipoDaVariahvel no exemplo acima).

- também podemos atribuir à variável variahvel do objeto nomeDoObjeto o valor de uma variável var da nossa classe [entende-se aqui "valor" no sentido geral, podendo ser uma referência a um objeto]:

```
TipoDaVariahvel var = algumValor;
nomeDoObjeto.variahvel = var;
```

- Se variahvel for uma referência a um outro objeto, pode-se concatenar operadores pontos para alcançar variáveis deste objeto:

```
TipoDaVariahvel var;
var = nomeDoObjeto.variahvel.outraVariahvel;
```

Neste caso, é claro, var e outraVariahvel devem ser do mesmo tipo.

- Se a variável a ser acessada for uma variável de classe, pode-se usar a classe, em vez de uma instância particular, para acessar a variável:

```
TipoDaVariahvel var;
var = nomeDaClasse.variahvel;
```

- Embora esta seja a maneira mais direta de permitir que um objeto tenha acesso às variáveis de outro objeto, não é a mais segura, pois permite que um objeto modifique uma variável do outro sem restrição. Uma maneira mais segura consiste em declarar a variável como privada, mas incluir na classe métodos especiais controlando o acesso à variável:

```
public class nomeDaClasse{
  private TipoDaVariahvel variahvel;
   ...
  public TipoDaVariahvel getVariahvel(){
     return variahvel;
  }
  public void setVariahvel( TipoDaVariahvel valor ){
/* aqui pode-se colocar testes para determinar se o valor é aceitável */
    ...
    variahvel = valor;
  }
}
```

Assim, pode-se colocar no método setVariahvel código para conferir que o valor passado como argumento é adequado. Omitindo o método setVariahvel, impedimos qua a variável seja modificada por qualquer outro objeto, embora ela possa ser lida chamando o método getVariahvel.
O operador ponto também serve para chamar os métodos de outro objeto (veja adiante nessa apostila), de maneira que a classe que deseja manipular as variáveis do outro objeto conterá agora comandos do tipo:

```
  TipoDaVariahvel var, varp;
  var = nomeDoObjeto.getVariahvel();
  ...
  nomeDoObjeto.setVariahvel( varp );
```

Esta técnica de programação é chamada encapsulação de variáveis.

Variáveis globais e variáveis locais

- As variáveis discutidas acima são declaradas fora de qualquer método (usualmente no cabeçalho da classe) e são acessíveis por qualquer método da classe. Tais variáveis são chamadas globais. Muitas vezes, variáveis auxiliares são declaradas dentro de um determinado método, ou até dentro de um bloco menor. Tais variáveis são chamadas locais. Elas existem somente durante a execução daquele método ou bloco. A parte de código que "enxerga" uma determinada variável é chamada o escopo da variável. Assim, o escopo de uma variável global é a classe inteira, e o escopo de uma variável local é o método, ou um bloco contido dentro do método, ao qual ela pertence.

- Exemplo:

```
class nomeDaClasse{
/* a variável abaixo é global */
   TipoDaVariahvel variahvel1;
   ...
   TipoDeRetornão nomeDoMehtodo()
   {
/* a variável abaixo é local; está definida somente dentro deste método */
     TipoDaVariahvel variahvel2;
     for( int i = 0; i < 10; i++ ){
/* a variável i é local, definida só dentro deste bloco */
       ... //
     }
   }
}
```

- Embora não pareça uma boa ideia, é possível dar a uma variável local um nome que já foi atribuído a uma variável global. Neste caso, a variável local "encobre" a variável global, mas o acesso à variável

global é possível usando a palavra-chave this (que fornece uma referência ao próprio objeto) e o operador ponto:

```
class nomeDaClasse{
   TipoDaVariahvel variahvel; // variável global
   ...
   TipoDeRetornão nomeDoMehtodo()
   {
     TipoDaVariahvel variahvel; // variável local
   ...
/* a variável abaixo recebe o valor de variahvel local */
     variahvel2 = variahvel;
/* a variável abaixo recebe o valor de variahvel global */
     variahvel3 = this.variahvel;
   }
}
Alguns programadores fazem uso desta construção em métodos set:
public class nomeDaClasse{
   private TipoDaVariahvel variahvel;
   public void setVariahvel( TipoDaVariahvel variahvel )
   {
     this.variahvel = variahvel;
   }
}
```

Variáveis constantes

- Esta expressão um tanto paradoxal refere-se a variáveis cujo valor não pode mudar durante a execução do programa. Tais variáveis são caracterizadas pela palavra-chave final, e por convenção recebem usualmente nomes escritos inteiramente em maiúsculas.

- Por exemplo, na classe Math do pacote lang, encontramos

```
public static final double PI;
```

que contem o valor da famosa constante matemática.

Métodos

Valor de retorno e argumentos

- Em geral, um método recebe argumentos cujos valores lhe são passados pelo objeto que o chamou, efetua um conjunto de operações e retorna algum resultado. A declaração do método especifica o nome do método, o tipo de retorno, o nome e o tipo de cada argumento. Os argumentos são variáveis locais do método em questão. O valor de retorno é devolvido utilizando-se a palavra chave return:

```
TipDeRetorno nomeDoMehtodo( TipoDoArg1 arg1, TipoDoArg2 arg2, ...){
  TipoDeRetorno valorDeRetorno;
  ...
  return valorDeRetorno;
}
```

Se o método não utiliza nenhum argumento, parêntese vazia, devem ser incluídas na declaração.

- Se o método não retorna nenhum valor, isto deve ser declarado usando-se a palavra-chave void:

```
void nomeDoMehtodo( TipoDoArg1 arg1, TipoDoArg2 arg2, ... ){
  ...
}
```

Métodos estáticos

- Por "default", um método efetua uma determinada operação sobre um determinado objeto, ou seja uma instância da classe na qual o método está declarado. Existem métodos que realizam operações genéricas, não relativas a uma instância particular. Tais métodos são chamados estáticos e são declarados acrescentando-se a palavra-chave static à declaração do método.
- Por exemplo, os métodos da classe Math do pacote lang, que realizam operações matemáticas sobre números, são estáticos:

```
public static int min( int a, int b ){
  ... // retorna o menor dos 2 inteiros a e b
}
```

Restrições de acesso

- Algumas palavras-chaves são diponíveis para ampliar ou restringir o acesso a um método. Estas palavras-chaves são acrescentadas à declaração do método.
- Um método que pode ser acessado por qualquer outra classe é dito público, e é declarado usando-se a palavra-chave public.
- Um método que pode ser acessado somente por métodos da própria classe é dito privado, e é declarado usando-se a palavra-chave private.
- Um método que, além de poder ser acessado por todas as classes do mesmo pacote, também pode ser acessado pelas subclasses da classe na qual ele é declarado, é dito protegido e é declarado usando-se a palavra-chave protected.
- Um método para o qual não foi especificada nenhuma destas palavras-chaves é dito amigável e pode ser chamado por todas as classes que pertencem ao mesmo pacote.

- Exemplo:

```
package algumPacote;
public class NomeDaClasse{
   public int mehtodo1( ){
     ... // acessível por qualquer classe
   }
```

```
   protected int mehtodo2( ){
     ... // acessível por esta classe e suas subclasses
   }
   int mehtodo3( ){
     ... // acessível pelas classes deste pacote
   }
   private int mehtodo4( ){
     ... // acessível por esta classe
   }
}
```

Como um objeto chama um método de outro objeto

- Supondo que as restrições de acesso discutidas acima o permitam, uma classe que possui uma referência a um objeto pode chamar (executar) um método pertencente este objeto através do ***operador ponto***:

```
/* o método abaixo não requer argumento e não retorna nada */
   nomeDoObjeto.mehtodo1( );
/* o método abaixo requer dois argumentos e não retorna nada */
   nomeDoObjeto.mehtodo2( var1, var2 );
/* o método abaixo não requer argumento e retorna um valor do tipo Resultado
   */
   Resultado resultado = objeto.mehtodo3( );
```

- Se a variável de retorno for um objeto, pode-se concatenar operadores pontos para chamar métodos deste objeto:

nomeDoObjeto.mehtodo3().mehtodo4();

- Se o método for estático, usa-se a classe, em vez de uma instância particular, para chamar o método:

```
int x = Math.min( a, b );
```

Como criar um objeto - método construtor

- Para criar uma objeto, usa-se a palavra chave new, seguida de uma chamada ao método construtor, cujo nome é idêntico ao da classe:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse( );
```

- O método construtor não precisa ser incluído explicitamente na classe, mas pode ser incluído para realizar tarefas no ato da criação. A sintaxe é:

```
class NomeDaClasse{
   NomeDaClasse( ){
     ... // comandos executados na criação do objeto
   }
}
ou, para uma classe pública:
public class NomeDaClasse{
   public NomeDaClasse( ){
     ... // comandos executados na criação do objeto
   }
}
```

- O método construtor pode receber argumentos:

```
public class NomeDaClasse{
   public NomeDaClasse( Tipo1 arg1, Tipo2 arg2 ){
     ... // comandos executados na criação do objeto
   }
}
```

Valores para estes argumentos devem então ser passados ao construtor no comando de criação:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse( valorArg1, valorArg2 );
```

Prática

Classe representando um vetor tridimensional

Desenvolva uma classe que represente um vetor tridimensional. Dicas:

- Suponha (por enquanto) que o vetor é construído a partir das suas componentes cartesianas.
- Deve ser possível para qualquer outro objeto descobrir as componentes cartesianas e as componentes esféricas do vetor.
- Funções matemáticas úteis podem ser encontradas no pacote java.lang.
- Quais outros recursos você gostaria de incluir na sua classe?

Máquina para cálculo vetorial

Desenvolva uma classe que permita realizar as operações básicas envolvendo dois vetores: soma, diferença, produto escalar, produto vetorial. Dicas:

- Os vetores em questão são representantes da classe desenvolvida na primeira parte da prática.
- Como você quer uma máquina de calcular genérica, análoga à classe Math do pacote java.lang, você vai querer que os métodos que realizam as operações sejam estáticos.

Utilização no programa da prática anterior

Modifique o programa desenvolvido na prática anterior para fazer uso dos recursos desenvolvidos nesta prática.

Programação orientada a objetos: Aspectos fundamentais

Sobrecarga de métodos

- É permitido incluir numa classe métodos que possuem o mesmo nome e o mesmo tipo de retorno, mas que diferem pelo número e/ou pelos tipos dos argumentos.
- Qual destes métodos é executado depende do número e/ou dos tipos dos argumentos passados pelo programa que chama o método. Esta técnica é chamada sobrecarga ("overloading") de métodos. Ela é frequentemente utilizada, por exemplo, para definir vários construtores para uma determinada classe.

- Exemplo:

```
public class Ponto{
   protected double x, y;
   public Ponto(){
     setPonto( 0, 0 );
   }
   public Ponto( double a, double b ){
     setPonto( a, b );
   }
   public void setPonto( double a, double b ){
     x = a;
     y = b;
   }
   public String emPalavras(){
     return "[" + x + ", " + y + "]";
   }
}
```

Herança

- Pode-se criar uma nova classe, acrescentando recursos a uma classe já construída, ou modificando alguns recursos desta classe. A nova classe herda (possivelmente com modificações) os recursos já disponíveis na classe anterior.
- A classe herdeira é chamada subclasse da classe anterior, que é a sua superclasse.
- Este processo de herança pode ser repetido em cascata, criando várias gerações de classes.
- Vale notar que a linguagem Java permite somente herança simples, i.e. uma classe pode herdar diretamente de uma única classe (possui uma única superclasse direta). Em contrapartida, uma classe pode possuir várias subclasses diretas.
- A relação de herança é indicada através da palavra-chave extends. Por "default", uma classe herda da classe Object.

- Exemplo: O exemplo abaixo cria uma subclasse da classe definida no exemplo anterior:

```
public class Circulo extends Ponto{
   protected double r;
   public Circulo(){
     /* aqui ocorre uma chamada implicita ao construtor da superclasse */
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
     super( a, b );
     setRaio( r );
   }
   public void setRaio( double r ){
     this.r = r;
   }
}
```

- No exemplo acima, as variáveis x e y da superclasse são acessíveis à subclasse, pois foram declaradas protected.
- A palavra-chave super permite referência à superclasse, e em especial, se for utilizada como nome de método, ao construtor da superclasse.
- Nota-se que se não houver, no construtor da subclasse, nenhuma chamada explicita ao construtor da superclasse, o construtor sem argumento é chamado por "default". Se for incluída uma chamada ao construtor da superclasse, ela deve ser o primeiro comando executado no construtor da subclasse.

Superação de métodos

- Na construção de uma subclasse a partir de uma superclasse, pode-se também substituir um método já presente na superclasse por outro, de mesmo nome, mesmo tipo de retorno, igual número e mesmos tipos de argumentos. Este procedimento é chamado "overriding" em inglês, o que será traduzido aqui por superação.

- Exemplo: na classe Circulo acima, podemos substituir o método emPalavras por uma versão incrementada cujo valor de retorno inclui também o raio do círculo:

```
public class Circulo extends Ponto{
   protected double r;
   public Circulo(){
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
     super( a, b );
     setRaio( r );
   }
   public void setRaio( double r ){
     this.r = r;
   }
   public String emPalavras(){
     return "[" + x + ", " + y + ", " + r + "]";
   }
}
```

- Vale notar que o método superado ainda pode ser utilizado, caso for necessário, fazendo-se uso da palavra-chave super. Por exemplo, uma possível alternativa à construção acima (produzindo porém um resultado ligeiramente diferente) seria:

```
public class Circulo extends Ponto{
   protected double r;
   public Circulo(){
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
     super( a, b );
     setRaio( r );
   }
   public void setRaio( double r ){
     this.r = r;
   }
   public String emPalavras(){
     return super.emPalavras() + "[" + r + "]";
   }
}
```

Polimorfismo

- Já que uma subclasse herda as propriedades da superclasse, uma instância de uma subclasse é também uma instância da sua superclasse (e da superclasse desta, etc... até a classe raiz Object).
- Assim, no exemplo acima, um Circulo também é um Ponto e uma instância de Circulo pode ser atribuída a uma referência declarada como Ponto.

- Exemplo:

```
import java.awt.Label;
import java.applet.Applet;
public class Test extends Applet{
   private Ponto meusPontos[];
   private Label rohtulos[];
   public void init(){
     meusPontos = new Ponto[ 2 ];
     meusPontos[ 0 ] = new Ponto( 3, 5 );
     meusPontos[ 1 ] = new Circulo( 15, 5, 1.5 );
     rohtulos = new Label[ 2 ];
     for( int i = 0; i < meusPontos.length; i++ )
     {
       rohtulos[ i ] = new Label( meusPontos[ i ].emPalavras() );
       add( rohtulos[ i ] );
     }
   }
}
```

Neste exemplo, apesar de ambos objetos serem declarados como Ponto, o segundo é de fato uma instância da subclasse Circulo e portanto, quando for chamado o método emPalavras deste objeto, será executado o método da classe Circulo (que retorna informação também sobre o raio). Este recurso da linguagem é chamado polimorfismo, pois ele faz com que objetos declarados como instâncias de uma classe (no caso, a classe Ponto) possam assumir diversas formas (no caso, um deles é um mero Ponto e o outro é um Circulo, que é de verdade um ponto incrementado).

- Se uma instância de uma subclasse for atribuída a uma referência do tipo da superclasse, somente os métodos que já estão definidos na superclasse podem ser chamados como descrito acima. Para chamar um método definido na subclasse, mas não na superclasse, é necessário efetuar uma coerção explicita. Por exemplo, se acrescentarmos à classe Circulo um método para calcular a área, ou seja,

```
public class Circulo extends Ponto{
   protected double r;
   public Circulo(){
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
     super( a, b );
     setRaio( r );
   }
   public void setRaio( double r ){
     this.r = r;
   }
   public double ahrea( ){
     return Math.PI * r * r;
   }
   public String emPalavras(){
     return "[" + x + ", " + y + ", " + r + "]";
   }
}
```

devemos usar a coerção para chamar este método numa instância de Circulo que foi atribuídda a uma referência de tipo Ponto:

```
import java.awt.Label;
import java.applet.Applet;
public class Test extends Applet{
   private Ponto meusPontos[];
   private Label rohtulos[];
   public void init(){
     meusPontos = new Ponto[ 2 ];
     meusPontos[ 0 ] = new Ponto( 3, 5 );
     meusPontos[ 1 ] = new Circulo( 15, 5, 1.5 );
     rohtulos = new Label[ 2 ];
     for( int i = 0; i < meusPontos.length; i++ ){
       rohtulos[ i ] = new Label( meusPontos[ i ].emPalavras() );
       add( rohtulos[ i ] );
     }
     double a = ( ( Circulo ) meusPontos[ 1 ] ).ahrea();
     rohtulos[ 1 ].setText( rohtulos[ 1 ].getText() + "[" + a + "]" );
   }
}
```

- Em linguagens orientadas a objeto, a palavra polimorfismo é também usada num sentido um tanto diferente, qual seja para referir-se às várias "formas" que pode tomar uma classe que herda de várias outras. Embora Java não permite herança múltipla, ele oferece um recurso alternativo, chamado interface, que possibilita este tipo de polimorfismo (veja abaixo).

Recursos adicionais

Interfaces

- Como já mencionado, Java não permite que uma classe seja herdeira dos recursos de mais de uma outra classe. Esta limitação é imposta para evitar que o programador possa cometer erros sutís e perigosos, caso ele quiser construir uma classe que herde de duas outras que possuem um método de mesmo nome (e tipos de argumentos), mas com implementações diferentes nas duas classes. Neste caso, haveria ambiguidade no funcionamento do método herdado. Do outro lado, a herança múltipla é um recurso poderoso, cujo total abandono limitaria bastante a linguagem.
- A saída encontrada é o conceito de interface, que nada mais é que um conjunto de declarações de métodos (nome, tipo de retorno, tipos dos argumentos) desprovidos de implementação. Cabe ao programador que deseja implementar a interface em questão providenciar uma implementação destes métodos na classe que ele está desenvolvendo.
- Desta forma, além de estender alguma superclasse, a classe em desenvolvimento pode implementar várias interfaces.
- A palavra chave implements é utilizada para indicar que uma classe implementa uma determinada interface.

- Exemplo: definimos a interface:

```
interface Forma{
   public double ahrea();
   public String emPalavras();
}
```

Podemos então declarar que a classe Circulo construida acima implementa esta interface:

```
public class Circulo extends Ponto implements Forma{
   protected double r;
   public Circulo(){
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
     super( a, b );
     setRaio( r );
   }
   public void setRaio( double r ){
     this.r = r;
   }
   public double ahrea( ){
     return Math.PI * r * r;
   }
   public String emPalavras(){
     return "[" + x + ", " + y + ", " + r + "]";
   }
}
```

Poderíamos definir uma outra classe que implementa a mesma interface:

```
public class Quadrado extends Ponto implements Forma{
   protected double l;
   public Quadrado(){
     setLado( 0 );
   }
   public Quadrado( double a, double b, double l ){
     super( a, b );
     setLado( l );
```

```
    }
    public void setLado( l ){
      this.l = l;
    }
    public double ahrea( ){
      return l * l;
    }
    public String emPalavras(){
      return "[" + x + ", " + y + ", " + l + "]";
    }
}
```

- No exemplo acima, um objeto de tipo Circulo é também uma instância de uma Forma. O mesmo é verdade de um objeto de tipo Quadrado. O uso deste polimorfismo está ilustardo no exemplo abaixo:

```
import java.awt.Label;
import java.applet.Applet;
public class Test extends Applet{
   private Forma minhasFormas[];
   private Label rohtulos[];
   public void init(){
     minhasFormas = new Forma[ 2 ];
     minhasFormas[ 0 ] = new Circulo( 3, 5, 1.5 );
     minhasFormas[ 1 ] = new Quadrado( 15, 5, 2.5 );
     rohtulos = new Label[ 2 ];
     for( int i = 0; i < minhasFormas.length; i++ ){
       String descr = minhasFormas[ i ].emPalavras();
       double ar = minhasFormas[ i ].ahrea();
       rohtulos[ i ] = new Label( descr + "; área: " + ar );
       add( rohtulos[ i ] );
     }
   }
}
```

Classes abstratas

- A linguagem Java ainda oferece um meio termo entre uma classe na qual está providenciada uma implementação para todos os métodos declarados, e uma interface na qual nenhum dos métodos declarados está implementado. Trata-se da classe abstrata, na qual alguns dos métodos declarados estão implementados e outros não.
- Uma classe abstrata deve ser declarada tal usando-se a palavra chave abstract.
- Uma classe abstrata não pode ser instanciada diretamente. Pode-se construir subclasses da classe abstrata, nas quais os métodos declarados mas ainda não implementados devem receber uma implementação.

- Exemplo: uma alternativa ao esquema do exemplo anterior seria definir uma classe abstrata Forma:

```
public abstract class Forma extends Ponto{
   abstract double ahrea( );
}
```

Podemos então definir as classes Circulo e Quadrado como subclasses concretas desta classe abstrata:

```
public class Circulo extends Forma{
   protected double r;
   public Circulo(){
     setRaio( 0 );
   }
   public Circulo( double a, double b, double r ){
```

```
      super( a, b );
      setRaio( r );
   }
   public void setRaio( double r ){
      this.r = r;
   }
   public double ahrea( ){
      return Math.PI * r * r;
   }
   public String emPalavras(){
      return "[" + x + ", " + y + ", " + r + "]";
   }
}

public class Quadrado extends Forma{
   protected double l;
   public Quadrado(){
      setLado( 0 );
   }
   public Quadrado( double a, double b, double l ){
      super( a, b );
      setLado( l );
   }
   public void setLado( l ){
      this.l = l;
   }
   public double ahrea( ){
      return l * l;
   }
   public String emPalavras(){
      return "[" + x + ", " + y + ", " + l + "]";
   }
}
```