

## Sumário

1 Visão geral de paradigmas de programação.....	5
1.1 Breve histórico das linguagens de programação.....	5
1.2 Paradigmas de Programação.....	6
1.2.1 Paradigma Imperativo.....	6
1.2.2 Paradigma Orientado a Objetos.....	7
1.2.3 Paradigma de Programação em Lógica.....	8
1.2.4 Paradigma Funcional.....	8
1.3 Pe versus Poo.....	9
1.4 Bibliografia do capítulo.....	10
2 Introdução à Tecnologia Java.....	11
2.1 A linguagem de programação Java.....	11
2.1.1 Simples.....	11
2.1.2 Orientada a objetos.....	11
2.1.3 Multithread.....	11
2.1.4 Independente de arquitetura.....	11
2.1.5 Portável.....	11
2.1.6 Distribuída.....	12
2.1.7 De alto desempenho.....	13
2.1.8 Robusta.....	13
2.1.9 Segura.....	13
2.1.10 Dinâmica.....	13
2.2 A plataforma Java.....	13
2.3 Como instalar.....	14
2.3.1 Ambiente Windows.....	14
2.3.2 Ambiente Ubuntu.....	16
2.3.3 Editor de código-fonte.....	16
2.4 Resumo do capítulo.....	16
2.5 Bibliografia do capítulo.....	17
3 Elementos Básicos da Linguagem Java.....	18
3.1 Estrutura de um programa Java.....	18
3.2 Parâmetros pela linha de comando.....	20
3.3 Comentários.....	21
3.4 Identificadores.....	21
3.5 Tipos de dados primitivos.....	21
3.6 Sequência de caracteres (string).....	22
3.7 Tipos de dados Referência.....	22
3.8 Criação e uso de variáveis.....	22
3.9 Operadores.....	23
3.9.1 Concatenação.....	23
3.9.2 Aritméticos.....	24
3.9.3 Unários.....	24
3.9.4 Relacionais.....	25
3.9.5 Lógicos.....	26
3.9.6 Atribuição.....	27
3.10 Comandos de entrada.....	27
3.11 Comandos de saída.....	28
3.12 Estruturas de decisão.....	28
3.12.1 If.....	28
3.12.2 Switch.....	29
3.13 Estruturas de repetição.....	31
3.13.1 For.....	31
3.13.2 While.....	32

3.13.3 Do while.....	32
3.14 Break e continue.....	33
3.15 Arrays.....	34
3.16 Resumo do capítulo.....	35
3.17 Bibliografia do capítulo.....	36
4 Implementação de classes em Java.....	37
4.1 Classes e objetos.....	37
4.1.1 Criando Classes.....	39
4.1.2 Criando Objetos.....	40
4.2 Atributos.....	41
4.3 Métodos.....	42
4.4 Mensagens.....	43
4.5 Um programa com objetos.....	43
4.6 Métodos Construtores.....	46
4.7 Membros estáticos.....	47
4.7.1 Atributos estáticos.....	47
4.7.2 Métodos estáticos.....	48
4.8 Passagem de parâmetros.....	49
4.9 Escopo de variáveis.....	49
4.10 Resumo do capítulo.....	49
4.11 Bibliografia do capítulo.....	50
5 Encapsulamento.....	51
5.1 Modificadores de visibilidade.....	51
5.2 Métodos de acesso e métodos modificadores.....	53
5.3 Resumo do capítulo.....	55
5.4 Bibliografia do Capítulo.....	56
6 Operações de Abstração.....	57
6.1 Classificação/Instanciação.....	57
6.2 Associação.....	58
6.3 Agregação/Decomposição.....	58
6.4 Generalização / Especialização.....	60
6.5 Resumo do capítulo.....	60
6.6 Bibliografia do capítulo.....	60
7 Herança.....	61
7.1 Implementação de herança em Java.....	63
7.2 Sobreposição de métodos.....	64
7.3 O uso de super.....	65
7.4 Herança e construtores.....	65
7.5 Resumo do capítulo.....	66
7.6 Bibliografia do Capítulo.....	67
8 Classes Abstratas e Interfaces.....	68
8.1 Classes Abstratas.....	68
8.2 Métodos abstratos.....	69
8.3 Interfaces.....	70
8.4 Resumo do capítulo.....	73
8.5 Bibliografia do Capítulo.....	73
9 Polimorfismo.....	74
9.1 Ligação tardia (late binding).....	76
9.2 Conversão (casting).....	77
9.2.1 Upcasting.....	77
9.2.2 Downcasting.....	78
9.2.3 O operador instanceof.....	78
9.3 Argumentos polimórficos.....	79
9.4 Polimorfismo com interfaces.....	80
9.5 Sobrecarga (Overloading).....	80

9.6 Resumo do capítulo.....	81
9.7 Bibliografia do Capítulo.....	81
10 Tratamento de Exceções.....	82
10.1 Como tratar exceções.....	82
10.1.1 Tratamento múltiplo.....	83
10.2 O Bloco finally.....	85
10.3 Jogando (ou lançando) exceções.....	86
10.4 Resumo do capítulo.....	88
10.5 Bibliografia do Capítulo.....	88
11 Pacotes e Arquivos JAR.....	89
11.1 Pacotes.....	89
11.2 Utilização de classes de pacotes.....	89
11.2.1 Usar o nome qualificado completo da classe.....	90
11.2.2 Importar uma classe específica de um pacote.....	90
11.2.3 Importar um pacote inteiro.....	91
11.3 Organização física dos arquivos.....	91
11.4 Compilando com a flag -d.....	94
11.5 Arquivos JAR.....	95
11.6 Resumo do capítulo.....	96
11.7 Bibliografia do Capítulo.....	96
12 Interface gráfica com o usuário.....	97
12.1 Swing.....	97
12.2 Componentes do Swing.....	97
12.2.1 JFrame.....	98
12.2.2 JPanel.....	100
12.2.3 JOptionPane.....	102
12.2.4 JLabel.....	102
12.2.5 JButton.....	104
12.2.6 JTextField e JPasswordField.....	105
12.2.7 JRadioButton.....	107
12.2.8 JCheckBox.....	109
12.2.9 JTextArea.....	110
12.2.10 JComboBox.....	112
12.2.11 JList.....	113
12.2.12 Listas de seleção múltipla.....	114
12.3 Gerenciadores de Layout.....	115
12.3.1 FlowLayout.....	115
12.3.2 BorderLayout.....	116
12.3.3 BoxLayout.....	117
12.3.4 GridLayout.....	117
12.3.5 Modelo de Tratamento de Eventos.....	117
12.3.6 Interface ActionListener.....	119
12.3.7 Interface ItemListener.....	119
12.3.8 Interface ListSelectionListener.....	119
12.3.9 Interface MouseListener.....	120
12.3.10 Interface MouseMotionListener.....	120
12.3.11 Interface KeyListener.....	120
12.3.12 Classe MouseEvent.....	121
12.4 Resumo do Capítulo.....	122
12.5 Bibliografia do Capítulo.....	122
13 Persistência de objetos.....	123
13.1 Serialização.....	123
13.1.1 Gravando um objeto serializado em disco.....	124
13.1.2 Recuperando um objeto serializado.....	125
13.1.3 Interface Serializable.....	125

13.1.4 Serialização e associações/composições.....	125
13.1.5 Atributos <i>transient</i> .....	126
13.1.6 Variáveis estáticas.....	127
13.1.7 Um exemplo de serialização.....	127
13.1.8 Serializando objetos em XML.....	130
13.2 Arquivos de texto.....	131
13.2.1 FileWriter e FileReader.....	131
13.2.2 BufferedWriter e BufferedReader.....	133
13.3 Bibliografia do capítulo.....	135

# 1 Visão geral de paradigmas de programação

Neste capítulo trataremos de alguns conceitos importantes sobre programação, incluindo os principais paradigmas de programação.

## 1.1 BREVE HISTÓRICO DAS LINGUAGENS DE PROGRAMAÇÃO

A origem da Programação Orientada a Objetos, ou POO, data de meados da década de 1960, embora sua divulgação ao público tenha começado de forma efetiva apenas no início da década de 1990.

Por volta de 1960, havia uma grande preocupação com a qualidade no desenvolvimento de programas, o reaproveitamento de código escrito e o tempo de desenvolvimento dos sistemas, que começava a ser demorado. Dessa necessidade desenvolveu-se a técnica de programação estruturada, a qual se torna a base fundamental para o surgimento da técnica de programação orientada a objetos.

O cenário para a área de software não era nada animador, pois a indústria de hardware estava muitos anos à frente. O modo de concepção dos programas era muito rústico, pois um programa tinha de ser escrito para um computador específico. Não existia um programa que fosse executado em qualquer computador ou plataforma. Essa concepção só apareceu muito tempo depois.

A era da computação comercial inicia-se em 1950, quando os computadores deixaram de ser ferramentas de uso militar (como ocorreu na década de 1940). A única linguagem utilizada era a de máquina, formada por códigos binários. Como o tempo, surgiu a linguagem Assembly, mais fácil do que a de máquina. Apesar de mais fácil, é uma linguagem intimamente vinculada aos recursos de máquina, o que a torna de difícil compreensão.

Um programa escrito em linguagem Assembly para um computador X não pode rodar em um computador Y. O mesmo programa necessita ser novamente escrito, segundo as regras do computador Y, o que, além de trabalhoso, é extremamente dispendioso. Com as linguagens de máquina e Assembly surge a **primeira geração** de linguagens de programação, que são de baixo nível.

Por volta de 1954 até 1968, surgem as primeiras linguagens de programação de alto nível (maior proximidade com a forma escrita humana, em inglês). A primeira a ser lançada foi FORTRAN, depois vieram o COBOL, ALGOL e BASIC, para citar as mais conhecidas, já que existem mais de 2.500 linguagens de programação catalogadas. Essas linguagens passaram a definir a **segunda geração**.

Além destas linguagens serem mais próximas da escrita humana, possibilitaram maior portabilidade, pois já era possível escrever um programa para um computador X e utilizá-lo (com algumas modificações) em um computador Y.

As linguagens de segunda geração, na sua maioria, são lineares, identificando seus comandos com linhas numeradas. Essa característica dificulta a adoção de técnicas estruturadas de codificação de programas e leva programadores inexperientes a cometerem verdadeiros absurdos de codificação.

Devido a este e outros detalhes técnicos, começa a se discutir uma maneira de tentar reaproveitar um código escrito e a necessidade de alterá-lo de forma mais rápida e ágil. Essa ideia inicial da programação estruturada acaba por nortear a programação orientada a objetos, que surge a partir de 1960 com a linguagem de programação SIMULA I, para o computador UNIVAC.

Em 1967, foi apresentada a linguagem SIMULA 67, que introduziu o conceito de classes, que é um dos principais pilares da POO. Durante a década de 1970, surge a linguagem de programação orientada a objetos SmallTalk, que incentivou e popularizou o uso da POO.

Surge então a concepção das primeiras linguagens de programação pertencentes à **terceira geração**, com uma estruturação de dados maior que as linguagens de segunda geração, mas que ainda não davam suporte à POO, que ainda estava no início.

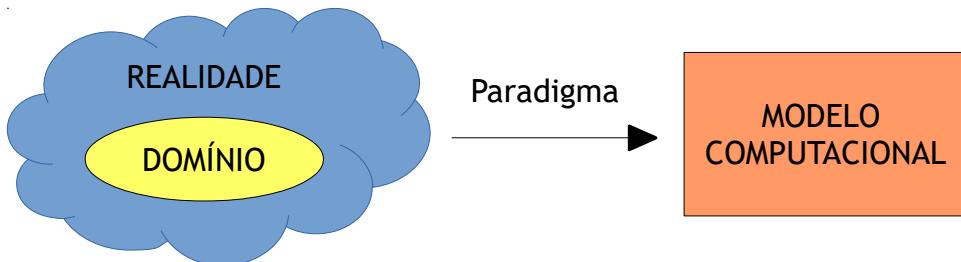
Entre 1968 e 1972, aparecem as linguagens de programação PASCAL e C (que não eram orientadas a objetos, mas estruturadas). Depois vieram outras linguagens de programação, as quais já embutiam os conceitos de orientação a objetos: SMALLTALK, EIFFEL, ADA, CLOS, SELF, BETA, JAVA, OBJECT PASCAL (linguagem Pascal orientada a objetos), C++ (linguagem C orientada a objetos), dentre outras.

A programação orientada a objetos é uma filosofia de trabalho. Não é a ferramenta em si, mas a forma de pensar e de usar a lógica de programação para a solução de um problema do mundo real através do computador. Desta forma, é necessário ao desenvolvedor mudar a forma de pensar em um problema computacional. Os primeiros programadores pensavam de forma linear, depois aprenderam a pensar de forma estruturada. Na POO, é preciso utilizar a estrutura de dados de um programa baseada na estrutura do mundo real.

## 1.2 PARADIGMAS DE PROGRAMAÇÃO

Um paradigma de programação é um modelo, padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns.

Um paradigma reflete na forma como o programador lida com um determinado problema. Não existe um paradigma melhor ou pior do que outro. Cada é mais adequado a determinado tipo de problema.



*Figura 1: Um paradigma reflete em como modelamos o domínio do problema.*

Os principais paradigmas utilizados atualmente são o imperativo, o orientado a objetos, o declarativo e o funcional. Veremos cada um deles a seguir.

### 1.2.1 Paradigma Imperativo

Um programa imperativo é composto por um conjunto de variáveis e uma sequência de comandos, que são executados passo a passo, sequencialmente. O estado do programa é mantido em suas variáveis, sendo que a execução dos comandos transforma os valores das variáveis a fim de chegar ao resultado esperado.

Exemplos de linguagens imperativas são Basic, Algol, C e Pascal.

A **programação estruturada** se enquadra no paradigma imperativo, sendo também denominada de procedural, por incluir sub-rotinas ou procedimentos como mecanismo de estruturação. Em um programa estruturado, os blocos de código são interligados por três estruturas de controle: sequência, decisão e repetição.

Veja abaixo um exemplo de um programa em C para cálculo de fatorial:

```
#include <stdlib.h>
#include <stdio.h>
```

```

int factorial(int x) {
    int result = 1;
    int i;
    for (i = 1; i <= x; i++)
        result = result * i;
    return result;
}

int main() {
    printf("O fatorial de 5 é %d\n", factorial(5));
    return EXIT_SUCCESS;
}

```

Cada comando é uma ordem dada ao sistema: atribua `1` à variável `result`; para cada `i` de `1` até o valor do argumento `x`, multiplique o valor de `result` pelo valor de `i`; retorne o valor de `result`, etc. Por isso o paradigma se chama **imperativo**, pois através de comandos, damos ordens explícitas ao computador.

### 1.2.2 Paradigma Orientado a Objetos

O paradigma orientado a objetos busca abstrair conceitos do mundo real por meio de objetos, criados a partir de classes. Cada objeto possui atributos (características) e métodos (ações), definidos em sua classe de origem. Os objetos comunicam-se chamando os métodos uns dos outros, mecanismo este denominado troca de mensagens. Algumas mensagens apenas disparam ações, enquanto que outras modificam os valores dos atributos. Um software orientado a objetos é, então, um conjunto de objetos trocando mensagens.

As classes podem ser estendidas através de um mecanismo denominado herança.

Veja abaixo um exemplo de um programa em Java para cálculo de fatorial:

```

class Fatorial {
    public int calcula(int n) {
        if (n == 0)
            return 1;
        else
            return n * calcula(n-1);
    }
}

class FatorialMain {
    public static void main (String[] args) {
        int resultado;
        Fatorial fat = new Fatorial();
        resultado = fat.calcula(5);
        System.out.println("O fatorial de 5 é: " + resultado);
    }
}

```

```

        resultado = fat.calcula(6);
        System.out.println("O fatorial de 6 é: " + resultado);
    }
}

```

Observe que criamos uma classe chamada `Fatorial`. Esta classe possui apenas um método, chamado `calcula`, que requer um parâmetro `n` e retorna um valor inteiro (`int`).

A outra classe, chamada `FatorialMain`, é o programa principal. Dentro dela temos o método `main`, onde o programa efetivamente começará a ser executado. Temos ali uma variável local `resultado` e uma variável de referência `fat`, associada a um objeto do tipo `Fatorial` (cuja estrutura foi definida na classe logo acima). Assim, `fat` terá o método `calcula`. O comando `fat.calcula(5)` executa o método `calcula`, passando como parâmetro o inteiro 5. Em seguida, imprimimos o resultado e repetimos a operação, agora com o parâmetro 6. Observe que a variável `fat` não precisa saber "como" o fatorial é calculado, ela só precisa chamar o método `calcula()` com o respectivo parâmetro.

### 1.2.3 Paradigma de Programação em Lógica

É um paradigma que faz uso da lógica matemática, onde o programa obtém conclusões a partir de um conjunto de fatos e regras lógicas. Segue o estilo declarativo, onde ao invés de se fornecer ao computador instruções sobre "como" resolver um problema, a preocupação é em declarar o problema, ou seja, especificar "o que" se espera como resultado. Suas aplicações situam-se no campo de sistemas especialistas e bancos de dados.

Exemplos deste tipo de linguagem são Prolog, Planner, QLisp, Popler, Mercury.

Veja abaixo um exemplo de regras em Prolog para cálculo de fatorial:

```

fatorial(0, 1).
fatorial(N, F) :-
    N > 0,
    N1 is N - 1,
    fatorial(N1, F1),
    F is N * F1.

```

As regras acima especificam que o fatorial de 0 é 1 e que o fatorial de N é F quando N é maior que zero e N1 é o antecessor de N e o fatorial de N1 é F1 e F é igual a N \* F1.

Para utilizar o programa acima, bastaria digitar no *prompt* do Prolog a seguinte consulta:

```
?- fatorial(5, x).
```

A resposta seria:

```
x=120
```

### 1.2.4 Paradigma Funcional

Neste paradigma, programas são formados exclusivamente por funções. Um programa é uma função composta por funções menores, as quais, por sua vez, são definidas em termos de outras mais simples, até o nível em que as funções são primitivas da linguagem. Funções podem ser parâmetros ou valores de entrada para outras funções e podem ser os valores de retorno ou saída de uma função. Programas funcionais não contêm comandos de atribuição e a ordem dos comandos é irrelevante. São empregadas em inteligência artificial e prototipação em geral.

Exemplo de linguagens funcionais: Haskell, Scheme, Lisp, Common Lisp, ML, Miranda.

Veja abaixo um exemplo de um programa em Common Lisp para cálculo de fatorial:

```
(defun factorial ((x integer))
  (if (zerop x)
    1
    (* x (factorial (- x 1)))))
(format t "~%0 factorial de 5 é ~A~&" (factorial 5))
```

O programa acima é composto pela função `factorial`, que recebe um parâmetro inteiro nomeado `x`, é definida como o resultado da função `if`; a função `if` recebe como primeiro parâmetro o resultado da função `zerop`, que é verdadeiro quando `x` é igual a zero; o segundo parâmetro de `if`, `1`, é o resultado caso o primeiro parâmetro seja verdadeiro; o terceiro parâmetro é o resultado caso seja falso; o terceiro parâmetro é o resultado da função de multiplicação.

### 1.3 PE VERSUS POO

Como vimos acima, os paradigmas funcional e lógico possuem aplicações mais específicas nas áreas científicas e de inteligência artificial. Na programação de sistemas em geral, o mais comum é o uso dos paradigmas estruturado e orientado a objetos.

Apesar das acirradas discussões no mercado computacional, que colocam a PE (Programação Estruturada) como ultrapassada e obsoleta e a POO (Programação Orientada a Objetos) como vanguarda da programação, devemos entender que ambas são técnicas complementares. A POO originou-se a partir da PE e emprega muitos de seus recursos. Por este motivo, a PE continua, em geral, sendo o primeiro paradigma ensinado a um programador iniciante.

Um programa em PE é composto por procedimentos (ou funções) que manipulam valores de variáveis. O estado da aplicação é representado pelos valores de suas variáveis. O programa consiste em manipular e transformar os valores das variáveis até chegar ao resultado esperado.

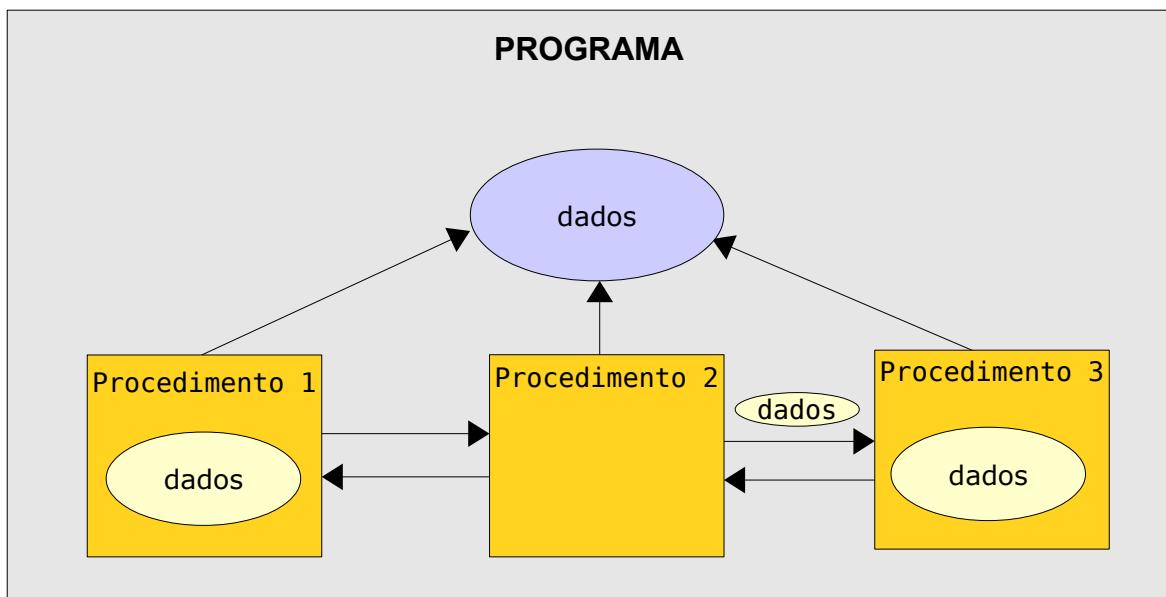
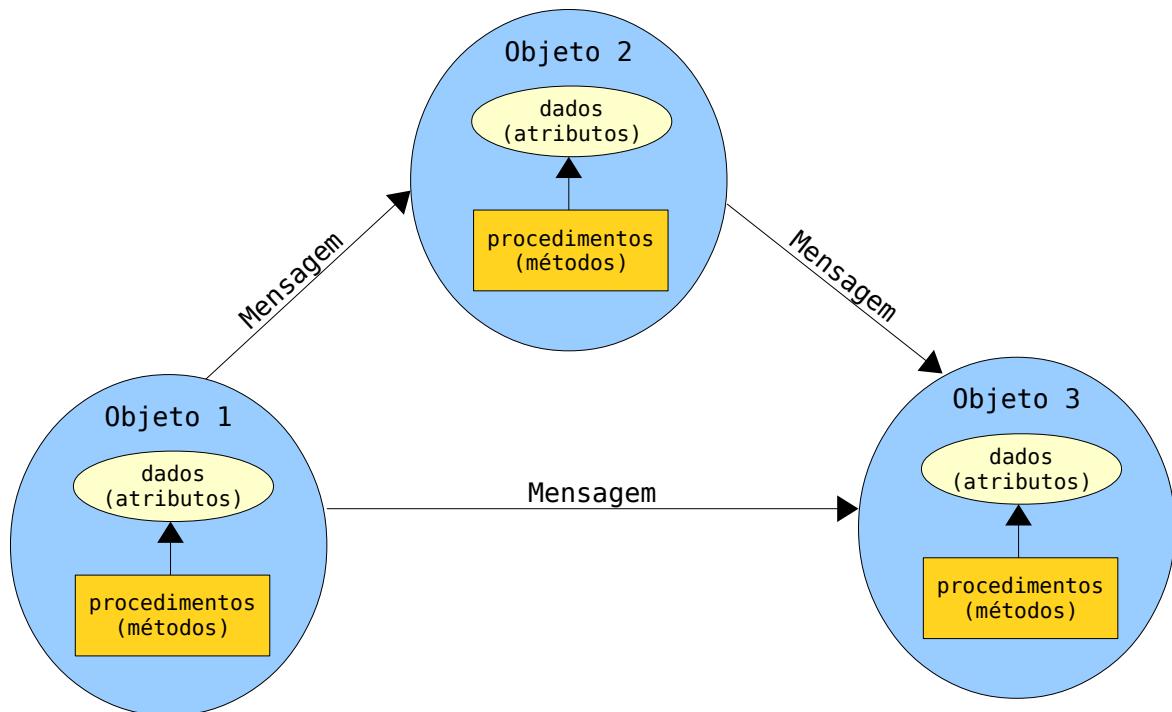


Figura 2: Representação de um programa estruturado.

Na POO, os programas são organizados como um conjunto de objetos cooperantes. O objeto encapsula dados e procedimentos em uma única unidade, e comunica-se com os demais objetos através de troca de mensagens. Uma mensagem chama (ou invoca) um método no objeto

que a recebe, podendo ou não realizar passagem de parâmetros.

O estado da aplicação é representado pelo estado de cada objeto.



*Figura 3: Representação de um programa orientado a objetos.*

Os objetos existem na memória do computador durante a execução do programa. Todo objeto é criado a partir de uma classe. É na classe que definimos quais atributos e métodos um objeto terá. Ou seja, uma classe é um modelo para a criação de objetos.

Estudaremos o paradigma de orientação a objetos com mais detalhes no capítulo 3.

#### 1.4 BIBLIOGRAFIA DO CAPÍTULO

MANZANO, José Augusto N.G.; OLIVEIRA, Jayr F. de. **Algoritmos: Lógica para Desenvolvimento de Programação de Computadores**. 23 ed. rev. São Paulo: Érica, 2010.

KODUMARO. **Paradigmas de Programação**. Disponível em <<http://kodumaro.blogspot.com/2010/08/paradigmas-de-programacao.html>>. Acesso em 14 fev. 2011.

## 2 Introdução à Tecnologia Java

O termo Tecnologia Java abrange dois elementos: uma linguagem de programação e uma plataforma de software (API e máquina virtual). A seguir, detalharemos estes dois componentes.

### 2.1 A LINGUAGEM DE PROGRAMAÇÃO JAVA

A linguagem de programação Java é uma linguagem de alto nível que possui como principais características:

- Simples
- Orientada a objetos
- *Multithread*
- Interpretada
- Independente de arquitetura
- Portável
- Distribuída
- De alto desempenho
- Robusta
- Dinâmica
- Segura



#### 2.1.1 Simples

Permite o desenvolvimento de sistemas em diferentes sistemas operacionais e arquiteturas de hardware, sem que o programador tenha que se preocupar com detalhes de infraestrutura. Dessa forma, o programador consegue desempenhar seu trabalho de uma forma mais produtiva e eficiente. Conceitos como herança múltipla, sobrecarga de operadores e ponteiros, encontrados em linguagens como C e C++, não são implementados em Java.

Com sintaxe semelhante à das linguagens C e C++, é simples de aprender e permite que os programadores sejam produtivos desde o início.

#### 2.1.2 Orientada a objetos

Java segue o paradigma da orientação a objetos, o qual traz um enfoque totalmente diferente da programação estruturada (como na linguagem C) no sentido de buscar formas mais próximas do mecanismo humano para gerenciar a complexidade de um sistema. Nesse paradigma, o mundo real é visto como sendo constituído de objetos autônomos que interagem entre si. Cada objeto possui seu próprio estado (atributos) e comportamento (métodos), de forma semelhante ao seu correspondente no mundo real. Conceitos como herança, encapsulamento e polimorfismo podem ser empregados.

#### 2.1.3 Multithread

Permite a criação de programas que implementam o conceito de *multithreading*, uma técnica de programação que possibilita que múltiplas linhas de execução (*threads*) executem concorrentemente de forma eficiente, incluindo sofisticados mecanismos de comunicação entre processos.

#### 2.1.4 Independente de arquitetura

A linguagem Java foi projetada para dar suporte a sistemas que serão implementados em ambientes de rede com plataformas (hardware e software) heterogêneas, como os ambientes Unix, Linux e Mainframe. Nesses ambientes, o sistema deve ser capaz de ser executado em diferentes hardwares, como servidor Unix da HP ou servidor Unix da IBM. Para acomodar esta situação de interoperabilidade, o compilador Java gera os programas em um formato conhecido como *bytecode*, permitindo que o mesmo possa ser executado em qualquer arquitetura.

#### 2.1.5 Portável

A portabilidade dos programas Java é garantida pela Máquina Virtual Java (Java Virtual Machine – JVM). O compilador de cada plataforma baseia-se na especificação da JVM

correspondente para gerar o código em *bytecode*.

Na linguagem de programação Java, todo o código fonte é escrito em arquivos no formato de texto puro (ASCII) e salvo em um arquivo com extensão `.java`.

Após a compilação desse arquivo pelo compilador `javac.exe`, um novo arquivo será automaticamente criado, com o mesmo nome do arquivo original, mas com extensão `.class`. O arquivo `.class` representa o arquivo em formato *bytecode* - a linguagem de máquina da Máquina Virtual Java (Java VM). Portanto, um arquivo `.class` pode ser executado em qualquer dispositivo que tenha uma JVM instalada. A execução do programa se dá através do `java.exe`, que executa um programa Java como uma instância da JVM.

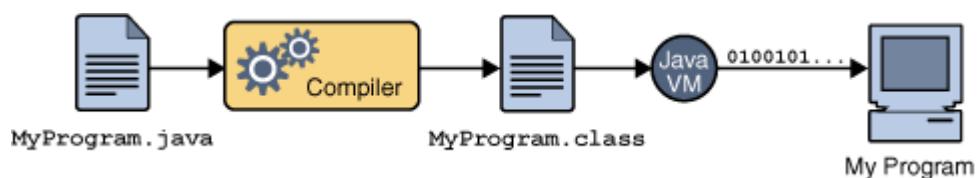


Figura 4: Etapas de compilação e execução de um programa Java.

Como a JVM está disponível para vários sistemas operacionais e plataformas de hardware, os mesmos arquivos com extensão `.class` são capazes de executar no Windows, no Linux, no Mac OS, no Solaris, etc.

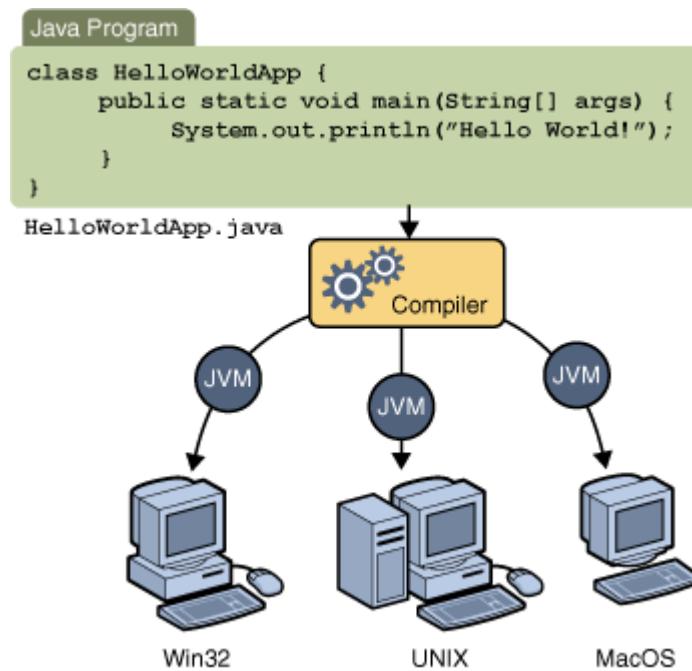


Figura 5: Através da JVM, a mesma aplicação pode rodar em diferentes plataformas.

## 2.1.6 Distribuída

Permite o desenvolvimento de aplicações em rede através de implementações cliente-servidor, applets, sockets, chamada remota de procedimentos (RPC) e servlets.

### 2.1.7 De alto desempenho

A plataforma Java oferece um bom desempenho, pois executa um código que foi previamente analisado e convertido para um formato intermediário.

Outro elemento que auxilia no bom desempenho é o recurso *garbage collector* (coletor de lixo), que é executado em segundo plano como uma *thread* de baixa prioridade, procurando liberar memória que não está mais sendo utilizada para que esta possa ser reutilizada por outra parte do sistema.

Sendo um ambiente independente de plataforma, a execução de um programa Java pode ser um pouco mais lenta do que de um código nativo dependente de plataforma. A linguagem Java também permite que um programa seja compilado para uma plataforma específica. Entretanto, nesse caso o programa não poderá ser portado automaticamente para outra plataforma, exigindo a necessidade de uma nova compilação.

### 2.1.8 Robusta

A linguagem Java foi projetada para a criação de sistemas altamente confiáveis. São feitas diversas checagens em tempo de compilação, seguidas por um segundo nível de checagem em tempo de execução (*run-time*). As características da linguagem guiam naturalmente o programador a desenvolver bons hábitos de programação.

### 2.1.9 Segura

Tanto a linguagem Java como a JVM garantem que em um ambiente de rede, nenhum programa Java permita que outro programa escrito em qualquer outra linguagem possa se esconder em um código Java a fim de se instalar automaticamente.

### 2.1.10 Dinâmica

A linkagem do programa no formato bytecode é realizada durante a execução de forma simples e totalmente gerenciada pela JVM. Classes são carregadas somente no momento da utilização e novos módulos de código podem ser carregados sob demanda a partir de várias fontes, inclusive através de uma rede. Isso permite a atualização transparente das aplicações, resultando em serviços on-line que evoluem conforme constantemente.

## 2.2 A PLATAFORMA JAVA

Uma plataforma é o ambiente de hardware e software no qual um programa é executado. As plataformas mais comuns, como Microsoft Windows, Linux, Solaris OS e Mac OS, podem ser descritas como a combinação de sistema operacional e hardware subjacente. Nesse sentido, a plataforma Java difere das demais por ser uma plataforma apenas de software, que roda sobre outras plataformas baseadas em hardware.

A plataforma Java possui dois componentes:

- *Java Virtual Machine (JVM)*: conforme já vimos, é a responsável por executar os arquivos compilados em *bytecode* e está disponível para várias plataformas de sistema operacional/hardware.
- *Java Application Programming Interface (API)*: a API Java é uma extensa coleção de componentes de software prontos que fornecem funcionalidades bastante úteis ao programador. Estes componentes são agrupados em bibliotecas de classes e interfaces conhecidas como pacotes (*packages*).

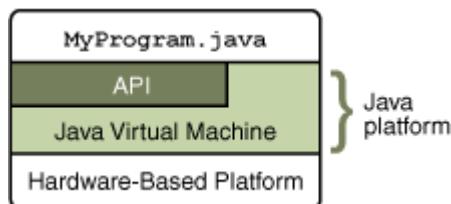


Figura 6: A API e a Java Virtual Machine isolam o programa da dependência do hardware.

A plataforma Java possui quatro diferentes "versões", cada uma incluindo a JVM e a API correspondente a um determinado tipo de aplicação. São elas:

- **Java SE (Java Platform, Standard Edition)**: é a base da plataforma, incluindo o ambiente de execução e as bibliotecas comuns. Em geral, quando alguém fala em Java, está se referindo à edição SE (edição padrão). É esta plataforma que adotaremos nesta disciplina.
- **Java EE (Java Platform, Enterprise Edition)**: construída sobre a plataforma SE, se destina ao desenvolvimento de aplicações corporativas de larga escala, seguras, escaláveis e de múltiplas camadas para ambientes de rede.
- **Java ME (Java Platform Micro Edition)**: oferece um ambiente para o desenvolvimento de aplicações voltadas a dispositivos móveis e embarcados, como celulares e PDAs.
- **JavaFX**: plataforma voltada ao desenvolvimento das chamadas RIA – *Rich Internet Applications*, aplicações multimídia ricas e com visual agradável que integram áudio, vídeo, animação, texto e web services.

## 2.3 COMO INSTALAR

Um programa escrito em Java requer dois componentes para poder ser executado: a máquina virtual (JVM) e um conjunto de bibliotecas de classe (API) que disponibilizam serviços para este programa. O ambiente de execução que fornece estes componentes é o JRE (*Java Run-Time Environment*). A instalação do JRE torna o computador apto a rodar programas Java.

Por outro lado, se além de executar você também deseja desenvolver e compilar seus programas Java, é necessário instalar o kit de desenvolvimento Java, o JDK – *Java Development Kit*. O JDK inclui o compilador e o ambiente de execução JRE (o qual, como foi dito, contém a JVM e as bibliotecas da API).

### 2.3.1 Ambiente Windows

Faça o download do JDK na página:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Clique no botão *Download Java Platform (JDK)*. Em seguida, aceite os termos da licença, selecione seu sistema operacional e prossiga com o download.

Após a instalação, configure a variável de ambiente PATH do Windows para incluir o caminho para a pasta bin da instalação do JDK. Assim, você poderá utilizar o compilador a partir de qualquer pasta do sistema, sem a necessidade de acessar toda vez o caminho **C:\Program Files\Java\jdkversion\bin**, por exemplo.

Para isso, acesse o item **Sistema** no **Painel de Controle**. Na opção **Configurações Avançadas do Sistema**, clique no botão **Variáveis de Ambiente**. Na área **Variáveis do sistema**, localize a variável Path, conforme visto na Fig. 7.

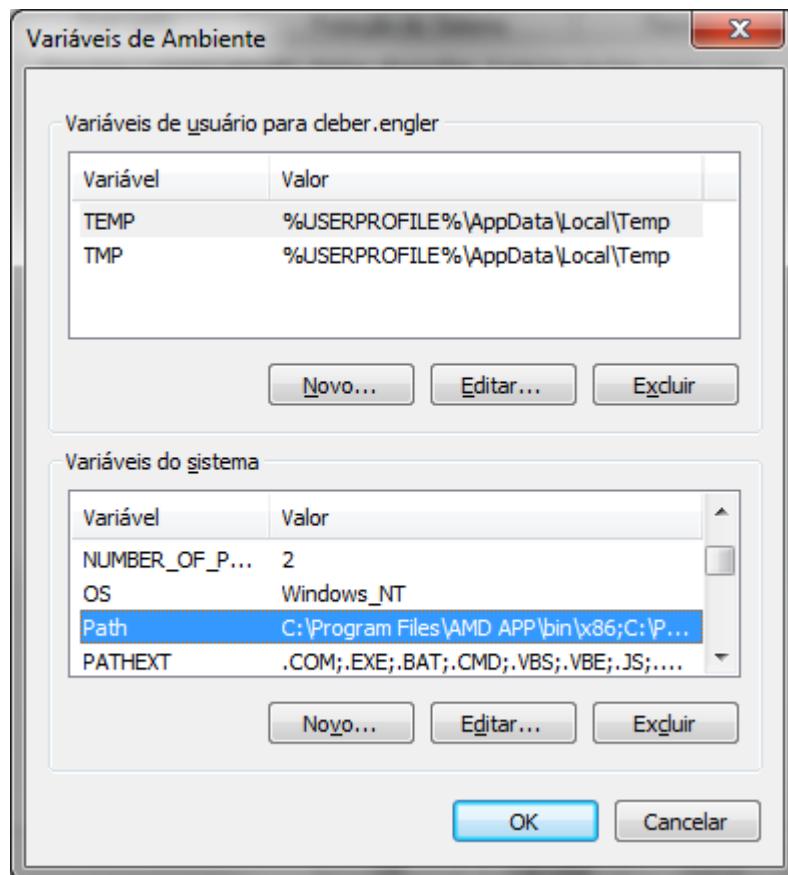


Figura 7: Localizando a variável de ambiente Path.

Você deve abrir a variável Path e, ao final do seu conteúdo, adicionar um ponto e vírgula, seguido do caminho completo para a pasta bin da sua instalação do JDK (normalmente será algo como **C:\Program Files\Java\jdkversion\bin**), onde version é o número da versão. Veja na Fig. 8 a configuração do Path para a versão 1.8.0\_0102.

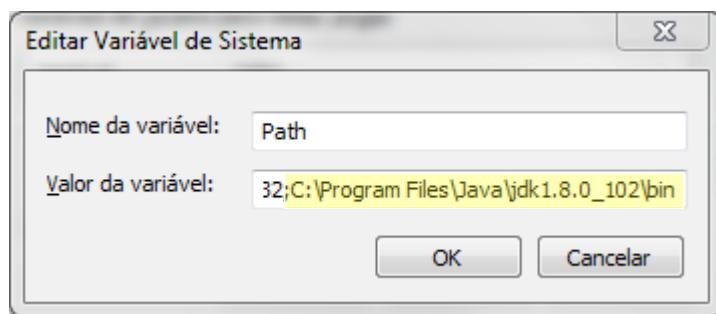


Figura 8: Configurando a variável de ambiente PATH.

Em seguida, abra o Prompt de comando (localize-o no menu Iniciar) e digite **java -version**. Observe se é exibida a versão instalada, como na figura abaixo. Teste também o compilador digitando **javac** e veja se o comando é reconhecido.

```
C:\>java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) Client VM (build 25.102-b14, mixed mode, sharing)
```

Figura 9: Testando o comando **java** em ambiente Windows.

```
C:\>javac
Usage: javac <options> <source files>
where possible options include:
  -g                           Generate all debugging info
  -g:none                      Generate no debugging info
  -g:{lines,vars,source}        Generate only some debugging info
  -nowarn                      Generate no warnings
  -verbose                     Output messages about what the compiler is doing
  -deprecation                 Output source locations where deprecated APIs are u
sed
  -classpath <path>           Specify where to find user class files and annotati
on processors
```

Figura 10: Testando o comando **javac** em ambiente Windows.

### 2.3.2 Ambiente Ubuntu

Vamos instalar o OpenJDK, um Java Development Kit baseado em software livre e de código aberto.

Execute os comandos abaixo no terminal para adicionar o repositório, atualizar a lista de pacotes e instalar o OpenJDK:

```
sudo add-apt-repository ppa:openjdk-r/ppa
sudo apt-get update
sudo apt-get install openjdk-8-jdk
```

Se há mais de uma versão do Java instalado em seu sistema, execute os comandos abaixo para definir, respectivamente, o interpretador e o compilador Java padrão (para trocar basta digitar o número à esquerda da versão):

```
sudo update-alternatives --config java
sudo update-alternatives --config javac
```

Por fim, os comandos abaixo verificam a versão que está sendo utilizada:

```
java -version
javac -version
```

### 2.3.3 Editor de código-fonte

Para escrever o código-fonte, podemos usar um editor de texto qualquer, como o bloco de notas no Windows ou o Gedit no Linux. Entretanto, existem editores específicos para programação chamados IDE (*Integrated Development Environment*). Para Java, existem os IDEs Eclipse, NetBeans, JBuilder, BlueJ, dentre outros. Eles integram a edição de código com várias funcionalidades bastante úteis, mas recomendadas para usuários mais experientes.

Na nossa disciplina, usaremos o **Geany**, um editor de texto simples indicado para iniciantes. O Geany pode ser baixado pela central de programas do Ubuntu.

## 2.4 RESUMO DO CAPÍTULO

- Os arquivos de código-fonte escritos em linguagem Java possuem extensão .java.
- Os arquivos compilados estão em um formato chamado de *bytecode* e possuem extensão .class.
- A JVM é um interpretador do código Java que fica entre as aplicações Java e o

sistema operacional.

- A JVM interpreta o código Java compilado para uma plataforma (hardware e sistema operacional) específica, fazendo assim com que os programas Java possam ser executados em várias plataformas.
- Para executar programas Java, devemos ter o ambiente de execução JRE instalado.
- Para desenvolver e compilar programas Java, devemos instalar o kit de desenvolvimento JDK.

## 2.5 BIBLIOGRAFIA DO CAPÍTULO

**The Java Tutorials: Getting Started.** Disponível em <<http://download.oracle.com/javase/tutorial/getStarted/TOC.html>>.

GOSLING, James; MCGILTON, Henry. **The Java Language Environment**. 1996. Disponível em <<http://java.sun.com/docs/white/langenv/>>.

ORACLE. **Differences between Java EE and Java SE**. 2010. Disponível em <<http://download.oracle.com/javaee/6/firstcup/doc/gkhoy.html>>.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

WILLIAN, Renan. **Configurando o JDK no Windows 7**. <http://renanwillian.wordpress.com/2010/05/10/java-configurando-jdk-no-windows-7/>.

### 3 Elementos Básicos da Linguagem Java

Neste capítulo iniciaremos o estudo dos principais elementos da linguagem de programação Java, como uso de variáveis, comandos de entrada e saída, tipos primitivos, estruturas de controle, operadores, entre outros. Por enquanto, desenvolveremos programas simples utilizando uma única classe. Portanto, não estaremos realmente programando de forma orientada a objetos. O objetivo é obter um primeiro contato com a linguagem para familiarizar-se com sua sintaxe antes que passemos realmente aos conceitos de programação orientada a objetos.

#### 3.1 ESTRUTURA DE UM PROGRAMA JAVA

No capítulo anterior, vimos que devemos escrever nosso código-fonte e armazená-lo num arquivo com extensão `.java`, para depois compilá-lo em um arquivo `.class` de mesmo nome, o qual será então executado pela JVM. Algumas perguntas que podem lhe ocorrer são:

##### O que existe em um arquivo fonte?

Um arquivo de código-fonte (com extensão `.java`) contém a definição de uma classe. Uma classe representa uma parte de seu programa (embora um aplicativo muito pequeno possa conter apenas uma classe). Isto é, um programa Java é uma coleção de uma ou mais classes. Todo o conteúdo de uma classe deve estar dentro de um par de chaves `{ }`.

```
public class Cachorro {  
}  
  
classe
```

##### O que existe em uma classe?

Uma classe pode conter atributos e métodos. A classe Cachorro, exibida ao lado, possui os atributos *idade* e *nome* (por enquanto, pense nos atributos como variáveis). Já o método *latir* conterá instruções de como o cachorro deve latir. Os métodos e atributos devem ser declarados dentro da classe, ou seja, dentro do par de chaves da classe.

```
public class Cachorro {  
    int idade;  
    String nome;  
  
    void latir() {  
    }  
  
atributos e método
```

##### O que existe em um método?

Cada método é também delimitado por um par de chaves. Dentro das chaves do método estarão as instruções de como ele deve ser executado. O código do método é basicamente um conjunto de instruções, atribuições, laços de repetição, decisão, comandos de entrada e saída, etc. Por enquanto, pense nos métodos como funções ou procedimentos.

```
public class Cachorro {  
    int idade;  
    String nome;  
  
    void latir(){  
        Instrução 1;  
        Instrução 2;  
        Instrução 3;  
    }  
  
corpo do método
```

## O método main()

Um programa Java deve conter pelo menos uma classe e apenas um método especial chamado `main` (principal). Isto é, se o programa for composto por várias classes, apenas uma delas deve conter o método `main`. O método `main` possui a seguinte estrutura:

```
public static void main (String[] args) {  
    //seu código entra aqui  
}
```

Embora um programa possa conter várias classes, é sempre a classe que contém o método `main` que deverá ser executada na linha de comando. Executar um programa em Java significa informar à JVM para carregar a classe e executar as instruções contidas dentro das chaves de seu método `main()`.

Vamos agora escrever nosso primeiro programa em Java. Digite o programa abaixo em um editor de textos (bloco de notas, gedit) e salve-o com `Exemplo1.java`.

```
public class Exemplo1 {  
    public static void main (String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

**Importante: o arquivo deve ser salvo sempre com o mesmo nome da classe.**

Abra o terminal no Linux (ou o prompt de comando no Windows) e acesse a pasta onde seu arquivo foi salvo. Digite:

```
javac Exemplo1.java
```

O comando `javac` aciona o compilador Java. Se não houver nenhum erro de sintaxe, após alguns segundos será criado, na mesma pasta, um arquivo chamado `Exemplo1.class` e o prompt será exibido novamente. Se for gerado algum erro, retorne para o editor de texto, corrija-o, salve e execute novamente o comando acima.

Agora já temos nosso arquivo compilado em *bytecodes*. Para executá-lo e ver sua saída, digite:

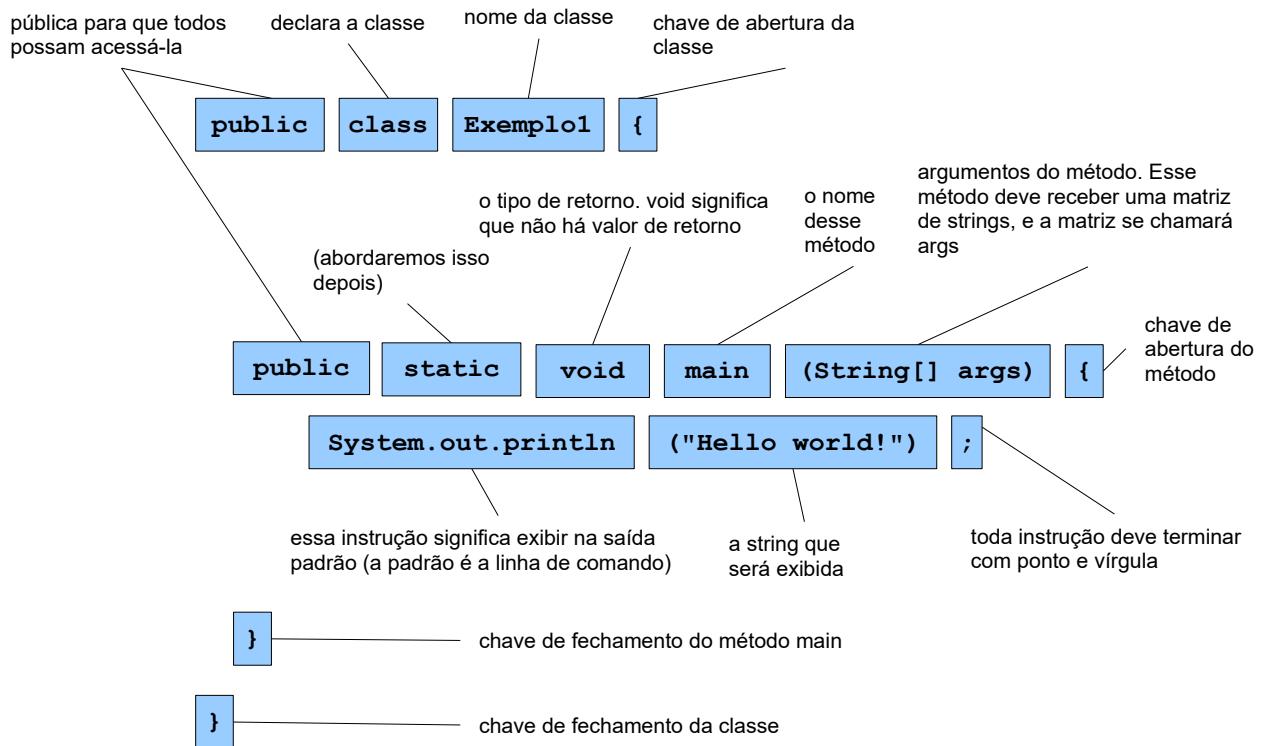
```
java Exemplo1
```

Note que não é necessário especificar a extensão `.class`. O comando `java` aciona a JVM e solicita que seja carregada a classe `Exemplo1`. O código existente dentro do método `main()` será então executado. O resultado deve ser o seguinte:

```
Hello world!
```

É importante destacar que a linguagem Java é *case sensitive*, isto é, diferencia entre letras maiúsculas e minúsculas.

Vamos agora explorar o código acima para entender seus componentes.



O centro de qualquer aplicação Java é seu método `main()`, pois será a partir dele que todos os outros métodos requeridos para executar uma aplicação serão executados. Se o interpretador, por meio da JVM, não encontrar o método `main()`, o programa não será executado.

Observe que usamos os caracteres `{` e `}` para iniciar e finalizar blocos de código.

### 3.2 PARÂMETROS PELA LINHA DE COMANDO

O método `main()` aceita um parâmetro do tipo vetor de Strings. Cada string desse vetor representa um parâmetro passado através da linha de comando no momento da execução do programa com o comando `java`. Dessa forma, é possível acessar dentro do programa os valores recebidos por este meio.

```
public class Exemplo2 {
    public static void main (String[] args) {
        System.out.println("Parâmetro 1: " + args[0]);
        System.out.println("Parâmetro 2: " + args[1]);
        System.out.println("Bem vindo ao Mundo Java!");
    }
}
```

Para compilar o programa acima, proceda normalmente:

```
javac Exemplo2.java
```

Para executar o programa acima proceda da seguinte forma, em uma linha de comando:

```
java Exemplo2 10 20
```

O resultado será:

```
Parâmetro 1: 10
```

```
Parâmetro 2: 20
Bem vindo ao Mundo Java!
```

### 3.3 COMENTÁRIOS

Os comentários são palavras ou frases escritas no interior do código fonte de um programa, mas que são ignorados pelo compilador. Seu propósito é documentar o código para que o programador ou mesmo outros programadores possam entendê-lo com facilidade.

Em Java, podemos criar um comentário de uma única linha através dos caracteres // no início da mesma. Já para comentários cujo texto se estenda para mais de uma linha, devemos colocá-lo entre os caracteres /\* e \*/. Veja os exemplos:

```
// este é um comentário de uma linha
/* este é um
comentário com várias
linhas */
```

### 3.4 IDENTIFICADORES

Cada objeto, classe, atributo ou método deve ter um nome que o identifica, de modo que possam ser referenciados dentro do código.

Os identificadores em java devem seguir as seguintes regras:

- o identificador pode ser composto por letras, dígitos numéricos, cifrão (\$) e o caractere sublinhado;
- o primeiro caractere não pode ser um dígito.

Portanto, não são permitidos espaços em branco ou outros caracteres especiais.

Exemplos de identificadores **válidos**:

<code>codigo</code>	<code>valor1</code>	<code>\$nome</code>
<code>salarioMinimo</code>	<code>valor_1</code>	<code>nome\$</code>
<code>salario_minimo</code>	<code>_nome</code>	<code>código</code>

Exemplos de identificadores **inválidos**:

<code>10Salario</code> (começa com dígito)	<code>Salario Minimo</code> (espaço)
<code>Brasil!</code> (caractere especial)	<code>Salario-Minimo</code> (caractere especial)

É importante destacar que a linguagem Java diferencia letras maiúsculas e minúsculas, isto é, é **case sensitive**. Desta forma, `nome` e `Nome` não representam o mesmo identificador.

Além disso, um identificador não pode ser igual a uma palavra reservada (`public`, `class`, `void`, `return`, `int`, etc.). As palavras reservadas são sempre escritas em minúsculo.

### 3.5 TIPOS DE DADOS PRIMITIVOS

Tipos primitivos são os tipos de dados básicos fornecidos pela linguagem de programação. A partir de tipos primitivos podemos construir outros tipos ou classes.

A tabela a seguir mostra os tipos de dados primitivos da linguagem Java.

Tipo	Tamanho	Valores válidos
boolean	8 bits	true ou false
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
float	32 bits	-3,4028234663852886E+38 a 3,4028234663852886E+38 (com 9 dígitos de precisão)
double	64 bits	-4,94065645841246544E-324 a 4,94065645841246544E-324 (com 18 dígitos de precisão)
char	16 bits caractere unicode	'\u0000' a '\uFFFF' – 0 a 65.535

### 3.6 SEQUÊNCIA DE CARACTERES (STRING)

Em Java, uma sequência de caracteres é tratada através de um objeto pertencente à classe String. A classe String é uma classe da biblioteca (API) Java, podendo ser utilizada na construção de qualquer outra classe.

Mesmo sendo um objeto da classe String, uma sequência de caracteres pode ser manipulada de maneira similar a uma variável:

```
String nome;
nome = "Universidade Federal da Fronteira Sul";
```

As strings são sempre incluídas entre aspas duplas. Se você esquecer de colocar aspas duplas, ao compilar seu programa será exibida a mensagem "*unclosed string literal*".

Também é possível instanciar um objeto da classe String, ativando diretamente o construtor.

```
String nome = new String("Minha string");
```

### 3.7 TIPOS DE DADOS REFERÊNCIA

Toda classe criada pelo programador (ou integrante da API Java) representa um tipo de dado diferente dos tipos primitivos já apresentados.

Por exemplo, se criarmos uma classe Aluno, esta representará um novo tipo de dados, o tipo Aluno. Poderemos a partir daí declarar variáveis do tipo Aluno. Estas variáveis são denominadas variáveis de referência. Usaremos variáveis de referência para armazenar referências a objetos em nossos programas, como veremos adiante.

### 3.8 CRIAÇÃO E USO DE VARIÁVEIS

Uma variável representa um local de memória que armazenará um valor e é sempre declarada como sendo de um determinado tipo, primitivo ou de referência.

A forma mais simples de declarar uma variável é através do formato

```
tipo identificador;
```

As variáveis em Java devem ser sempre inicializadas. Se declararmos uma variável e não a inicializarmos, ao tentar utilizá-la teremos o erro "*Variable <identificador> might not have been initialized*".

Para inicializar uma variável, basta atribuir-lhe um valor compatível com seu tipo:

```
identificador = valor;
```

É possível realizar as duas ações, declarar e inicializar uma variável, em uma mesma linha de comando:

```
tipo identificador = valor;
```

Também podemos declarar várias variáveis do mesmo tipo em um único comando, separando seus identificadores por vírgulas:

```
tipo identificador1, identificador2,..., identificadorN;
```

ou

```
tipo identificador1=valor1, identificador2=valor2,..., identificadorN=valorN;
```

- Exemplos de criação e uso de variáveis de tipos primitivos:

```
int idade;
idade = 25;
int x, y;
x = 5;
y = 2 * x;
float salario;
char letra = 'a';
```

- Exemplos de criação de variáveis de referência:

```
String s1;
Aluno a1;
// Criação e inicialização:
String s2 = new String("Minha string");
Float f = new Float(2.464566);
Aluno a2 = new Aluno();
```

## 3.9 OPERADORES

### 3.9.1 Concatenação

O operador `+`, além de ser um operador aritmético (como veremos logo a seguir), também serve para concatenar cadeias de caracteres (strings). Já vimos seu uso anteriormente:

```
System.out.println("Parâmetro 1: " + args[0])
```

No exemplo acima, estamos imprimindo a string "Parâmetro 1: " e logo na sequência, o valor da variável `args[0]`. O operador `+` serve para juntar estas duas informações.

Podemos usar este operador também para atribuir valores de strings a variáveis:

```
nome = "Ana";
nome = nome + " Maria";
sobrenome = "Santos";
completo = nome + " " + sobrenome;
```

Será impresso Ana Maria Santos.

### 3.9.2 Aritméticos

Representam as operações básicas da matemática.

Operador	Uso	Descrição
+	a + b	Adição
-	a - b	Subtração
*	a * b	Multiplicação
/	a / b	Divisão. A divisão entre dois números inteiros resulta em outro número inteiro. Ex: 11/3 = 3
%	a % b	Módulo. O operador % retorna o resto da divisão inteira. Ex: 11%3 = 2

```
// Exemplo do uso dos operadores aritméticos
public class OperadoresAritmeticos {
    public static void main(String args[]) {
        int var1 = 9;
        int var2 = 2;
        System.out.println("var1 = " + var1);
        System.out.println("var2 = " + var2);
        System.out.println("-var2 = " + (-var2)); //imprime como valor negativo
        System.out.println("var1 + var2 = " + (var1 + var2));
        System.out.println("var1 - var2 = " + (var1 - var2));
        System.out.println("var1 * var2 = " + (var1 * var2));
        System.out.println("var1 / var2 = " + (var1 / var2));
        System.out.println("(float) var1/ var2 = " + ((float) var1 / var2));
        System.out.println("var1 % var2 = " + (var1 % var2));
    }
}
```

### 3.9.3 Unários

Os operadores unários realizam as operações básicas levando em consideração apenas uma variável.

Operador	Uso	Descrição
++	a++	Incremento de uma unidade. Quando se coloca o operador após a variável, o incremento será feito após o uso do valor da mesma na expressão (usa-se o valor e depois ele é incrementado).
	++a	Quando se usa o operador antes da variável, o incremento será feito antes do uso do valor da mesma na expressão (incrementa-se primeiro e depois se usa o valor).

--	a--	Decremento de uma unidade.
	--a	Quando se usa o operador antes da variável, o decremento será feito antes do uso do valor da mesma na expressão (incrementa-se primeiro e depois se usa o valor).

```
// Exemplo do uso dos operadores unários
public class ExemploOperadorUnario {
    public static void main(String args[]) {
        int var1 = 10;
        int var2 = 20;
        int res = 0;
        res = var1 + var2;
        System.out.println("res: " + res); // imprime 30
        // após a execução do operador = será executado o operador ++ de var1
        res = var1++ + var2; // var1 vai valer 11 após a execução do operador =
        System.out.println("res: " + res); // imprime 30
        res = var1 + var2;
        System.out.println("res: " + res); // imprime 31
        res = var1 + --var2;
        System.out.println("res: " + res); // imprime 30
    }
}
```

### 3.9.4 Relacionais

Os operadores relacionais realizam as operações de comparação entre valores ou variáveis, e retornam verdadeiro ou falso. São usados nas estruturas de decisão (IF) e repetição (WHILE, DO WHILE e FOR).

Operador	Uso	Descrição
==	a == b	Igualdade
!=	a != b	Desigualdade
<	a < b	Menor
<=	a <= b	Menor ou igual
>	a > b	Maior
>=	a >= b	Maior ou igual

```
// Exemplo do uso dos operadores relacionais
public class OperadoresRelacionais {
    public static void main(String abcd[]) {
```

```

int var1 = 27;
int var2 = 74;
System.out.println("var1 = " + var1);
System.out.println("var2 = " + var2);
System.out.println("var1 == var2 -> " + (var1 == var2));
System.out.println("var1 != var2 -> " + (var1 != var2));
System.out.println("var1 < var2 -> " + (var1 < var2));
System.out.println("var1 > var2 -> " + (var1 > var2));
System.out.println("var1 <= var2 -> " + (var1 <= var2));
System.out.println("var1 >= var2 -> " + (var1 >= var2));
}
}

```

### 3.9.5 Lógicos

Os operadores lógicos são muito usados em estrutura de controle e loop e retornam um valor booleano (true ou false).

Operador	Uso	Descrição
&&	a && b	"E" lógico a e b verdadeiros, avalia condicionalmente b (se a é falso, não avalia b).
	a    b	"OU" lógico a ou b verdadeiro, avalia condicionalmente b (se a é verdadeiro, não avalia b).
!	!(a < b)	Negação lógica
^	a ^ b	"OU" exclusivo

```

// Exemplo do uso dos operadores lógicos
public class OperadoresLogicos {
    public static void main(String abcd[]) {
        boolean var1 = true;
        boolean var2 = false;
        System.out.println("var1 = " + var1);
        System.out.println("var2 = " + var2);
        System.out.println("NOT var1 -> " + (! var1));
        System.out.println("NOT var2 -> " + (! var2));
        System.out.println("var1 AND var2 -> " + (var1 && var2));
        System.out.println("var1 OR var2 -> " + (var1 || var2));
        System.out.println("var1 XOR var2 -> " + (var1 ^ var2));
    }
}

```

### 3.9.6 Atribuição

Os operadores de atribuição são usados para atribuir um novo valor a uma variável.

Operador	Uso	Equivale a:
=	a = b	Atribuição
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

## 3.10 COMANDOS DE ENTRADA

Os comandos de entrada disponíveis nas primeiras versões da linguagem Java representavam uma grande dificuldade para o desenvolvedor. Somente na versão 5.0 é que o problema foi resolvido, com a inclusão da classe Scanner, disponível no pacote `java.util`. Ou seja, para usar essa classe devemos usar o comando `import java.util.Scanner` antes da criação da classe.

A classe Scanner permite a leitura de tipos de dados primitivos e strings, lidos a partir do teclado ou de um arquivo.

```
import java.util.Scanner;

class ExemploScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Digite seu nome completo: ");
        String name = scanner.nextLine();

        System.out.print("Digite seu signo: ");
        String zodiac = scanner.next();

        System.out.print("Digite seu peso (kg): ");
        double weight = scanner.nextDouble();

        System.out.print("Digite seu número da sorte: ");
        int luckyNum = scanner.nextInt();

        System.out.println("\n\nOlá, " + name + "!");
        System.out.println("Seu número da sorte é " + luckyNum + ".");
        System.out.println("Você pesa " + weight + " kg.");
        System.out.println("Seu signo é " + zodiac + ".\n");
    }
}
```

O parâmetro `System.in` representa a entrada padrão, que será o teclado, caso não seja informado nenhum nome de arquivo na linha de comando.

### 3.11 COMANDOS DE SAÍDA

Nos exemplos já apresentados usamos diversas vezes o comando `System.out.println("texto qualquer")` para apresentar um determinado resultado na tela. Esse comando é o mais usado para apresentar dados na tela quando o programa Java é do tipo batch. Este exemplo usa o atributo estático `out`, definido na classe `System`, que por sua vez executa o método `println()`.

Para que o cursor não pule uma linha após imprimir o texto na tela, use `print()` no lugar de `println()`. Para exibir uma saída formatada, utilize `printf` (conforme linguagem C).

### 3.12 ESTRUTURAS DE DECISÃO

#### 3.12.1 If

Os comandos `if` e `else` definem de forma seletiva qual bloco de comandos será executado. Se a condição do comando `if` for avaliada como verdadeira, será executado o bloco de comandos dentro do `if`. Caso contrário, o bloco de comandos dentro do `else` será executado.

Sintaxe do comando IF/ELSE	Representação gráfica
<pre>if (condição) {     bloco de comandos 1; } else {     bloco de comandos 2; }</pre>	<pre> graph TD     Inicio[Início] --&gt; Cond{Condição}     Cond -- V --&gt; Bloco1[Bloco de comandos 1]     Cond -- F --&gt; Bloco2[Bloco de comandos 2]     Bloco1 --&gt; Fim[Fim]     Bloco2 --&gt; Fim   </pre>

Veja abaixo o exemplo do uso simplificado do comando `if` (sem `else`):

```
// Exemplo do comando if
public class ExemploIf {
    public static void main(String args[]) {
        int var1 = 20;
        int var2 = 10;
        if (var1 > var2) {
```

```
// bloco de comandos do if
System.out.println("var1 é maior que var2");
}
}
}
```

**Importante:** a condição deve ser sempre expressa entre parênteses.

A seguir, um exemplo do uso do if com else.

```
// Exemplo comando if e do comando else
public class ExemploIfElse {
    public static void main(String args[]) {
        int var1 = 10;
        int var2 = 20;
        if (var1 > var2) {
            // bloco de comandos do if
            System.out.println("var1 é maior que var2");
        } else { // condição avaliada como falso
            // bloco de comandos do else
            System.out.println("var1 é menor que var2");
        }
    }
}
```

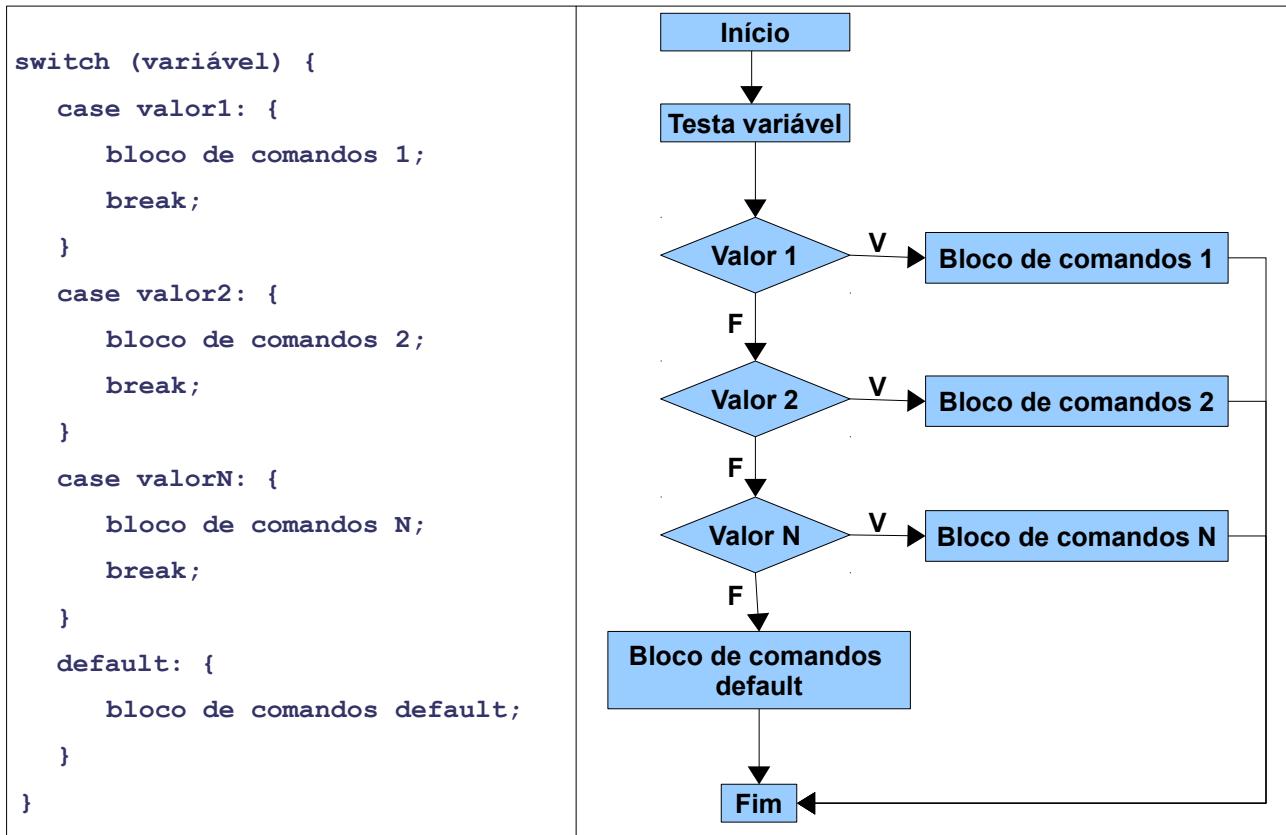
### 3.12.2 Switch

O comando `switch` também realiza a execução de um bloco de comandos de acordo com uma decisão. Devemos optar pelo `switch` quando, no teste realizado, usamos uma mesma variável, igualando-a com vários valores diferentes.

Quando usamos o comando `switch` devemos também usar em conjunto o comando `case`, que com base no valor da variável do comando `switch` define qual opção será executada. Para que somente um entre vários comandos `case` seja executado, devemos executar o comando `break`, logo após a execução dos comandos contidos no bloco do comando `case` selecionado. O comando `break` interrompe a execução do `case`.

Há também o comando `default`, que representa uma exceção a todas as opções listadas nos comandos `case`. É importante destacar que no comando `switch` só são aceitas variáveis do tipo `int` ou `char`. O uso de outro tipo primitivo gera um erro de compilação.

Sintaxe do comando SWITCH	Representação gráfica
---------------------------	-----------------------



```

import java.util.Scanner;
public class ExemploSwitch {
    public static void main(String args[]) {
        Scanner s = new Scanner(System.in);
        System.out.println("Digite um número de 1 a 7: ");
        int dia = s.nextInt();
        switch (dia) {
            case 1: {
                System.out.println("Domingo");
                break;
            }
            case 2: {
                System.out.println("Segunda-feira");
                break;
            }
            case 3: {
                System.out.println("Terça-feira");
                break;
            }
            case 4: {
                System.out.println("Quarta-feira");
            }
        }
    }
}

```

```

        break;
    }
    case 5: {
        System.out.println("Quinta-feira");
        break;
    }
    case 6: {
        System.out.println("Sexta-feira");
        break;
    }
    case 7: {
        System.out.println("Sábado");
        break;
    }
    default: {
        System.out.println("Número inválido! Informe um valor de 1 a
7");
    }
}
}
}

```

### 3.13 ESTRUTURAS DE REPETIÇÃO

#### 3.13.1 For

A estrutura de repetição **for** executa um bloco de comandos um número pré-definido de vezes. Abaixo, temos a sintaxe do comando for, bem como sua representação por fluxograma.

Sintaxe do comando FOR	Representação gráfica
<pre> for (inicialização; condição de fim;       incremento) {     bloco de comandos; } </pre>	<pre> graph TD     Start((Início)) --&gt; Init[Inicialização das variáveis]     Init --&gt; Cond{Condição de fim}     Cond -- V --&gt; Block[Bloco de comandos]     Block --&gt; Incremento[Incremento]     Incremento --&gt; Cond     Cond -- F --&gt; End[Fim] </pre>

```
// Exemplo do comando for
```

```
public class ExemploFor {
    public static void main(String[] args) {
        for (int var1 = 0; var1 < 27; var1++) {
            System.out.println(var1);
        }
    }
}
```

### 3.13.2 While

A estrutura de repetição **while** executa um bloco de comandos enquanto uma dada condição for verdadeira, realizando o teste da condição no início. Isto é, se a condição for falsa logo de início, o bloco de comandos não será executado nenhuma vez.

Sintaxe do comando WHILE	Representação gráfica
<pre>while (condição) {     bloco de comandos; }</pre>	<pre> graph TD     Inicio[Início] --&gt; Condicao{Condição}     Condicao -- V --&gt; Bloco[Bloco de comandos]     Bloco --&gt; Condicao     Condicao -- F --&gt; Fim[Fim]   </pre>

```
// Exemplo do comando while
public class ExemploWhile {
    public static void main(String[] args){
        System.out.println("While: ");
        int num = 1;
        while (num <= 10) {
            System.out.println(num);
            num++;
        }
    }
}
```

### 3.13.3 Do while

A estrutura de repetição **do while** executa um bloco de comandos enquanto uma dada condição for verdadeira, realizando o teste da condição no final. Isso garante que o bloco de comandos será executado pelo menos uma vez, mesmo que a condição seja sempre falsa.

Sintaxe do comando DO WHILE	Representação gráfica
<pre>do {     bloco de comandos; } while (condição);</pre>	<pre> graph TD     Inicio[Início] --&gt; Bloco[Bloco de comandos]     Bloco --&gt; Cond{Condição}     Cond -- V --&gt; Bloco     Cond -- F --&gt; Fim[Fim]     </pre>

```
// Exemplo do comando while
public class ExemploDoWhile {
    public static void main(String[] args){
        System.out.println(" Do While: ");
        int num = 11;
        do {
            System.out.println(num);
            num++;
        }
        while (num <= 10);
    }
}
```

### 3.14 BREAK E CONTINUE

O comando break tem a função de interromper a execução de um laço de repetição.

```
// Exemplo do comando break
public class ExemploBreak {
    public static void main(String args[]) {
        for (int i = 0; i <= 30; i++) {
            if (i == 10) {
                break;
            }
            System.out.println("i = " + i);
        }
    }
}
```

Já o comando `continue` tem a função de fazer com que a condição do loop seja novamente testada, mesmo antes de alcançar o fim do comando.

```
// Exemplo do comando continue
public class ExemploContinue {
    public static void main(String args[]) {
        for (int i = 0; i <= 30; i++) {
            if ((i > 10) && (i < 20)) {
                continue;
            }
            // mostra na tela quando o i não estiver entre 10 e 20
            System.out.println("i = " + i);
        }
    }
}
```

### 3.15 ARRAYS

Arrays são objetos que contém um número fixo de valores de um determinado tipo. O tamanho do array é informado quando ele é instanciado e, depois que isso é feito, seu tamanho não pode mais ser alterado. Cada item dentro do array é chamado de *elemento* e cada um deles pode ser acessado através de um índice numérico. A numeração dos índices inicia no zero.

Um array é declarado da seguinte forma:

```
int[] meuArray;
```

A declaração de um array segue o mesmo princípio da declaração de uma variável, que é o tipo seguido do nome, como `float a` ou `int t`. A diferença é que o tipo do array é escrito no formato `tipo[]`, onde *tipo* define o tipo dos elementos do array e os colchetes `[]` indicam que aquela variável é um array.

Da mesma forma como as variáveis de outros tipos, a simples declaração não cria o array. Para isso, é preciso explicitamente instanciar o array com o tamanho desejado:

```
int[] meuArray;
meuArray = new int[10]; // instancia um array de 10 inteiros.
MeuArray[0] = 23;      // atribui 23 ao primeiro elemento do array.
MeuArray[1] = 576;     // atribui 576 ao segundo elemento do array
...
MeuArray[9] = 34;      // atribui 34 ao último elemento do array.
```

Também é possível instanciar um array no momento da sua criação:

```
int[] arrayCriado = {23, 434, 543, 65, 0, 98};
```

No exemplo acima, o tamanho do array é calculado pelo compilador. A declaração acima é equivalente a seguinte:

```
int[] arrayCriado;
arrayCriado = new int[6];
arrayCriado[0] = 23;
arrayCriado[1] = 434;
arrayCriado[2] = 543;
arrayCriado[3] = 65;
arrayCriado[4] = 0;
arrayCriado[5] = 98;
```

O tamanho de um array qualquer pode ser encontrado através da propriedade *length* de qualquer objeto array, como no exemplo:

```
int[] meuArray;
meuArray = new int[6];
System.out.println("Tamanho = " + meuArray.length); //Imprime "Tamanho = 6"
```

A propriedade *length* é muito útil para que possamos iterar sobre os elementos de um determinado array:

```
int[] arrayCriado = {23, 434, 543, 0};
int i;
for(i = 0; i < arrayCriado.length; i++) {
    System.out.println("Elemento" + i + " = " + arrayCriado[i]);
}
```

O programa acima imprime o seguinte na tela:

```
Elemento 0 = 23
Elemento 1 = 434
Elemento 2 = 543
Elemento 3 = 0
```

### 3.16 RESUMO DO CAPÍTULO

- Todo programa em Java é composto por um conjunto de classes, devendo existir uma classe contendo um método especial denominado main.
- Quando for solicitada a execução de um programa Java, a JVM executará o grupo de instruções que compõe o método main.
- Toda instrução deve finalizar com ponto-e-vírgula (;).
- Toda classe deve ser salva em um arquivo cujo nome deve ser o mesmo da classe e cuja extensão deve ser .java.
- Um grupo de zero ou mais instruções especificadas internamente aos caracteres { } é denominado bloco.

- Comentários de uma linha devem ser precedidos por `//`.
- Comentários de múltiplas linhas devem estar entre `/*` e `*/`.
- Um identificador pode ser formado por letras, dígitos e os caracteres `$` e `_`. O primeiro caractere não pode ser um dígito.
- Java é *case sensitive*.

### **3.17 BIBLIOGRAFIA DO CAPÍTULO**

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

BORATTI, Isaias Camilo. **Programação orientada a objetos em Java**. Florianópolis: Visual Books, 2007.

SIERRA, Kathy; BATES, Bert. **Use a Cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2007.

## 4 Implementação de classes em Java

No capítulo anterior escrevemos pequenos programas em Java. Tais programas foram apenas introdutórios e não apresentaram a construção e utilização de nenhum objeto. Vimos que todo programa em Java é composto por um conjunto de uma ou mais classes e que deve existir uma classe contendo um método denominado `main`. A partir de agora, começaremos a desenvolver programas que envolvam a construção e manipulação de objetos.

### 4.1 CLASSES E OBJETOS

Ao escrever um programa de computador em uma linguagem orientada a objetos, você criará, em seu computador, um modelo de alguma parte do mundo real denominada *domínio do problema*. As partes que compõe esse modelo são as representações dos objetos que aparecem no domínio do problema.

Um **objeto** é uma abstração de uma entidade, concreta ou abstrata, que tenha sua própria existência, características e que apresente alguma função no mundo real. Exemplos de objetos são:

- a aluna Maria
- o aluno João
- o livro O Senhor dos Anéis
- uma viagem
- um mouse
- uma compra em uma loja
- um aparelho de DVD
- a disciplina de Programação I

Cada um dos itens mencionados acima representa algo no mundo real, seja algo concreto (como a aluna Maria ou um mouse) ou abstrato (como uma viagem ou uma compra). Cada um destes objetos possui características próprias, que os distinguem de outros objetos.

Os objetos contém **atributos**, que representam suas características. Por exemplo, o objeto aluna Maria possui os atributos nome, idade, matrícula, endereço, etc. O objeto livro possui título, autor, ano, número de páginas, etc. Uma compra possui uma data, um cliente (que por sua vez é outro objeto), os produtos adquiridos, etc. Podemos dizer que os atributos são as informações que cada objeto conhece sobre si mesmo. Os valores dos atributos definem o estado de um objeto. Quando o valor de um atributo é alterado, dizemos que houve uma mudança de estado.

Além dos atributos, um objeto pode possuir **métodos** que definem seu comportamento. Por exemplo, o objeto aluna Maria pode conter, no domínio de uma instituição de ensino, os métodos *matricular-se*, *frequentar aula*, *realizar prova*, etc. O objeto livro tem os métodos *abrir*, *fechar*, *avançar página* e *voltar página*. O aparelho de DVD possui os métodos *ligar*, *desligar*, *ejetar disco*, *executar disco*, etc.



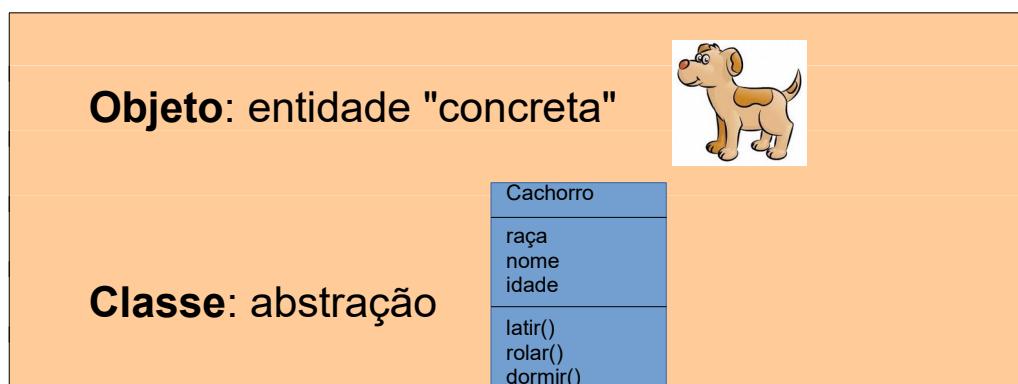
Figura 11: Um objeto reúne atributos e métodos em sua estrutura.

Ao observar objetos, podemos perceber que eles muitas vezes compartilham atributos e métodos comuns. Por exemplo, um Gol e um Vectra são ambos carros e, embora diferentes na aparência, possuem 4 rodas, uma cor, um proprietário, um número da placa, um número do chassi, dentre outras características; além disso, ambos podem frear, acelerar, trocar marcha, dar a partida e desligar. Assim, objetos com características semelhantes podem ser agrupados em categorias (ou classes).

Uma **classe** descreve de maneira abstrata todos os objetos de um tipo particular. Ou seja, uma classe define um modelo para a criação de um tipo específico de objetos. Assim, uma classe encapsula (reúne) num único componente dois tipos de **membros**:

- **Atributos** (ou campos, ou variáveis de instância) – representam os dados que cada objeto da classe irá guardar. Cada objeto terá um valor específico para cada atributo.
- **Métodos** (ou funções) – conjuntos de comandos (instruções) que definem as operações que podem ser feitas pelos objetos da classe.

É importante destacar que uma classe não representa nenhum objeto em particular, e sim um modelo para a criação dos objetos. Lembre-se:



É bastante comum aos iniciantes em programação orientada a objetos (POO) confundir os termos classe e objeto.

Para esclarecer os conceitos, vamos fazer uma analogia. Uma receita de bolo e um bolo são a mesma coisa? Obviamente não. A receita é uma descrição de como um bolo deve ser feito (ingredientes e modo de preparo). Podemos comparar a receita a uma classe: as quantidades de cada ingrediente são os atributos e o modo de preparo são os métodos. Já um bolo é um objeto criado a partir de uma receita, ou seja, uma realização prática da receita. Podemos fazer vários bolos com a mesma receita (bolos redondos, quadrados, com cobertura, sem cobertura, com recheio, sem recheio, etc), da mesma forma que podemos criar vários objetos a partir de uma mesma classe.

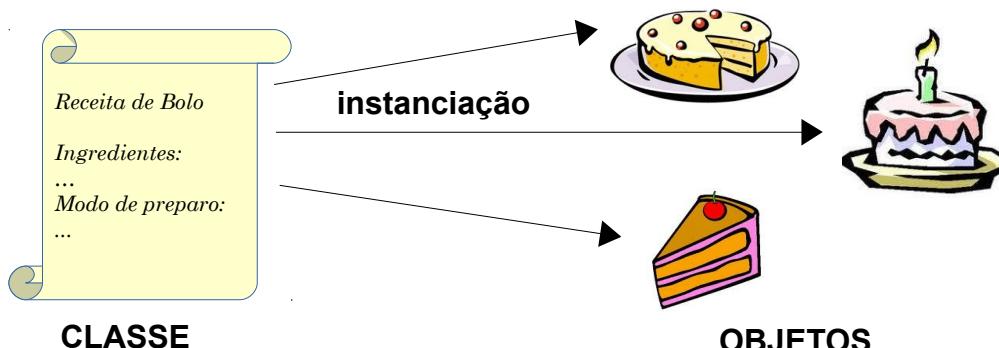


Figura 12: Analogia de classes e objetos.

- Os objetos são criados a partir de classes;
- A classe descreve o tipo do objeto, quais atributos e quais métodos ele possui;
- Os objetos representam instâncias individuais de uma classe, terão um valor para cada atributo e poderão executar os métodos definidos na mesma.

Uma classe pode ser representada graficamente por um retângulo dividido em três partes, contendo respectivamente seu nome, seus atributos e seus métodos. Essa representação está de acordo com a linguagem de notação orientada a objetos UML (*Unified Modeling Language*), empregada nas etapas de análise e projeto de um sistema orientado a objetos.

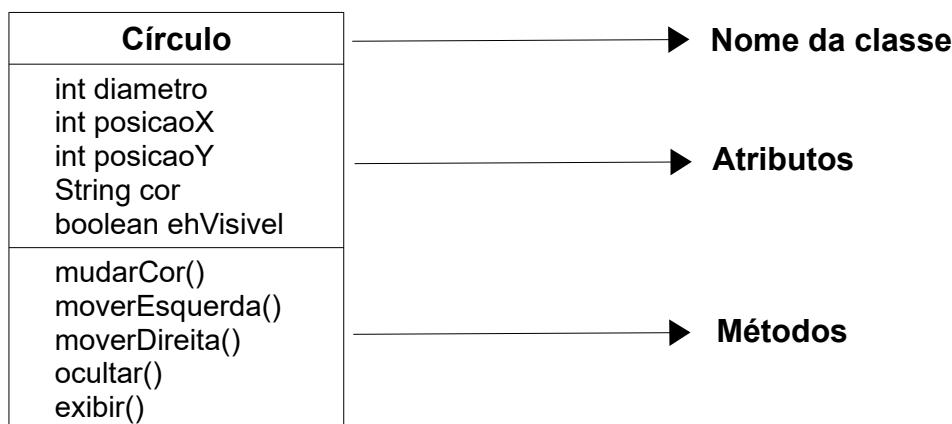


Figura 13: Representação gráfica de uma classe.

#### 4.1.1 Criando Classes

Uma classe pode ser escrita (declarada) de acordo com a seguinte forma geral:

```
modificadores class Nome_da_classe {
    corpo da classe - especificação de atributos e métodos.
}
```

O corpo da classe sempre inicia com `{` e sempre finaliza com `}`. Entre as chaves, uma classe pode definir somente atributos, métodos e códigos de inicialização.

Toda classe tem um nome. Por convenção, o nome da classe inicia com letra maiúscula. Em geral, um nome de classe é um substantivo ou locução substantiva (dois ou mais substantivos juntos). Neste caso, convenciona-se escrever todas as palavras juntas, com somente as iniciais em maiúsculo.

Exemplos de nomes de classe: Veiculo, ContaCorrente, ApoliceSeguro, etc.

```
public class Funcionario {
    // aqui vão os atributos
    private String nome;
    private double salario;
    private int numeroDeDependentes;
    // aqui vão os métodos:
    public void aumento(double valorAumento) {
        salario = salario + valorAumento;
    }
}
```

```

    }
}

```

#### 4.1.2 Criando Objetos

A definição de uma classe não implica na existência de um objeto desta classe. Um objeto só passa a existir se for explicitamente construído.

Uma vez que tenhamos uma classe, podemos criar tantos objetos (instâncias) desta classe quanto quisermos. Com base na classe Funcionário, por exemplo, podemos criar vários objetos funcionário; com base na classe Livro, pode-se criar vários objetos livro. Cada um destes objetos terá seu próprio título, seu autor, seu ano e seu número de páginas.

Objetos são instâncias de classes. Ao serem criados, possuem uma identificação única que é atribuída pelo ambiente de execução (OID), além de conter todos os atributos e métodos definidos em sua classe. Estes atributos assumem valores que compõem o estado do objeto.

A criação de um novo objeto em Java é feita em três etapas:

1. declarar uma variável cujo tipo é a classe da qual o objeto será uma instância;
2. criar uma nova instância para o objeto, usando o operador new, que aloca espaço de memória o novo objeto;
3. atribuir o endereço do objeto recém-criado à variável objeto.

Ex:

```

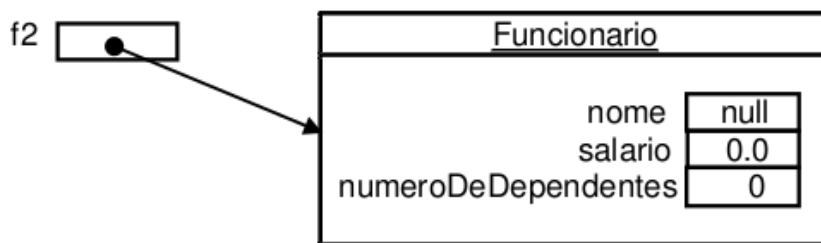
// declara a variável de referência de nome f2 do tipo Funcionario (etapa 1):
Funcionario f2;

// aloca área de memória para o novo objeto e atribui a f2 o endereço dessa
área (etapas 2 e 3)

f2 = new Funcionario();

```

Na memória, temos:



O operador `new` cria um novo objeto e já inicializa automaticamente seus campos com valores default (padrão). Os valores padrão são 0, para os tipos numéricos, e null, para o tipo String.

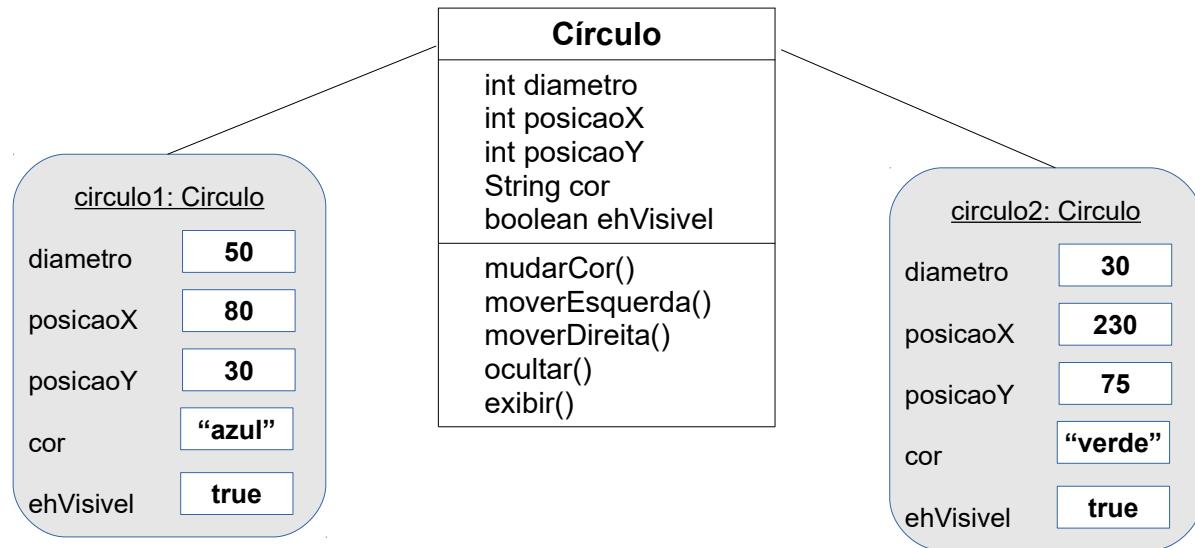
As três etapas podem ser reunidas em uma só instrução

```
Funcionario f2 = new Funcionario();
```

De forma geral, a criação de um objeto na linguagem Java pode ser feita por meio do comando:

```
Classe variável = new Classe();
```

Objetos da mesma classe terão todos os mesmos atributos. O que pode variar de um objeto para outro são os valores dos atributos, mas o tipo e nome dos atributos será sempre o mesmo, da mesma forma que os métodos. Isso se deve ao fato que atributos e métodos são definidos na classe e não nos objetos.



Um objeto só pode ser manipulado, isto é, só pode receber uma mensagem se o mesmo tiver sido declarado e construído. Construir um objeto implica em alocar espaço na memória para ele e inicializar seus atributos. Para que um objeto possa ser explicitamente construído, sua classe deve possuir um método construtor.

Assumindo que na classe Círculo tem-se declarado um construtor, a execução do comando:  
`circulo1 = new Círculo();`

fará com que seja executado o método construtor Círculo(), criando o objeto circulo1. Além de alocar espaço na memória para o objeto, um método construtor define qual a sequência de comandos a ser executada pelo computador, quando da construção do objeto.

## 4.2 ATRIBUTOS

Conforme visto, os atributos descrevem as características dos objetos. Um atributo pode ser um valor ou outro objeto. Se for um valor, é uma variável de tipo primitivo (int, double, float, etc). Se for outro objeto, é uma variável de referência.

Por convenção, nomes de atributos iniciam com letra minúscula. Caso se trate de uma locução substantiva (expressão composta por duas ou mais palavras), as iniciais das demais palavras devem iniciar em maiúsculo.

Ex: `nomeCliente, saldoConta, numeroConta, numeroAgencia`

O conjunto de valores de todos os atributos que definem um objeto é também referido como estado do objeto. Alguns métodos quando chamados alteram atributos e consequentemente, o estado dos objetos.

Um atributo pode ser acessado diretamente pelo seu nome dentro da classe à qual pertence.

```
public class Camiseta {
    String cor;
    char tamanho;
    float preco;
    public void imprimir() {
        cor = "verde";
        System.out.println(cor);
    }
}
```

```
}
```

Para acessar um atributo a partir de outra classe, é necessário primeiramente criar um objeto da classe onde está o atributo. Após a criação, se o atributo for público, pode-se acessá-lo pela identificação do objeto seguida de um ponto seguido do nome do atributo. Exemplo:

```
Camiseta objeto = new Camiseta();
objeto.cor = "verde";
```

Já se o atributo for privado (private), somente podemos acessá-lo ou alterá-lo através de métodos de acesso e métodos modificadores (como recomendado pelo encapsulamento).

```
Camiseta objeto = new Camiseta();
objeto.setCor("verde"); // método modificador
```

Em qualquer dos casos acima, a alteração afetará este objeto em particular, mas não os outros.

### 4.3 MÉTODOS

Podemos nos comunicar com objetos invocando seus métodos. Um método consiste numa sequência de instruções que serão executadas por um objeto quando este receber uma solicitação para tal. Utilizando a terminologia comum, dizemos que os métodos são chamados ou invocados.

Os métodos de uma classe representam as operações que a mesma contém. Tais operações usam os atributos da classe, seja para alteração ou apenas para leitura, processam e geram resultados. Um objeto só sabe executar os métodos que estão definidos em sua classe.

Por convenção iniciamos o nome de um método com letra minúscula e sendo um verbo no infinitivo (terminado em ar, er, e ir) seguido de um substantivo.

Ex: `calcularPedido()`, `arquivarDocumento()`, `imprimirFichaCadastral()`, `efetuarSaque()`, `efetuarDeposito()`.

A forma geral para se declarar um método é:

```
modificadores tipo_de_retorno nome (lista de parâmetros) {
    //instruções
}
```

onde :

- *modificadores*: indicam a visibilidade do método. Em geral, os métodos devem ser public se quisermos permitir que outras classes utilizem nossa classe, o que é recomendável na maioria dos casos.
- *tipo de retorno*: alguns métodos devolvem um resultado a quem o acionou (chamou), cujo tipo deve ser indicado (como os que já conhecemos, int, String, double, ou outro tipo qualquer, que veremos mais adiante, podendo ser, inclusive, um objeto); outros métodos não devolvem nenhum valor específico, caso em que a palavra void deve ser usada em lugar do tipo de retorno. Para métodos construtores, não se deve especificar nenhum retorno.
- *lista de parâmetros*: alguns métodos são escritos de maneira que precisem de

algumas informações adicionais para serem executados. Esses valores adicionais são chamados de **parâmetros** e ficam no interior do par de parênteses. Cada parâmetro deve ter um tipo e um nome, como, por exemplo (String nome, int quantidade, double taxa) . Mesmo que a lista de parâmetros seja vazia, o par de ( ) deve aparecer. Os parâmetros comportam-se como variáveis locais dentro do método.

- As *chaves* { } delimitam o bloco de código com as instruções.

O cabeçalho do método chama-se **assinatura**. Além de fornecer as informações necessárias para invocar esse método, a assinatura garante que não há dois métodos iguais no programa. É possível codificar dois métodos com o mesmo nome, mas com assinaturas diferentes.

Os elementos do método que fazem parte da sua **assinatura** são:

- o nome do método;
- a quantidade de parâmetros existentes;
- o tipo de cada parâmetro;
- a ordem destes parâmetros.

Obs.: o tipo de retorno de um método não faz parte de sua assinatura.

Dentro dos métodos podemos criar variáveis, mas estas têm validade (escopo) somente para uso dentro do método que as criou. Não devemos confundir variáveis do método com os atributos, que são criados logo abaixo da definição da classe. Os atributos não pertencem a nenhum método específico, logo, podem ser acessados por qualquer um deles.

#### 4.4 MENSAGENS

Uma mensagem é a forma de comunicação entre objetos. As mensagens são usadas para ativar métodos nos objetos. Para isso, é necessário criar um objeto, identificar o nome do método a ser executado e, caso necessário, identificar também os parâmetros que o método recebe ou retorna. De posse destes dados, podemos enviar uma mensagem da seguinte forma:

```
objeto1.efetuarSaque(100)
```

neste exemplo, estamos enviando uma mensagem para que o objeto objeto1 execute seu método **efetuarSaque**, com um parâmetro 100.

Para que um objeto possa receber uma mensagem, isto é, receber a solicitação de execução de determinado método, os seguintes requisitos devem ser satisfeitos:

- o objeto que recebe a mensagem deve existir, isto é, deve ter sido construído;
- o respectivo método deve estar definido na classe do objeto que recebe a mensagem.

#### 4.5 UM PROGRAMA COM OBJETOS

Seja o seguinte problema:

"Escrever um programa que determine a área de um círculo".

Inicialmente, devemos analisar um problema com o intuito de identificar os objetos que o integram. A partir da identificação dos objetos, podemos definir que características os mesmos devem apresentar para que seja possível a resolução do problema. As características de um

objeto são definidas pela sua classe, logo, todo o processo de resolução de um problema passa pela identificação dos objetos e, fundamentalmente, pela construção (modelagem) de suas classes.

Não há uma regra geral para identificar os objetos que pertencem a determinado problema, visto que cada problema apresenta uma solução diferente e depende da abstração realizada. Lembre-se que qualquer coisa que tenha algum significado dentro do contexto do problema pode ser considerada um objeto. Nesse sentido, deve-se procurar, dentro do contexto do problema, por coisas, pessoas, instituições, documentos, procedimentos, registros, etc. Cada um desses elementos pode se constituir em um objeto da solução do problema.

Como nosso problema consiste na determinação da área de um círculo, podemos facilmente concluir que nosso programa terá um objeto círculo. Como todo objeto deve ter uma identificação, chamaremos esse objeto de *umCirculo*. Podemos considerar que este objeto deverá ter como atributo o seu raio e um método capaz de calcular sua área.

Como um objeto é sempre criado a partir de uma classe, concluímos que devemos criar uma classe que defina os atributos e métodos necessários ao cálculo da área do círculo. Assim, será a classe do objeto *umCirculo* que definirá quais serão seus atributos e métodos. Chamaremos esta classe de Circulo. Esta classe está representada abaixo.

Circulo
double raio
Circulo() calculaArea()

Pela representação ao lado, podemos perceber que, além do método `calculaArea()`, a classe `Circulo` possui também um método chamado `Circulo()`. Este é um método construtor. Um método construtor possui sempre o mesmo nome da classe e tem o objetivo de construir a instância da classe, isto é, quando executado faz com que determinado objeto passe a existir na memória do computador. Trataremos de métodos construtores em maior profundidade mais adiante.

A seguir, temos a classe `Circulo` já implementada (escrita) em Java. Lembre-se que você deve salvar o arquivo com o nome de `Circulo.java`.

```
public class Circulo {
    double raio;

    public Circulo() {
        raio = 0.0;
    }

    public double calcularArea() {
        double area;
        area = 3.1416 * raio * raio;
        return area;
    }
}
```

A classe acima deverá ser compilada, mas não poderá ser executada (lembre-se de que somente a classe contendo um método `main` pode ser executada). Portanto, devemos criar também outra classe de teste, contendo o método `main`. Chamaremos esta classe de `CirculoMain`. É nela que declararemos e instanciaremos o objeto `umCirculo`.

```
import java.util.Scanner;
```

```

public class CirculoMain {
    public static void main(String[] args) {
        double valorRaio, valorArea;
        Circulo umCirculo;
        umCirculo = new Circulo();
        System.out.println("Digite o valor do raio: ");
        Scanner sc = new Scanner(System.in);
        valorRaio = sc.nextInt();
        umCirculo.raio = valorRaio;
        valorArea = umCirculo.calcularArea();
        System.out.println("A área é " + valorArea);
    }
}

```

Salve o programa acima como CirculoMain.java, compile-o e execute-o.

Neste programa, obtemos um objeto da classe Circulo através dos comandos:

```

Circulo umCirculo;           (1)
umCirculo = new Circulo();   (2)

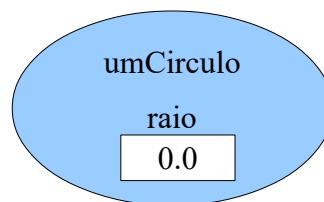
```

A linha (1) do trecho acima declara uma variável chamada umCirculo, do tipo Circulo. Observe que a sintaxe é a mesma que estamos acostumados a utilizar na declaração de variáveis (<tipo> <identificador>; ), só que desta vez estamos usando um tipo de dados não primitivo (classe). Assim, a variável umCirculo será uma variável de referência para um objeto da classe Circulo, diferentemente de valorRaio e valorArea, que são variáveis de um tipo primitivo (double).

O comando `new Circulo()` cria um novo objeto (ou instância) da classe Circulo. Assim, a linha (2) cria o novo objeto e o atribui à variável de referência `umCirculo`.

É importante compreender que não basta declarar uma variável de referência; é necessário criar o objeto com o comando `new` e atribuí-lo a esta variável. Caso contrário, a variável de referência não estaria associada a nenhum objeto, sendo uma referência nula.

A execução do comando `umCirculo = new Circulo();` fará com que o computador execute o método construtor da classe Circulo, identificado por `Circulo()`. Tal execução criará na memória o objeto `umCirculo`, inicializando seu atributo `raio` com o valor 0.0. Este objeto poderia ser representado da seguinte forma:



*Figura 14: Representação do objeto umCirculo.*

Também poderíamos juntar os comandos (1) e (2) em apenas uma linha, da seguinte forma:

```
Circulo umCirculo = new Circulo();
```

Considerando que o objeto `umCirculo` já foi construído, podemos alterar seus atributos e solicitar que execute seus métodos. Por exemplo, o comando abaixo faz com que o atributo `raio` do objeto `umCirculo` receba o valor que foi lido pelo programa:

```
umCirculo.raio = valorRaio;
```

O comando a seguir solicita que o objeto `umCirculo` execute seu método `calculaArea()`. O valor retornado por este método será armazenado na variável `valorArea`.

```
valorArea = umCirculo.calculaArea();
```

Observe a correspondência entre os tipos: a variável `valorArea` e o tipo de retorno do método `calculaArea` são o mesmo tipo (`double`).

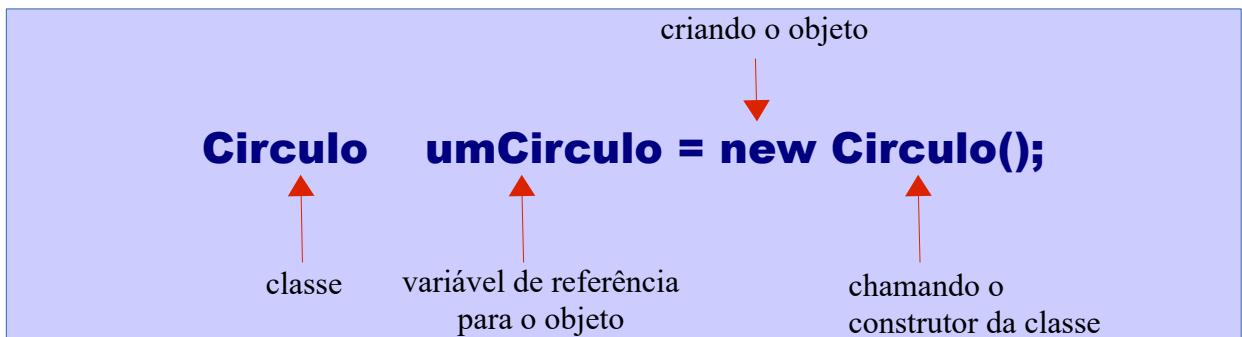


Figura 15: Componentes da declaração e instanciação de um objeto.

## 4.6 MÉTODOS CONSTRUTORES

Um objeto só passa a existir se for construído, e só pode ser construído se sua classe tiver um método construtor. Trata-se de um método com o mesmo nome da classe e sem tipo de retorno.

Podem existir vários outros construtores em uma classe, mas o método construtor sem parâmetros, se não for explicitamente criado pelo programador, será implicitamente criado pelo compilador Java sempre que nenhum outro construtor tiver sido criado.

```
public Funcionario( ) {
}

public Funcionario(String vNome) {
    nome = vNome;
}

public Funcionario(String vNome, float vSalario, int vDep) {
    nome = vNome;
    salario = vSalario;
    numeroDeDependentes = vDep;
}
```

No trecho de código acima temos 3 construtores: o primeiro sem parâmetros, o segundo com um parâmetro e o terceiro com 3 parâmetros. Observe que os 3 métodos possuem o mesmo nome, mas assinaturas diferentes. Isso se chama **sobrecarga de métodos**. No momento de chamar o método, será executado aquele cujo número e a ordem dos argumentos fornecidos coincida com a assinatura.

Usando o exemplo acima, poderíamos criar um objeto `Funcionário` utilizando qualquer um dos formatos abaixo. Para cada caso, o construtor correspondente seria chamado.

```

Funcionario f1 = new Funcionario();
Funcionario f2 = new Funcionario("João");
Funcionário f3 = new Funcionario("Maria", 1500, 2);

```

## 4.7 MEMBROS ESTÁTICOS

A palavra-chave *static* indica que o escopo do membro (atributo ou método) é a classe, e não a instância.

### 4.7.1 Atributos estáticos

Há situações em que se torna desejável que todos os objetos de uma classe compartilhem o conteúdo de um atributo. Esse efeito pode ser obtido com o uso do modificador *static* na definição do atributo.

Os atributos vistos até agora (não estáticos) são chamados de variáveis de instância, já que cada objeto da classe possuirá uma cópia deste atributo e poderá ter valores distintos para o mesmo (ex: cada funcionário tem um valor diferente para o atributo nome).

Ao declararmos um atributo como estático, o atributo se torna uma variável de classe, isto é, todos os objetos daquela classe terão acesso ao mesmo valor atributo. Se um objeto da classe mudar o valor do atributo *static*, todos os objetos da classe terão acesso ao novo valor. Atributos estáticos são o mais próximo de variáveis globais que Java oferece.

Suponha que declaremos, na classe Funcionário, o atributo abaixo:

```
static int totalDeFuncionarios = 0;
```

Ao ser declarado como um atributo estático, poderemos compartilhar o valor do atributo *totalDeFuncionarios* entre todos os objetos desta classe. Na classe de teste, se utilizarmos:

```

Funcionario f1 = new Funcionario();
f1.totalDeFuncionarios++;

Funcionario f2 = new Funcionario();
f2.totalDeFuncionarios++;

System.out.println("Total em F1: "+f1.totalDeFuncionarios);
System.out.println("Total em F2: "+f2.totalDeFuncionarios);

```

Será impresso na tela:

```
Total em F1: 2
Total em F2: 2
```

Repare que *totalDeFuncionarios* foi incrementado em cada objeto, porém seu valor é compartilhado pelos dois objetos.

O melhor jeito de referenciar uma variável estática é utilizando o **nome da classe** e não o nome do objeto. No exemplo acima, em vez de utilizar as linhas

```
f1.totalDeFuncionarios++;
f2.totalDeFuncionarios++;
```

Poderíamos utilizar:

```
Funcionario.nroDeFuncionarios++;
```

Para imprimir o total de funcionários, também poderíamos fazer:

```
System.out.println("Total na classe: " + Funcionario.nroDeFuncionarios);
```

**OBS:** O uso **abusivo** de elementos estáticos é desencorajado, pois elementos estáticos são armazenados uma vez na memória e todos os objetos com acesso permitido acessarão o mesmo local. Isso pode ser fonte de inconsistência, já que diversos objetos estarão lendo e escrevendo no mesmo local concorrentemente. Adicionalmente, na orientação a objetos deve-se evitar que informações desnecessárias sejam visíveis por objetos externos.

#### 4.7.2 Métodos estáticos

Os métodos estáticos são declarados com o modificador **static** e não exigem instanciação de um objeto para que possam ser chamados.

Todo método de nossas classes programadas até agora (exceção feita ao main) operam sobre alguma instância particular da classe (um objeto daquela classe), criada através do comando new. Por este motivo, podem ser chamados de métodos de instância, da mesma forma que um atributo não static pode ser chamado variável de instância.

Um método static (ou método de classe) é aquele declarado com a palavra-chave static e não opera sobre um objeto em particular. Usa-se o método static para casos em que não há necessidade de instanciar um objeto para chamar um método.

Por exemplo, na classe funcionário, poderíamos ter o seguinte método estático:

```
public static int getTotalDeFuncionarios() {  
    return totalDeFuncionarios;  
}
```

Assim, em qualquer outra classe poderíamos chamar este método através do nome da classe, mesmo sem instanciar um objeto com new.

```
System.out.println(Funcionario.getTotalDeFuncionarios());
```

É importante destacar que um método static não pode acessar membros não-static.

Apesar da aparente facilidade, deve-se reduzir ao mínimo a utilização de static. As classes utilitárias do Java, como a classe Math, são situações que mais justificam o emprego dos membros estáticos.

## 4.8 PASSAGEM DE PARÂMETROS

A passagem de parâmetros é o meio utilizado nas linguagens de programação para transferir valores entre os métodos. No momento em que o método é chamado, devem ser passados argumentos que serão colocados, pela ordem, nos respectivos parâmetros.

Existem duas abordagens para passar os argumentos (parâmetros) para os métodos de classe ou de instância: chamada por valor e chamada por referência.

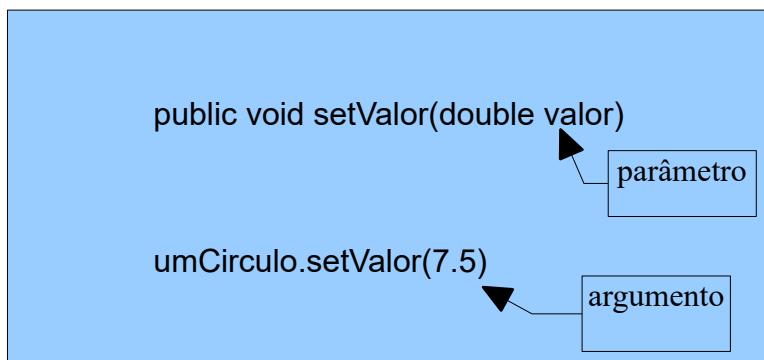


Figura 16: Parâmetro e argumento.

As **chamadas por valor** fazem uma cópia de cada argumento e passam essa cópia para o método. Assim, o valor original não é modificado. Em Java, somente os **tipos primitivos** (int, double, float...) são passados por valor. Isso significa que, se durante seu processamento, o método realizar alguma mudança nos conteúdos dos parâmetros, as variáveis originais não serão afetadas (no método que realizou a chamada).

Em contraste, as **chamadas por referência** passam somente uma referência ao endereço do argumento original, tornando possível ao método chamar os métodos e acessar os atributos (elementos) deste argumento e modificá-los. Em Java, quando passamos objetos como parâmetros, a passagem é por referência.

## 4.9 ESCOPO DE VARIÁVEIS

O escopo de uma variável é o espaço do programa em que a referida variável é reconhecida. O uso de uma variável fora de seu escopo causa um erro em tempo de compilação.

Java restringe o escopo da variável ao bloco onde foi declarada. Assim, quanto ao escopo, as variáveis se dividem em:

- *Atributo privado*: seu escopo é a classe toda. A alocação ocorre quando o objeto é instanciado e eliminação quando o objeto é destruído.
- *Parâmetro*: seu escopo é o método ou o construtor onde foi declarado. A alocação se dá quando o método onde foi declarado é chamado e a eliminação quando o método termina sua execução.
- *Variável local*: seu escopo é o bloco de código onde foi declarada. A alocação ocorre quando o bloco de código onde ela foi declarada inicia a execução e sua eliminação ocorre ao término desta execução.

## 4.10 RESUMO DO CAPÍTULO

- Classe e objeto são conceitos diferentes, apesar de estarem fortemente ligados. Uma classe descreve a estrutura (atributos) e comportamento (métodos) de um grupo de objetos. A classe pode ser visualizada como um molde de objetos.
- Todo objeto, além de apresentar um nome, apresenta também a estrutura e o comportamento definidos em sua classe.

- Os termos objeto e instância são sinônimos.
- Atributos e métodos são ditos membros da classe.
- Um atributo pode ser um valor ou outro objeto.
- Os objetos são construídos quando da execução do software pelo computador.
- Um objeto é criado pelo operador new. Ao executar o comando new Classe(), será criado um novo objeto da referida classe e seu método construtor será executado.
- Um método construtor deve ter o mesmo nome da classe e não especificar tipo de retorno.
- Com exceção dos métodos construtores, todo método em Java deve ter um tipo de retorno. Caso não apresente nenhum valor de retorno, então seu tipo será void.
- O modificador static faz com que o membro seja pertencente à classe, e não a instâncias individuais.
- Um atributo estático assemelha-se a uma variável global, pois tem seu valor compartilhado por todos os objetos da classe.
- Um método estático pode ser invocado diretamente através do nome da classe, sem a necessidade de instanciar um objeto.
- Parâmetros de tipos primitivos são passados por valor. Já parâmetros de tipos não primitivos (objetos) são passados por referência.

#### **4.11 BIBLIOGRAFIA DO CAPÍTULO**

BORATTI, Isaias Camilo. **Programação orientada a objetos em Java**. Florianópolis: Visual Books, 2007.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

BARNES, David J; Kölking, Michael. **Programação Orientada a Objetos com Java**: uma introdução prática usando o BlueJ. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

CARDODO, Aníbal Pereira. Notas de aula. Disponível em <<http://www.inf.unisinos.br/~anibal/>>. 2011.

## 5 Encapsulamento

Uma das grandes vantagens do paradigma de programação orientada a objetos é facilidade de criação de códigos reutilizáveis, ou seja, que são escritos uma vez e podem ser usados em diversos programas diferentes. A reutilização de código está intimamente ligada com a forma pela qual esse código é organizado e disponibilizado.

O encapsulamento de código é um conjunto de técnicas e boas práticas de programação que garantam que o código escrito seja visto como uma "cápsula", algo que contém uma parte externa (a "casca") e uma parte interna (o "núcleo"). A parte externa pode ser facilmente visualizada e entendida, porém não é possível (ou é desnecessário) o conhecimento do seu interior.

Utilizando um exemplo prático diretamente relacionado com o mundo real, podemos imaginar um celular. Ele possui uma parte externa bem definida (geralmente um invólucro de plástico com um teclado e uma tela) e uma parte interna desconhecida (circuitos, fios, chips, capacitores, etc). Ao utilizarmos um celular, não o vemos como um emaranhado de circuitos com uma antena; o vemos como uma espécie de caixa-preta que pode fazer e receber ligações. Podemos até conhecer e entender como as ligações são transmitidas (ondas de rádio através de antenas, etc), porém isso não é de nosso interesse. Não sabemos quais chips recebem corrente elétrica quando uma ligação é feita, ou quantos amperes cada capacitor recebe: apenas queremos fazer ligações e trocar mensagens.

Na visão de orientação a objetos, o celular é uma cápsula. Ele possui um conjunto de funcionalidades que podem ser utilizadas através de uma interface pública (a tela e o teclado), porém o conteúdo do aparelho em si (e seu funcionamento) não nos interessa. Dessa forma, se algum dia o fabricante do celular decidir modernizar os componentes eletrônicos internos do celular para que as ligações sejam melhores, basta que o celular seja levado em uma loja e que os componentes sejam trocados. Depois disso, o celular fará e receberá chamadas melhores, porém a sua tela e o seu teclado continuam iguais. Qualquer um que utilize aquele celular continuará utilizando-o da mesma forma, porém agora com capacidade melhorada.

O encapsulamento na programação orientada a objetos tenta criar esse cenário. As classes são desenhadas de forma a funcionarem como cápsulas, caixas-pretas: existe uma interface pública (os métodos públicos) que disponibiliza um conjunto de funcionalidades, porém por trás disso existem códigos que implementam essas funcionalidades. Os códigos que implementam as funcionalidades ficam indisponíveis, e a única preocupação do programador ao usar uma classe é saber **o que** ela faz, **não como** ela faz.

### 5.1 MODIFICADORES DE VISIBILIDADE

O encapsulamento é obtido através dos modificadores de visibilidade. Esses modificadores são palavras-chave que são colocadas junto a métodos e atributos da classe e indicam quais desses membros são visíveis para classes externas.

Em Java, existem três tipos de modificadores de visibilidade: **public**, **protected** e **private**. O código abaixo ilustra a utilização desses três modificadores.

```
class Pessoa {
    public String nome;
    protected String sobrenome;
    private int idade;

    public void metodoPublico() {
    }

    protected void metodoProtected() {
    }
}
```

```
    private void metodoPrivate() {
    }
}
```

No código mostrado acima, cada membro (atributo ou método) da classe possui um modificador de acesso diferente, assim como nos métodos. A tabela abaixo mostra se um membro está visível de acordo com seu modificador de visibilidade.

Modificador	Classe	Pacote	Subclasse	Mundo
public	Sim	Sim	Sim	Sim
protected	Sim	Sim	Sim	Não
private	Sim	Não	Não	Não
Sem modificador	Sim	Sim	Não	Não

Analizando-se a primeira linha da tabela, podemos perceber que se um membro for marcado como *public*, ele estará disponível para o resto da classe (coluna 1), para outras classes do mesmo pacote (coluna 2), para uma subclasse (coluna 3) e para qualquer outra classe externa (coluna 4). *Public* é o nível de acesso mais permissivo.

A segunda linha da tabela mostra que o modificador *protected* torna o membro acessível dentro da própria classe, a partir de outras classes do mesmo pacote e também a partir de subclasses (mesmo que estejam em outros pacotes). Ou seja, elementos marcados com *protected* ficam indisponíveis para classes externas, isto é, que não estão no mesmo pacote e não são subclasses.

O modificador *private*, por sua vez, faz com que atributos e métodos fiquem disponíveis apenas para a própria classe onde foram definidos. É o nível de acesso mais restritivo. Utilizando-se a classe *Pessoa* do exemplo anterior, vamos imaginar o seguinte programa:

```
class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        p.idade = 12;
    }
}
```

Esse programa não compilará e uma mensagem de erro será informada indicando que a classe *Teste* está tentando acessar o atributo *idade* do objeto *p* (da classe *Pessoa*), porém este está marcado como *private*. Isso não acontece com os atributos *nome* e *sobrenome* da classe *Pessoa*, porque eles estão marcados como *public* e *protected*, respectivamente. O seguinte código compilará e funcionará normalmente:

```
class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        p.nome = "Fulano";
        p.sobrenome = "Silva";
    }
}
```

Quando deixamos um membro sem modificador de acesso, o mesmo poderá ser acessado somente dentro do pacote.

A utilização dos modificadores de acesso garantem que o programador disponibilize ou não certos métodos e atributos para classes externas. Isso permite a criação das cápsulas/caixas-pretas, que são classes que desempenham uma determinada tarefa, porém parte do seu

conteúdo (métodos e atributos marcados como *private*, por exemplo) não estão visíveis para o mundo externo.

## 5.2 MÉTODOS DE ACESSO E MÉTODOS MODIFICADORES

Segundo as boas práticas de programação e o conceito de encapsulamento, um atributo deve ser alterado somente por métodos da classe que o possui; além disso um atributo deve ser lido através de um método, e não deve ser acessado diretamente.

A interpretação dessas ideias de encapsulamento deve ser analisada com bastante critério, porque às vezes é mais claro e lógico acessar o valor de um atributo diretamente do que através de um método. Por outro lado, restringir o acesso a um atributo através de um método garante de forma mais segura e enfática a reusabilidade do código.

Para isolar atributos e permitir que eles sejam lidos e escritos através de métodos existe o conceito de *métodos de acesso*. Por via de regra, a nomenclatura desses métodos segue uma regra simples: eles tem o nome do atributo em questão e um prefixo: métodos que **obtém** um valor de um atributo são prefixados com **get** e métodos que alteram seu valor são prefixados com **set**. Por terem esses prefixos, estes métodos são comumente chamados de *getters* e *setters* que, numa tradução literal, significam "obtentores" e "modificadores", respectivamente.

O exemplo abaixo mostra a classe *Pessoa* com dois atributos (*nomeCompleto* e *idade*), sendo que cada um deles possui métodos de acesso e modificadores (um *get* e um *set*):

```
class Pessoa {
    private String nomeCompleto;
    private int idade;

    public String getNomeCompleto() {
        return this.nomeCompleto;
    }

    public void setNomeCompleto(String n) {
        this.nomeCompleto = n;
    }

    public int getIdade() {
        return this.idade;
    }

    public void setIdade(int i) {
        this.idade = i;
    }
}
```

Ambos os métodos *get* do exemplo acima, o *getNomeCompleto()* e o *getIdade()*, não recebem parâmetros, porém possuem um valor de retorno. No caso de *getNomeCompleto()* o tipo de retorno é *String*, porque ele retorna o atributo *nomeCompleto*, que é do tipo *String*. Já no método *getIdade()* o tipo de retorno é *int*, porque o método retorna o atributo *idade*, que é do tipo *int*.

Os métodos *set*, por sua vez, recebem parâmetros para que possam modificar o valor do atributo aos quais estão associados. No exemplo acima, por exemplo, o método *setNomeCompleto()* recebe como parâmetro uma *String*, que será atribuída à *nomeCompleto*. O mesmo acontece com o método *setIdade()*, ele recebe um parâmetro do tipo *int* que, por sua vez, será atribuído à *idade* da classe.

Agora vamos ver os métodos de acesso (*getters/setters*) em ação. O exemplo abaixo mostra um programa que instancia um objeto da classe *Pessoa* e, em seguida, atribui valores aos seus

atributos através de chamadas a métodos *set*. Em seguida, os valores são mostrados através de chamadas a métodos *get*.

```
class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();

        p.setNomeCompleto("Fulano da Silva");
        p.setIdade(29);

        System.out.println("Nome = " + p.getNomeCompleto());
        System.out.println("Idade = " + p.getIdade());
    }
}
```

É muito importante notar que, para que a reutilização de código e o encapsulamento funcionem, os métodos e atributos públicos de uma determinada classe não podem ser alterados depois que essa classe estiver em uso. O conjunto de métodos e atributos públicos de uma classe são a sua interface pública para com o mundo, isto é, a forma como o mundo externo enxerga a classe. Recordando do exemplo do celular, o teclado e a tela são a interface pública do aparelho. Todos os componentes internos do celular podem ser alterados, porém ninguém notará (ou se importará com) isso se a interface pública permanecer a mesma.

No caso da classe *Pessoa* com *getters* e *setters*, podemos utilizá-la em quantos programas quisermos e modificá-la à vontade, desde que os métodos e atributos públicos não sejam alterados. Vamos imaginar que, por alguma razão, precisamos trocar o nome de **todos** os atributos da classe *Pessoa* e que precisamos adicionar um **método novo**. A regra básica do encapsulamento e da reusabilidade de código é não alterar a interface pública, então vamos identificar quais são os atributos e métodos dessa classe que não poderemos mexer:

- Atributos públicos: *não há*
- Métodos públicos:
  - `public String getNomeCompleto()`
  - `public void setNomeCompleto(String n)`
  - `public int getIdade()`
  - `public void setIdade(int i)`

Os métodos listados acima são aqueles cuja assinatura não deve ser alterada. O corpo do método pode ser alterado, contanto que o nome do método, o tipo de retorno e os parâmetros recebidos continuem os mesmos. Agora que já sabemos o que não pode ser alterado, vamos para a alteração (trocar o nome dos atributos e adicionar um método novo). Estão marcados em vermelho os códigos que sofreram alterações.

```
class Pessoa {
    private String meuNomeCompleto;
    private int minhaIdade;

    public String getNomeCompleto() {
        return this.meuNomeCompleto;
    }

    public void setNomeCompleto(String n) {
        this.meuNomeCompleto = n;
    }

    public int getIdade() {
        return this.minhaIdade;
    }
}
```

```

        public void setIdade(int i) {
            this.minhaIdade = i;
        }

        public void limpaDados() {
            this.meuNomeCompleto = "Sem nome?";
            this.minhaIdade = 0;
        }
    }
}

```

Como explicado antes na regra para nomenclatura de métodos *get/set*, os métodos *get/set* devem ter o nome dos respectivos atributos mais um prefixo. Nesse caso, teríamos que alterar o métodos *get*, por exemplo, para *getMinhaIdade()* e *getMinhaIdade()*. Não podemos fazer isso, porém, porque esses são métodos públicos da classe e, se alterássemos, estaríamos fazendo com que a interface pública de acesso da classe fosse diferente. Se fizermos essa troca de nome de métodos públicos, todos os programas que fazem uso da classe *Pessoa* deveriam ser adaptados e recompilados para que os novos métodos *get/set* fossem utilizados.

Como não alteramos o nome, tipo de retorno e/ou parâmetros dos métodos públicos, nossa classe *Pessoa* continua com a mesma interface pública, porém a sua estrutura interna está diferente. O programa anterior, que usa a classe *Pessoa*, continuaria funcionando normalmente e **não precisaria** ser adaptado ou recompilado para utilizar a nossa nova classe *Pessoa*. O código abaixo continuaria funcionando exatamente da mesma maneira que antes:

```

class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();

        p.setNomeCompleto("Fulano da Silva");
        p.setIdade(29);

        System.out.println("Nome = " + p.getNomeCompleto());
        System.out.println("Idade = " + p.getIdade());
    }
}

```

O desenvolvedor da classe *Teste* não precisa saber como chamamos as propriedades nome e idade da classe *Pessoa*, ele apenas quer uma classe que represente uma pessoa. No futuro, poderemos alterar novamente a classe *Pessoa* por quantas vezes desejarmos, inclusive adicionando novos métodos, desde que nenhum método e/ou propriedade pública sejam alterados.

### 5.3 RESUMO DO CAPÍTULO

- Os modificadores de acesso tornam propriedades e/ou métodos visíveis ou invisíveis para outras classes. Os modificadores são: ***public***, ***protected*** e ***private***.
- O encapsulamento de classes facilita a reutilização de código.
- Pelas regras de boa programação, os atributos devem ter seus valores alterados e acessados apenas por métodos da classe.
- Métodos de acesso são chamados *getters*.
- Métodos modificadores são chamados *setters*.

#### **5.4 BIBLIOGRAFIA DO CAPÍTULO**

BORATTI, Isaias Camilo. **Programação orientada a objetos em Java**. Florianópolis: Visual Books, 2007.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

BARNES, David J; Kölking, Michael. **Programação Orientada a Objetos com Java: uma introdução prática usando o BlueJ**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

## 6 Operações de Abstração

Abstração pode ser definida como a capacidade de descrever características essenciais do objeto que o distinguem de outros tipos de objetos. Em outras palavras, focar no que é relevante e ignorar detalhes irrelevantes para o contexto em que estamos trabalhando. A título de exemplo, vamos considerar um dado que todas as pessoas possuem: tipo sanguíneo e fator RH. Se estivermos modelando um sistema para uma clínica médica ou laboratório de análises clínicas, certamente este dado será relevante e deverá constar de alguma forma em nosso modelo da entidade “Pessoa”. Por outro lado, se o objetivo for a construção de um sistema para uma biblioteca ou loja on-line, é bem provável que não haja interesse em armazenar tal dado. Para cada caso, construímos então um modelo que contemple todos os objetos Pessoa (criando então a classe Pessoa), onde estão presentes apenas as características que atendam aos requisitos da aplicação ora em desenvolvimento, sendo que as demais são abstraídas. A este processo de identificação das classes e seus objetos damos o nome de **classificação/instanciação**.

Em um sistema orientado a objetos o software é construído com base em elementos do mundo real (entidades físicas ou abstratas). Os dados e procedimentos relativos a uma entidade, identificados por meio de abstração, são encapsulados em uma mesma unidade. Como vimos no capítulo anterior, o encapsulamento permite esconder detalhes de um objeto que não contribuem para suas características essenciais, possibilitando separar aspectos de implementação e aspectos de uso. Assim, cada objeto possui uma interface visível e uma implementação interna invisível. Isso contribui com o conceito de modularidade, segundo o qual um sistema deve ser composto por módulos coesos e fracamente acoplados (ou seja, o mais independentes possível uns dos outros). Teremos então, em um sistema de loja virtual, as classe Cliente, Produto, Categoria, Venda, Carrinho, etc., cada classe contendo dados (atributos) e procedimentos (métodos) relativos às entidades correspondentes a que representam (ou seja, não “misturamos” dados de clientes e dados de produtos em uma mesma classe). Quando há necessidade de relacionar um objeto a outro, devemos estabelecer uma **associação**.

Um sistema complexo e com muitos detalhes pode ainda ser decomposto em uma hierarquia de abstrações, onde podemos inserir detalhes de maneira controlada, por meio de **agregação/decomposição** ou **generalização/especialização**.

Na sequência, abordaremos estas operações de abstração com maiores detalhes.

### 6.1 CLASSIFICAÇÃO/INSTANCIACÃO

No mundo real, classificar significa agrupar objetos com características e comportamentos semelhantes em uma mesma categoria.

Na orientação a objetos, as categorias são classes. Cada objeto tem sua própria existência e características, e todos os objetos que apresentam as mesmas características são definidos como pertencentes a uma mesma classe.

Por outro lado, instanciar objetos significa gerar novos exemplares a partir de uma definição abstrata de um objeto genérico (classe). O objeto instanciado a partir de determinada classe contém todas as características e comportamentos lá definidos.

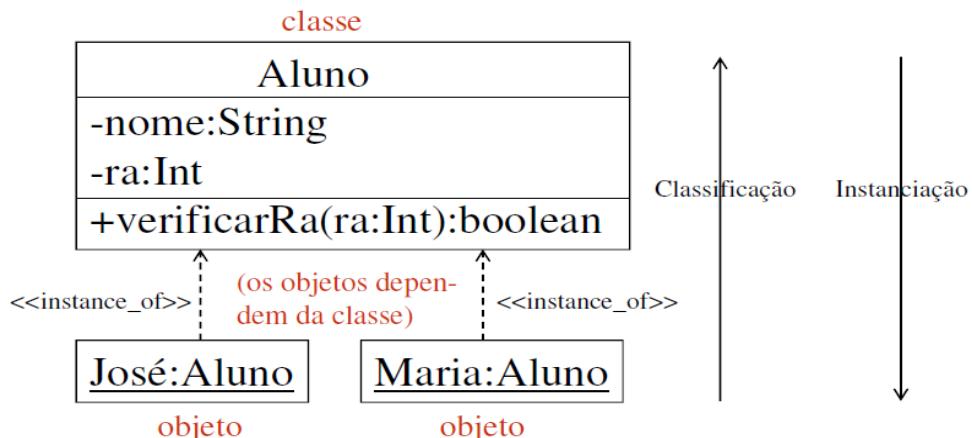


Figura 17: Classificação e Instanciação

## 6.2 ASSOCIAÇÃO

Em um programa orientado a objetos, assim como no mundo real, os objetos raramente existirão de forma isolada. Isso nos leva à necessidade de compreender e determinar a maneira como as classes se relacionam entre si, compartilhando informações.

A operação de associação é uma das formas mais simples de relacionamento entre classes e ocorre quando existe uma classe que possui um atributo de outra classe (ou ainda um array de objetos de outra classe). As entidades relacionadas existem independentemente uma da outra.

Este tipo de relacionamento também é conhecido por relacionamento “tem um(a)”. Por exemplo, considerando que temos as classes Usuário e Reserva, a associação abaixo diz que um Usuário pode ter várias Reservas (\* indica zero ou mais), ao passo que uma Reserva estará sempre vinculada a um único Usuário.

A associação é expressa no diagrama de classes UML como uma linha conectando duas classes. A multiplicidade do relacionamento pode ou não estar presente.

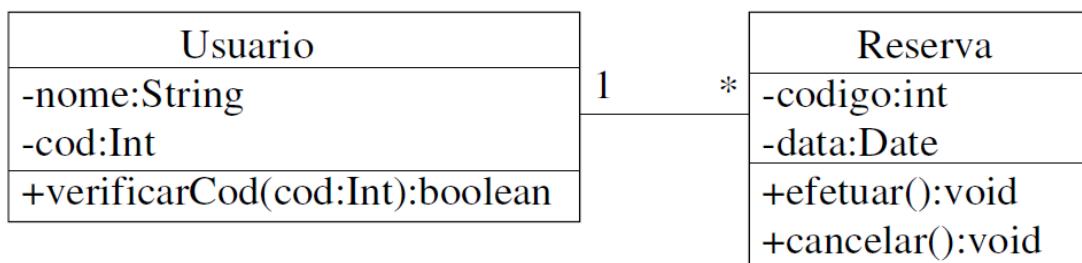
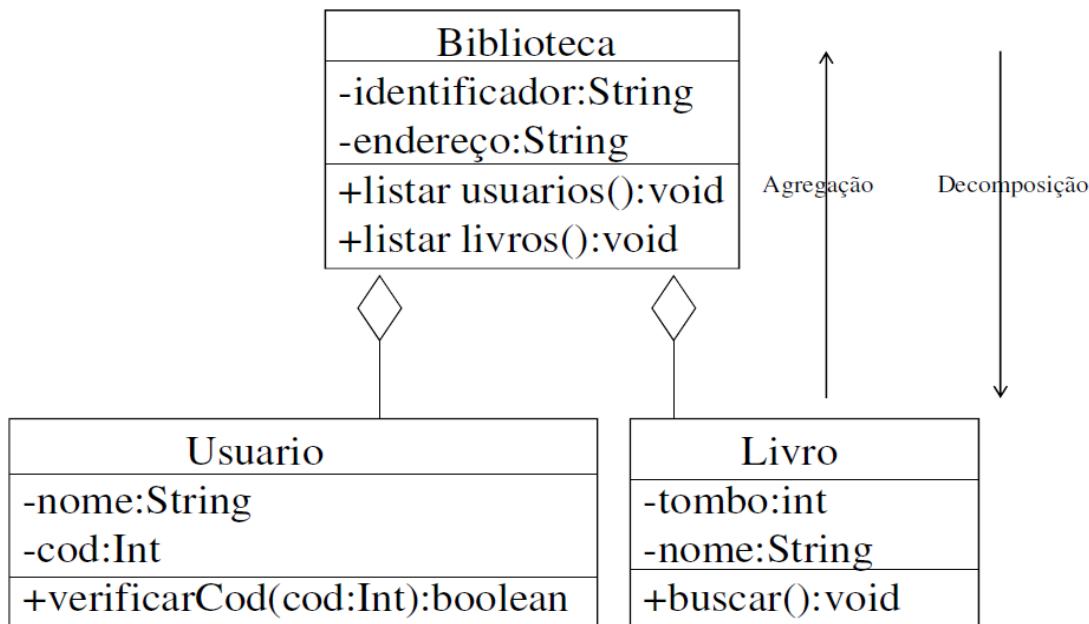


Figura 18: Associação.

## 6.3 AGREGAÇÃO/DECOMPOSIÇÃO

A agregação é uma forma especial de associação caracterizada por um relacionamento do tipo “todo-parte”, onde uma das classes é o todo e a outra é a parte. Por exemplo, uma turma é um todo composto por alunos (partes).

Em uma agregação, a classe “todo” contém elementos que a vinculam com a classe “parte”. No exemplo abaixo, vemos que a Biblioteca é composta por um conjunto de Usuários e um conjunto de Livros. Deste modo, ao invés de incorporar todos os dados dos usuários e dos livros na classe Biblioteca, a mesma foi decomposta em suas partes componentes. Biblioteca (todo) é, então, uma agregação de livros e usuários (partes).

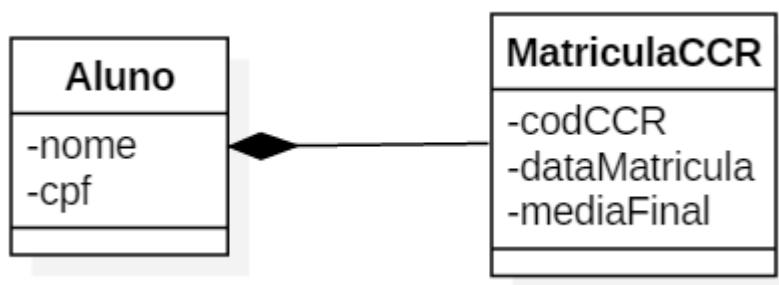


*Figura 19: Agregação e decomposição.*

Quando unimos um conjunto de objetos com o objetivo de formarmos um novo, estamos realizando uma agregação. Ao analisar um objeto, se isolamos cada um de seus componentes, estamos fazendo uma decomposição.

A agregação é expressa no diagrama UML como um losango aberto.

Existe um tipo de associação ainda mais forte que Agregação, chamado Composição (não confundir com a decomposição explicada acima). A composição também é um relacionamento caracterizado como parte / todo, mas, neste caso, o todo é responsável pelo ciclo de vida da parte (isto é, a parte não existe sem o todo). Neste caso, o diagrama UML mostrará um losango preenchido. Por exemplo, um Aluno possui muitas matrículas em CCR. Se excluirmos um aluno, suas matrículas também devem ser excluídas, pois não existem ou não fazem sentido sem o aluno a que se referem.



*Figura 20: Composição: um tipo mais forte de agregação.*

Cabe aqui destacar que a diferença entre agregação e composição é conceitual: dentro do Java e linguagens de programação em geral, a implementação desses conceitos é a mesma.

## 6.4 GENERALIZAÇÃO / ESPECIALIZAÇÃO

A generalização e a especialização implementam o conceito de herança.

Sempre que, a partir de uma classe mais genérica, se definir uma classe mais especializada, estamos fazendo uma Especialização. A classe mais especializada herda (mantém) as características da classe mais geral e, adicionalmente, pode definir características e comportamentos específicos.

A operação de Generalização, é quando, a partir de um grupo de classes, identificamos características que são comuns a todas e definimos uma nova classe, mais geral, com essas características.

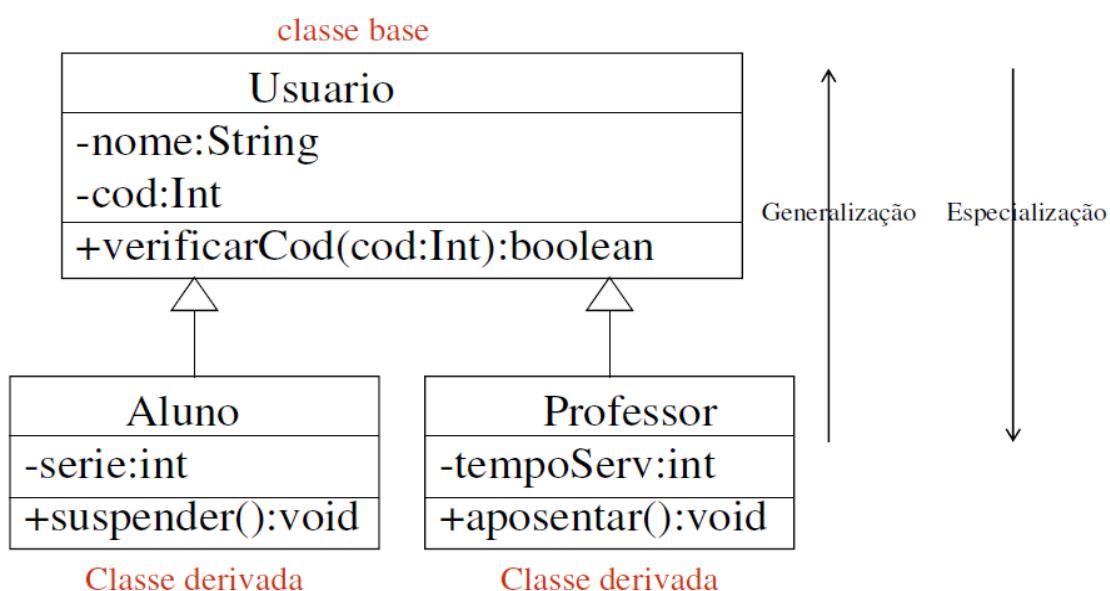


Figura 21: Generalização e especialização.

A generalização/especialização é mostrada no diagrama UML como uma seta aberta.

## 6.5 RESUMO DO CAPÍTULO

- Em um associação, uma classe possui uma ligação com outra e, deste modo, pode usar esta classe. O relacionamento é do tipo “tem um”.
- Em uma especialização, uma classe é a outra, acrescida de características próprias. O relacionamento é do tipo “é um”.
- Uma classe pode ser decomposta em outras. Agregação e composição são relacionamentos do tipo todo-parte entre classes.
- Em uma agregação, as partes podem existir sem o todo.
- Em uma composição, as partes não existem sem o todo.

## 6.6 BIBLIOGRAFIA DO CAPÍTULO

BRITO. Elaine. Conceitos de Orientação a Objetos. Disponível em <http://ebrito.com.br/profs.html>.

## 7 Herança

A possibilidade da reutilização de código é uma das características mais poderosas da programação orientada a objetos. O conceito de herança está intimamente ligado à ideia de abstração e especialização, na qual um conjunto de classes pode *compartilhar* características em comum, em vez de cada uma delas implementar repetidamente essas características. O princípio básico é reutilizar o que for comum e especializar o que for específico.

A herança permite que uma classe herde atributos e métodos de outra classe. Isso permite que novas classes sejam criadas com base em outra classe preexistente e, deste modo, já possuam de antemão todos os atributos e métodos desta. As subclasses podem redefinir métodos herdados (sobreposição ou *override*), além de conter seus próprios atributos e métodos.

Para exemplificar, vamos imaginar duas classes: *Cachorro* e *Gato*. Embora essas classes representem animais completamente diferentes, ambas possuem elementos em comum, como cor dos olhos. Em vez de adicionarmos um atributo *corOlhos* em ambas as classes, como seria feito em um modelo procedural (como C), podemos realizar uma generalização: criar uma terceira classe *Mamífero* que estará um nível acima de hierarquia. A classe *Mamífero* é chamada de superclasse (ou classe pai) e nela colocaremos o atributo *corOlhos* e quaisquer outros atributos/métodos comuns às classes *Cachorro* e *Gato*. *Cachorro* e *Gato* tornam-se então subclasses (ou filhas) de *Mamífero*. A notação UML para herança é uma seta aberta apontando para a superclasse, como mostra a figura abaixo.

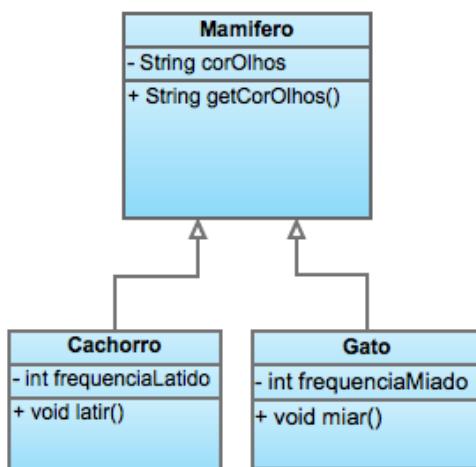


Figura 22: Herança com uma superclasse e duas subclasses.

A superclasse contém todos os atributos e métodos que são comuns às classes que herdam dela. A superclasse *Mamífero* contém todas as características que definem um mamífero, então não é necessário que essas características sejam reimplementadas nas subclasses. Sem a utilização de herança, ambas as classes *Cachorro* e *Gato* deveriam conter todos os atributos e métodos, mesmo os que são comuns, levando a uma replicação de código que tende a deixar o sistema difícil de ser mantido e propenso a falhas.

Ambas as classes *Cachorro* e *Gato* herdam de *Mamífero*. Isso quer dizer que os atributos e métodos da classe *Mamífero* também estão presentes nas classes *Cachorro* e *Gato*.

Analizando-se a classe *Cachorro*, podemos ver que ela tem os seguintes atributos:

- `corOlhos` (herdado da classe *Mamífero*);
- `frequenciaLatido` (definido na própria classe *Cachorro*);

e os seguintes métodos:

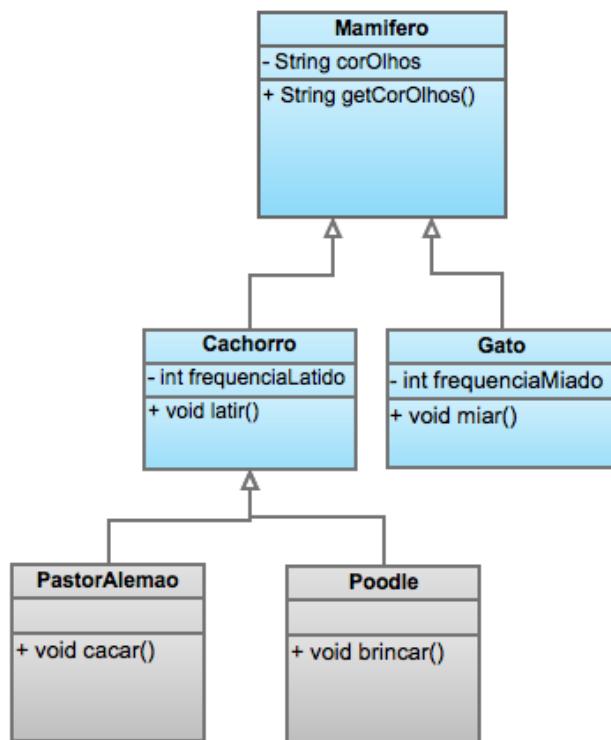
- `getCorOlhos()` (herdado da classe `Mamifero`);
- `latir()` (definido na própria classe `Cachorro`).

Quando um objeto da classe `Cachorro` for instanciado, ele terá todos os atributos e métodos definidos na própria classe `Cachorro` *mais* as propriedades e métodos definidos na classe `Mamífero` (que foram herdadas).

A herança envolve, portanto, duas operações de abstração:

- **Especialização:** operação em que, a partir de uma classe, identifica-se uma ou mais subclasses, cada uma especificando características adicionais em relação à classe mais geral;
- **Generalização:** operação de análise de um conjunto de classes que identifica características comuns a todas, tendo por objetivo a definição de uma classe mais genérica, a qual especificará essas características comuns.

A classe `Cachorro` pode ser ainda mais especializada se pensarmos que existem diversas raças de cachorro, cada uma delas com suas peculiaridades. Nessa ideia, a classe `Cachorro` poderia ser uma superclasse para outras classes, como `PastorAlemao` e `Poodle`, conforme mostra a figura abaixo:



*Figura 23: Especialização da classe Cachorro.*

As classes `PastorAlemao` e `Poodle` herdam as características da classe `Cachorro`. Isso quer dizer que ambas as classes possuem características em comum (como `frequenciaLatido`), porém possuem características próprias que as distinguem entre si. Além disso, elas também possuem as características que definem um mamífero (que foram herdadas a partir da classe `Mamífero`).

## 7.1 IMPLEMENTAÇÃO DE HERANÇA EM JAVA

Em Java, a herança é implementada através da utilização da palavra-chave `extends` na declaração da classe, como mostra o exemplo abaixo:

```
// Arquivo Mamifero.java

public class Mamifero {
    private String corOlhos;

    public String getCorOlhos() {
        return this.corOlhos;
    }
}

// Arquivo Cachorro.java

public class Cachorro extends Mamifero {
    private int frequenciaLatido;

    public void latir() {
        System.out.println("Au au!");
    }
}

// Arquivo PastorAlemao.java

public class PastorAlemao extends Cachorro {

    public void cacar() {
        System.out.println("Estou caçando!");
    }
}
```

Conforme explicado anteriormente, a herança permite que a classe filho herde as propriedades e métodos da classe pai. Isso quer dizer que, embora o método `getCorOlhos()` tenha sido definido na classe `Mamifero`, ele pode ser utilizado na classe `Cachorro` porque ele foi herdado. O código abaixo mostra um exemplo de utilização da classe `Cachorro`:

```
public class Teste {
    public static void main(String[] args) {
        Cachorro r = new Cachorro();

        r.latir();
        System.out.println("Cor dos olhos = " + r.getCorOlhos());
    }
}
```

Mas atenção: as classes filhas só conseguirão utilizar métodos e atributos que tenham sido marcados como `public` ou `protected` pela classe pai. Se algum atributo ou método for marcado como `private` pela classe pai, as subclasses não terão acesso a eles.

Outro aspecto que merece destaque é que uma classe pode ter várias filhas, mas pode ter apenas um pai, o que é chamado herança simples. **Em Java não existe herança múltipla.**

## 7.2 SOBREPOSIÇÃO DE MÉTODOS

É possível que, em determinadas situações, ao modelarmos uma classe como subclasse de outra, seja necessário redefinir a implementação de um ou mais métodos. Esse processo é denominado **sobreposição, sobrescrita** ou **override** e envolve a declaração de métodos com assinatura idêntica tanto na superclasse como na(s) subclasse(s). Um método herdado necessita ser sobreposto quando a forma como o mesmo deve ser implementado na subclasse é diferente da forma como foi implementado originalmente na superclasse.

Utilizando o exemplo anterior, vamos imaginar que o método `latir()` da classe `PastorAlemao` tenha que ser diferente da implementação feita na superclasse `Cachorro`. Nesse caso, teremos que sobrepor o método `latir()` da classe `PastorAlemao` para que ele seja diferente:

```
public class PastorAlemao extends Cachorro {

    public void cacar() {
        System.out.println("Estou caçando!");
    }

    // Método abaixo será sobreposto...
    @Override
    public void latir() {
        System.out.println("Woof Woof!");
    }
}
```

Para sobrepor um método na classe filha, basta redeclará-lo da **mesma** forma que na superclasse (mesmo nome, mesmo retorno, mesmos parâmetros, etc). No exemplo acima, o método `latir()` da classe `PastorAlemao` sobrepõe o método `latir()` da classe `Cachorro`; O método tem a mesma assinatura, porém a implementação é diferente. O exemplo abaixo mostra o uso desse método novo:

```
public class Teste {
    public static void main(String[] args) {
        Cachorro r = new Cachorro();
        PastorAlemao p = new PastorAlemao();

        r.latir(); // imprime "Au au!"
        p.latir(); // imprime "Woof woof!"
    }
}
```

Conforme mostrado no exemplo, o método `latir()` do objeto `Cachorro` imprime "Au au!" na tela. O método `latir()` do objeto `PastorAlemao` também imprimiria "Au au!" na tela, porém ele foi sobreposto para imprimir "Woof woof!".

No momento da execução do método sobreposto, o compilador Java opta pelo método relacionado ao tipo do objeto que o chamou.

A anotação `@Override` é opcional. Além de melhorar a legibilidade do código para o programador, ela informa explicitamente ao compilador que o método a seguir é uma sobreposição. Assim, se houver qualquer erro na declaração da assinatura do método que está sendo sobreposto, o compilador acusará. Sem a anotação `@override`, por exemplo, se o método `latir()` for redefinido na subclasse como `Latir()` (com letra maiúscula), este último será entendido como um novo método e não como sobreposição. Não haverá erro de compilação, mas certamente o funcionamento do programa poderá se dar de forma diferente do planejado, o que pode tomar um tempo maior do programador para descobrir o problema.

### 7.3 O USO DE SUPER

Em determinadas situações, precisamos sobrepor um método herdado, porém ao invés implementarmos ele de forma diferente, precisamos apenas adicionar um novo comportamento, mantendo o funcionamento anterior. No exemplo da classe `PastorAlemão`, vamos supor que o método `latir()` precisa fazer o mesmo que o método `latir()` da classe `Cachorro` (imprimir "Au au!"), porém ele precisa imprimir "Terminei o latido" após latir.

Para cumprirmos essa tarefa, precisamos fazer com que o método `latir()` da classe `PastorAlemão` faça o que fazia antes de ser sobreposto (ou seja, o que foi definido na classe `Cachorro`). Da mesma forma que o operador `this` acessa os métodos e atributos da própria classe, o operador `super` acessa os métodos e atributos da superclasse (como se fosse um `this` da superclasse). O exemplo abaixo mostra a utilização do operador `super`:

```
public class PastorAlemão extends Cachorro {

    public void cacar() {
        System.out.println("Estou caçando!");
    }

    public void latir() {
        super.latir(); // Imprime "Au au!"
        System.out.println("Terminei o latido");
    }
}
```

Através da utilização de `super.latir()`, podemos acessar o método `latir()` que foi definido na superclasse `Cachorro`. Com esse recurso, podemos sobrepor métodos com novas funcionalidades, porém ainda podemos usar as funcionalidades que existiam no método antes dele ser sobreposto.

### 7.4 HERANÇA E CONSTRUTORES

Quando um objeto de uma subclasse é criado, ao executar seu construtor, primeiramente será executado o construtor da superclasse de forma automática.

Na subclasse, podemos chamar explicitamente qualquer construtor da superclasse através do comando `super()` (construtor padrão) ou `super(lista de argumentos)` (outros construtores). Se um construtor da superclasse não for chamado explicitamente, o construtor padrão (sem argumentos) será chamado. Lembre-se: a partir do momento em que um construtor é criado, o construtor vazio deixa de ser criado automaticamente pelo compilador Java. Portanto, deve ser analisada a necessidade de se criar explicitamente o construtor vazio na superclasse, ou então a subclasse deve sempre chamar algum outro construtor com o uso do `super`.

O exemplo abaixo mostra a chamada do construtor da superclasse em diversos níveis da hierarquia. Todos os atributos envolvidos estão marcados como `private`, o que faz com que elas estejam indisponíveis para as classes filhas através da herança.

```
// Arquivo Mamífero.java

class Mamífero {
    private String corOlhos;

    public Mamífero(String novaCor) {
        this.corOlhos = novaCor;
    }
}
```

```

// Arquivo Cachorro.java

class Cachorro extends Mamifero {
    private int frequenciaLatido;

    // Cachorro tem 2 construtores:

    public Cachorro() {
        super("não informada"); // chama construtor da superclasse
        this.frequenciaLatido = 0;
    }

    public Cachorro(String cor, int freqLatido) {
        super(cor); // chama construtor da superclasse passando a cor
        this.frequenciaLatido = freqLatido;
    }
}

// Arquivo PastorAlemao.java

class PastorAlemao extends Cachorro {
    private int areaCaca;

    public PastorAlemao() {
        // Chamando o construtor sem parâmetros da classe Cachorro
        super();

        // Agora inicializamos as propriedades da classe
        // PastorAlemao
        this.areaCaca = 13;
    }
}

```

No exemplo acima, cada uma das classes da hierarquia chama o construtor da sua superclasse. É importante mostrar que o construtor da classe `PastorAlemao` invoca o construtor de sua superclasse (`Cachorro`), que por sua vez chamará o construtor de sua superclasse (`Mamifero`).

Quando `super` estiver sendo usado para invocar o construtor da superclasse, deverá ser o primeiro comando no construtor da subclasse, caso contrário, teremos um erro de compilação.

## 7.5 RESUMO DO CAPÍTULO

- Herança permite o compartilhamento de métodos e atributos entre classes em uma hierarquia.
- Propriedades e métodos marcados como `public` ou `protected` são visíveis pelas classes filho através da herança. Elementos marcados como `private` não são visíveis pelas classes filho.
- A classe mais alta na hierarquia chama-se superclasse (ou classe pai), já as mais baixas (que herdam da superclasse) chamam-se subclasses (ou classes filho).
- A sobreposição de métodos serve para mudarmos a implementação de um método na classe filho se a implementação da classe pai não é adequada para o contexto.
- Através do operador `super` é possível acessar métodos e propriedades não sobrepostos da classe pai. Ex.: `super.latir()`.

- A chamada `super()` é equivalente a chamar o construtor da classe pai.

## 7.6 BIBLIOGRAFIA DO CAPÍTULO

BORATTI, Isaias Camilo. **Programação orientada a objetos em Java**. Florianópolis: Visual Books, 2007.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

BARNES, David J; Kölking, Michael. **Programação Orientada a Objetos com Java: uma introdução prática usando o BlueJ**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

## 8 Classes Abstratas e Interfaces

### 8.1 CLASSES ABSTRATAS

No capítulo 7, vimos que podemos organizar classes em uma estrutura de herança, onde a superclasse contém atributos e métodos comuns que serão herdados por todas as suas subclasses. As subclasses possuem um relacionamento do tipo "é um" ou "é um tipo de" com a superclasse.

Todas as classes vistas eram classes concretas, isto é, classes a partir das quais podemos instanciar objetos através do operador `new`.

Entretanto, algumas classes são desenvolvidas para representar conceitos abstratos. Nestes casos, não é interessante que se possa criar objetos a partir da superclasse. Imagine, por exemplo, uma superclasse `Pet` (animal de estimação). Qual seria a aparência de um animal de estimação? Que tipo de som ele emite? Quantas patas ele tem? Ele tem pelos, escamas ou penas? Percebe-se que não podemos responder adequadamente a estas perguntas sem antes saber de que *tipo* de animal estamos falando, isto é, se trata-se de um cachorro, um gato, um peixe ou mesmo uma ave.

Portanto, a classe `Pet` deveria ser declarada como abstrata, ou seja, uma classe que não pode ser instanciada. Neste caso, somente os objetos das subclasses de `Pet` (como `Cachorro` e `Gato`) poderão ser instanciados.

Para tornar uma classe abstrata, a palavra reservada `abstract` deverá ser incluída em sua definição:

```
public abstract class Pet {  
    String nome;  
    int idade;  
    public void brincar() {  
        ...  
    }  
}
```

Em resumo, uma classe declarada como `abstract` nunca poderá ser instanciada, embora possam ser criadas variáveis de referência de seu tipo. Usando o exemplo acima, **nunca** poderíamos escrever:

```
Pet rex = new Pet(); // não compila
```

Mas poderíamos escrever sem problemas:

```
Pet rex; // compila
```

Mas qual a vantagem de criar uma classe abstrata, se não podemos instanciá-la?

Na verdade, a criação de uma classe abstrata só se justifica quando a estendemos através da herança, pois embora não possa ser instanciada, a classe abstrata pode conter atributos e métodos que serão herdados por suas subclasses. Assim, temos as vantagens da herança (reutilização de código), mesmo sem permitir que objetos da classe pai sejam instanciados.

Por exemplo, se criarmos as classes concretas `Cachorro` e `Gato` como subclasses de `Pet`, estas classes herdarão os atributos `nome` e `idade`, bem como o método `brincar()`. Além disso, poderão ter seus próprios atributos e métodos específicos.

Com o uso de classes abstratas, podemos diminuir nosso código, aumentar a capacidade

de reutilização, manutenção e, principalmente, extensibilidade, utilizando o conceito de polimorfismo, conforme veremos no próximo capítulo.

A API Java (conjunto de bibliotecas que acompanha o JDK) está repleta de classes abstratas, como a classe `JComponent`, do pacote `Swing`. Ela reúne atributos e métodos comuns a todos os tipos de componentes gráficos (botões, janelas, campos, etc). No entanto, não podemos instanciar um `JComponent`; somente podemos instanciar botões, janelas e campos, ou seja, suas subclasses.

Classes abstratas também são muito utilizadas em padrões de projeto (*design patterns*).

## 8.2 MÉTODOS ABSTRATOS

Métodos *abstratos* são métodos que não possuem o corpo, contendo apenas a assinatura. Sua declaração termina com ponto-e-vírgula (`;`) e sem especificar um bloco de código `{}`.

Métodos abstratos são criados para definir serviços, e não implementações, tarefa esta que fica a cargo das classes herdeiras. Ao declarar um método abstrato, você está assinando um contrato que obriga todas as classes herdeiras a implementarem o método, caso contrário haverá erro de compilação. Em outras palavras, fica obrigatória a sobreposição (*override*) do método abstrato nas subclasses.

Vamos pensar no método `emitirSom()` da classe `Pet`. Cachorros e gatos emitem som, mas o fazem da mesma forma? Logicamente não. Portanto, as subclasses `Cachorro` e `Gato` devem ter implementações específicas para o método `emitirSom()`. Para forçar que as subclasses de `Pet` tenham o método `emitirSom()`, declararemos o mesmo como abstrato.

```
public abstract class Pet {
    String nome;
    int idade;
    public void brincar() {
        ...
    }
    public abstract void emitirSom();
}
```

No exemplo acima, temos 2 métodos: `brincar` (método não abstrato) e `emitirSom` (método abstrato). Note que `emitirSom` não possui corpo.

As classes `Cachorro` e `Gato` mostradas abaixo são subclasses de `Pet`. Como `Pet` possui um método abstrato (`emitirSom()`), todas as suas subclasses serão **obrigadas** a sobrepor este método, fornecendo a ele um corpo com uma implementação específica. Os atributos (`nome` e `idade`) e o método `brincar()` serão herdados normalmente pelas subclasses.

```
public class Cachorro extends Pet {
    public void emitirSom() {
        System.out.println("Au au au");
    }
    public void cavar() {
        ...
    }
}
```

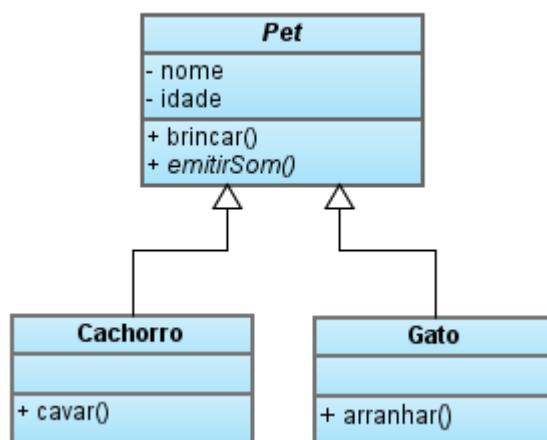
```

public class Gato extends Pet {
    public void emitirSom(){
        System.out.println("Miau");
    }
    public void arranhar() {
        ...
    }
}

```

Veja que o método não abstrato `brincar()` não precisa, necessariamente, ser sobreposto nas subclasses. Isso até pode acontecer, mas como o método já possui corpo na superclasse, se as subclasses não o fizerem, será utilizada a implementação da superclasse. Essa é a diferença entre um método abstrato e um não abstrato: **o método abstrato, por não ter corpo, precisa ser redefinido e ganhar um corpo nas subclasses.** Já para um método não abstrato, a redefinição é possível, mas não obrigatória.

Observe abaixo as classes representadas em um diagrama UML. Note que uma classe abstrata possui seu nome grafado em itálico.



Uma classe abstrata pode ter métodos concretos e abstratos, porém uma classe concreta não pode conter métodos abstratos. Ou seja, ao declararmos um método como abstrato, sua classe também deverá ser declarada abstrata.

### 8.3 INTERFACES

Uma interface é uma espécie de "classe totalmente abstrata", na qual todos os métodos são implicitamente públicos e abstratos e todos os atributos são implicitamente *static final*.

Interfaces são criadas para fornecer serviços comuns para classes não relacionadas<sup>1</sup>. Assim, podemos usar uma interface para amarrar elementos de várias classes em um conjunto com comportamento uniforme, mesmo que estas classes não possuam nenhum relacionamento.

Uma interface é declarada através da palavra *interface* no lugar de *class*. Embora não represente exatamente uma classe, deve ser gravada em um arquivo com extensão `.java` e compilada.

<sup>1</sup> Não confundir o termo *interface* aqui utilizado com Interface Gráfica com o Usuário (GUI), assunto que será visto no capítulo 12.

```
public interface MinhaInterface {
    // atributos estáticos constantes
    // assinatura dos métodos
}
```

Para que uma classe possa usar uma interface, é necessário que a classe implemente a interface com o uso da palavra reservada *implements*:

```
public class MinhaClasse implements MinhaInterface {
    // atributos específicos da classe
    // métodos específicos da classe
    // métodos da interface implementados
}
```

Uma interface é como um contrato: se uma classe implementa uma interface, então ela deve, obrigatoriamente, implementar todos os seus métodos (pois são todos abstratos).

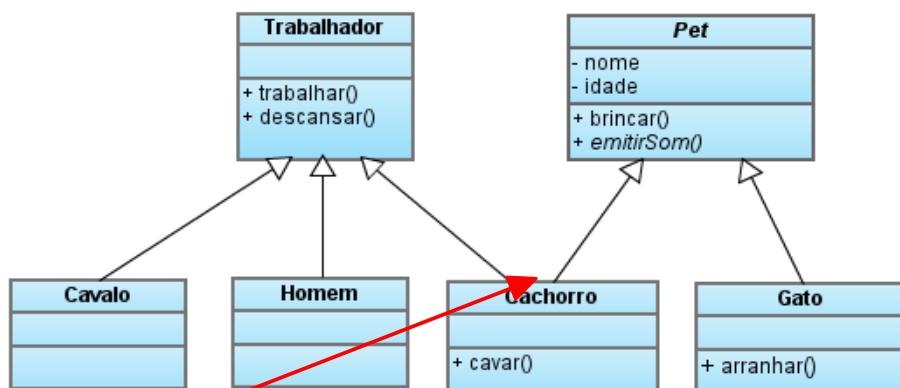
Uma interface não pode ser instanciada, mas podem ser definidas referências do seu tipo:

```
MinhaInterface ref; // definindo uma referência sem instanciar objeto
```

Interfaces podem ser herdadas (ou seja, pode-se ter uma sub-interface). Uma interface pode estender mais de uma interface, diferentemente de uma classe, que pode estender somente uma classe:

```
public interface MinhaInterface extends OutraInterfaceX, OutraInterfaceY
```

Utilizando o exemplo apresentado anteriormente, suponha que queremos representar o comportamento de uma classe **Trabalhador** através dos métodos **trabalhar()** e **descansar()**. Poderíamos fazer com que todos animais que trabalham (Homem, Cachorro, Cavalo, etc) fossem subclasses de **Trabalhador** e herdassem estes 2 métodos. No entanto, se uma classe já é subclasse de outra (como **Cachorro**, que já é subclasse de **Pet**), ela não poderá estender também outra classe. Em Java, não é permitido que uma classe herde de duas ou mais classes (herança múltipla).



Java não permite herança múltipla

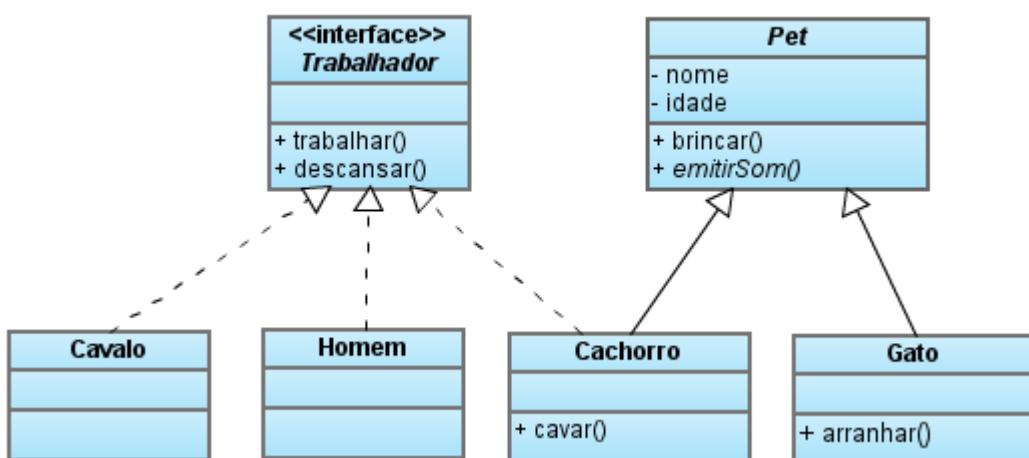
Colocar os métodos **trabalhar()** e **descansar()** na classe **Pet** também não é uma boa solução, uma vez que **Cavalo** e **Homem** também são trabalhadores, mas não são tipos de **Pet**.

A solução é definir **Trabalhador** como uma Interface. Assim, qualquer classe poderá implementá-la, mesmo que não haja relacionamento nenhum entre elas. Ao implementar

Trabalhador, as classes serão obrigadas a implementarem os métodos `trabalhar()` e `descansar()`. Assim garante-se a uniformidade de comportamento, ou seja, que todo **Trabalhador**, independentemente de que tipo seja, tenha esses dois métodos.

Cabe ressaltar que uma classe só pode herdar de uma classe, porém, pode implementar quantas interfaces for preciso.

Uma interface é representada num diagrama UML através do estereótipo `<<interface>>` antes de seu nome. Da mesma forma que uma classe abstrata, o nome deve aparecer em itálico. Para indicar que uma classe implementa uma interface, usa-se uma seta semelhante à usada para herança, porém com linha tracejada.



Veja que não há relação nenhuma de herança entre um cavalo, um homem e um cachorro, portanto são objetos não relacionados. Porém, ainda assim eles apresentam serviços em comum.

As interfaces também promovem o relacionamento "é-um" com quem a implementa, ou seja, no exemplo acima, "Cavalo é um Trabalhador", "Homem é um Trabalhador" e "Cachorro é um Trabalhador".

Vejamos como ficaria o código da interface `Trabalhador`:

```

public interface Trabalhador {
    void trabalhar();
    void descansar();
}
  
```

A classe `Cavallo` ficaria assim:

```

public class Cavallo implements Trabalhador {
    public void trabalhar(){
        System.out.println("Puxando carroça");
    }
    public void descansar(){
        System.out.println("Cavalo descansando");
    }
}
  
```

Já a classe `Cachorro` deve herdar de `Pet` (e portanto, sobrepor o método abstrato `emitirSom()`) e implementar `Trabalhador` (e portanto, implementar os métodos `trabalhar()` e `descansar()`).

```
public class Cachorro extends Pet implements Trabalhador {  
    public void emitirSom(){  
        System.out.println("Au au au");  
    }  
    public void cavar(){  
        System.out.println("Cavando");  
    }  
    public void trabalhar(){  
        System.out.println("Cão em serviço");  
    }  
    public void descansar(){  
        System.out.println("Cão descansando");  
    }  
}
```

## 8.4 RESUMO DO CAPÍTULO

- Uma classe concreta é uma classe que pode ser instanciada.
- Uma classe abstrata não pode ser instanciada; ela é criada unicamente para servir de superclasse para outras classes.
- Um método abstrato é um método que não possui corpo.
- Se declararmos um método como abstrato, sua classe também deverá ser declarada abstrata.
- As subclasses de uma classe abstrata devem sobrepor todos os seus métodos abstratos, fornecendo-lhes um corpo.
- Todos os atributos e métodos concretos de uma classe abstrata são herdados normalmente por suas subclasses.
- Uma classe abstrata é um meio termo entre uma classe concreta (que contém uma implementação para todos os métodos declarados) e uma interface (na qual nenhum dos métodos declarados é implementado).
- Uma interface não pode ser instanciada.
- Uma interface pode conter apenas atributos constantes, que já são declarados implicitamente como `public static final`.
- Todos os métodos de uma interface são implicitamente `public abstract`, ou seja, não possuem corpo.
- Classes abstratas e interfaces podem ser usadas para definir variáveis de referência.

## 8.5 BIBLIOGRAFIA DO CAPÍTULO

RUSSELL, John W. M.. **Tutorial 5 – Inheritance and Polymorphism**. Disponível em : <<http://home.cogeco.ca/~ve3ll/jatutor5.htm>>. Acesso em 04/05/2011.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

## 9 Polimorfismo

O termo polimorfismo significa “muitas formas” e é um dos conceitos mais importantes da orientação a objetos, ao lado da abstração, do encapsulamento e da herança. Enquanto a herança trata do estabelecimento de uma hierarquia de classes, o polimorfismo diz respeito ao comportamento quando da invocação dos métodos sobrepostos.

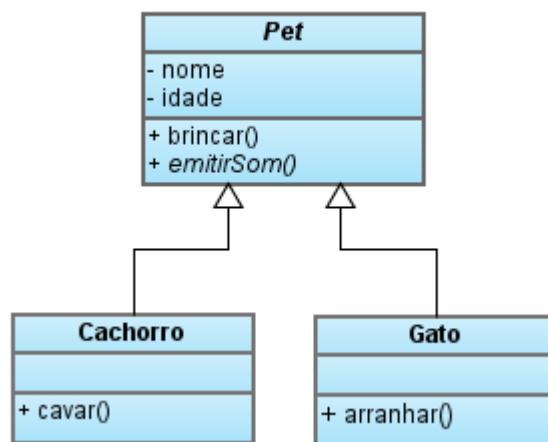
Segundo o polimorfismo, uma classe pode possuir subclasses que respondam de diferentes formas quando um determinado método é invocado. Assim, um mesmo método pode executar ações diferentes, com base no objeto sobre o qual está atuando. Os objetos serão referenciados por variáveis do tipo da superclasse, o que torna possível abstrair detalhes das classes mais especializadas, mascarando-os através de uma interface comum (a da superclasse).

De acordo com Ricarte (2000),

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia.

Este é a forma mais básica e usual de polimorfismo, também conhecida como polimorfismo por inclusão. Para que ele se configure, é necessário que exista uma relação de herança entre as classes envolvidas e que haja *override* (sobreposição) de algum método. Lembramos que o conceito de *override* já foi abordado na seção 7.2.

Para ilustrar, vamos utilizar como exemplo a herança de classes apresentada no Capítulo 8 (**Pet**, **Cachorro** e **Gato**), a qual replicamos abaixo. Note que o método `emitirSom()` é abstrato, ou seja, deve obrigatoriamente sofrer *override* nas subclasses. Com isso, a classe **Pet** também deve ser abstrata. Porém, o mecanismo de polimorfismo que vamos ilustrar funciona igualmente com superclasses e métodos concretos.



```

public abstract class Pet {
    private String nome;
    private int idade;
    public void setNome(String nome) {
        this.nome = nome;
    }
}
  
```

```

public String getNome(){
    return this.nome;
}
public void brincar(){
    System.out.println(nome+ " brincando");
}
public abstract void emitirSom();
}

```

```

public class Cachorro extends Pet {
    public void cavar(){
        System.out.println("Cavando");
    }
    @Override
    public void emitirSom(){
        System.out.println("Au au au");
    }
}

```

```

public class Gato extends Pet {
    public void arranhar(){
        System.out.println("Arranhando");
    }
    @Override
    public void emitirSom(){
        System.out.println("Miau");
    }
}

```

Temos acima uma superclasse e 2 subclasses que sobrepõe um método. Mas ainda não estamos usando polimorfismo.

Vamos agora criar uma classe de teste que contenha um vetor de 10 posições do tipo `Pet` (ou seja, do tipo da superclasse). Dentro deste vetor, instanciaremos objetos `Cachorro` para as posições com índice par e objetos `Gato` para as posições de índice ímpar. Depois, chamaremos o método `emitirSom()`, que será uma chamada polimórfica, ou seja, a definição de qual implementação do método `emitirSom()` será executada será feita em tempo de execução, de acordo com o tipo de objeto que o chamar (se Cachorro ou Gato).

```

public class PetTeste {
    public static void main(String[] args) {
        Pet vetor[] = new Pet[10]; // o vetor é do tipo da superclasse!
    }
}

```

```

        for (int i = 0; i < vetor.length; i++){
            if(i%2 == 0){
                vetor[i] = new Cachorro(); // isso se chama upcasting
            }
            else {
                vetor[i] = new Gato(); // isso se chama upcasting
            }
            vetor[i].emitirSom(); // chamada polimórfica
        }
    }
}

```

Mas, se vetor é do tipo Pet, não deveríamos escrever apenas `vetor[i] = new Pet()`? Posso escrever `vetor[i] = new Cachorro()`, sendo que vetor não é do tipo Cachorro?

A resposta é sim! Quando definimos um supertipo, objetos de qualquer de seus subtipos poderão ser atribuídos onde o supertipo for esperado. Em outras palavras, quando declararmos uma variável de referência (ou um vetor, como é o caso do exemplo), qualquer objeto que passar no teste “é-um” quanto ao tipo declarado poderá ser atribuído a esta referência. Como `Cachorro` “é um” `Pet`, então qualquer variável de referência ou vetor do tipo `Pet` pode receber um objeto `Cachorro`, bem como `Gato` ou qualquer outra subclasse de `Pet` que venhamos a criar no futuro. Isso se chama *upcast* (falaremos mais dele adiante).

O polimorfismo ocorre, de fato, na linha

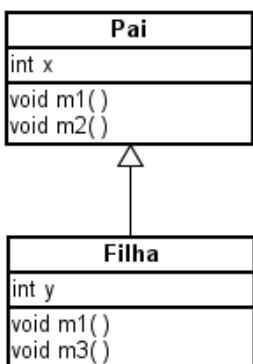
```
vetor[i].emitirSom();
```

A chamada ao método `emitirSom()` é a mesma. Porém, qual será o resultado? Será impresso “Au au au” ou “Miau”? Depende. Se naquela posição do vetor houver um objeto `Cachorro`, será executada a implementação de `emitirSom()` redefinida na classe `Cachorro` (Au au au). Se for um `Gato`, será executada a implementação de `emitirSom()` redefinida na classe `Gato` (Miau). Perceba, então, que o comando `vetor[i].emitirSom()` gera resultados diferentes, dependendo do tipo do objeto. Ou seja, o mesmo método se comporta de “várias formas” (neste ponto, recomenda-se reler a definição de polimorfismo presente no início deste capítulo).

## 9.1 LIGAÇÃO TARDIA (LATE BINDING)

Ligaçāo tardia (ou ligação dinâmica) é a capacidade de um programa de resolver referências a métodos de subclasse em tempo de execução (*runtime*). No exemplo anterior, a decisão de qual implementação de `emitirSom()` será executada é tomada durante a execução do programa.

Suponha agora as classes `Pai` e `Filha` ilustradas abaixo:



```

class Pai {
    int x;
    void m1(){
        System.out.println("Executando m1 da classe pai");
    }
    void m2(){
        System.out.println("Executando m2 da classe pai");
    }
}

```

```

class Filha extends Pai {
    int y;
    void m1(){
        System.out.println("Executando m1 da classe Filha");
    }
    void m3(){
        System.out.println("Executando m3 da classe Filha");
    }
}

```

Veja que a classe `Filha` estende `Pai`, portanto, herdará todos os atributos e métodos de `Pai`. Adicionalmente, `Filha` possui um atributo `y`, um método `m3()` e sobrescreve o método `m1()`.

Neste caso, dois objetos, sendo um da superclasse e outro da subclasse, responderão diferentemente à mesma chamada do método `m1()`. A decisão sobre qual método executar, o da superclasse ou o da subclasse, ocorre em tempo de execução. Ou seja, a chamada do método permanece a mesma, o que varia é o que foi instanciado com o operador `new`. Se o objeto for do tipo da superclasse, o método executado será o da superclasse; se o objeto criado for do tipo da subclasse, o método executado será o da subclasse. Assim, independentemente do tipo do objeto, a chamada do método permanece a mesma.

## 9.2 CONVERSÃO (CASTING)

Segundo Mendes (2008), “a operação de casting é usada quando o objetivo é ajustar o retorno de um método com a atribuição a uma variável”, o que “só é possível quando os tipos de dados são compatíveis entre si (por exemplo com o uso de herança)”. Ainda segundo o autor, podemos converter tipos mais específicos em mais genéricos (*upcasting*) ou mais genéricos em mais específicos (*downcasting*).

### 9.2.1 Upcasting

A instrução

```
Pai p;
```

declara `p` como uma referência da superclasse `Pai`. Isto define a vocação de `p`: referenciar qualquer objeto do tipo `Pai`. Atente, agora, para a seguinte regra:

**Regra 1:** Em Java, podemos atribuir um objeto da subclasse a uma referência de sua superclasse. (Esta operação é chamada *upcasting*, de *up type casting*)

A regra 1 é razoável, pois todo objeto da subclasse É UM objeto da sua superclasse. Então, podemos atribuir a `p` um objeto `Filha`:

```
p = new Filha();
```

Agora `p`, uma referência de superclasse, está apontando um objeto de subclasse. A vocação de `p` não foi contrariada, pois o objeto `Filha` também “é um” objeto `Pai`.

Observe, a seguir, `p` acionando métodos:

```
p.m1(); //chama m1() de Filha, porque ela sobrescreveu o m1 de Pai
p.m2(); //chama m2() de Pai, herdado por Filha
```

Porém, cuidado com a chamada seguinte:

```
p.m3(); //ERRO de compilação
```

Este último comando foi uma tentativa de acessar, através de `p` (referência de superclasse) um membro exclusivo da subclasse, o que contraria a vocação de `p`, expressa na regra 2:

**Regra 2:** Uma referência de superclasse só reconhece membros disponíveis na superclasse, mesmo que esteja apontando para um objeto de subclasse.

Resumindo, o *upcasting* é a conversão de um objeto de tipo mais específico para um tipo mais genérico, feita implicitamente através de atribuição. A partir da conversão, somente os membros do tipo mais genérico podem ser acessados sem artifícios.

### 9.2.2 Downcasting

Como podemos então acessar o método `m3()` de `p`? A resposta está no *downcasting*.

**Regra 3:** Em Java, a atribuição de um objeto de superclasse a uma referência de subclasse, sem uma coerção explícita, não é permitida.

Ex:

```
Filha f = p; // ERRO de compilação
```

Isto parece, também, razoável, pois `f` tem vocação de referenciar e saber coisas que não existem no objeto Pai (o atributo `y` e o método `m3`). Ou ainda, a relação É UM tem mão única, da subclasse para a superclasse, portanto, um objeto Pai não “é um” objeto Filha. Há casos, todavia, em que podemos “forçar a barra” através de coerção, se soubermos que o objeto atualmente com referência de superclasse é, na realidade, um objeto da subclasse para a qual estamos convertendo. Assim, poderíamos chamar o método `m3()`.

```
Filha f = (Filha) p; // nome da classe destino entre parênteses
f.m3();
```

Esta realidade, citada na frase anterior, é algo verificado por Java apenas em tempo de execução. Se o objeto for do tipo da subclasse, a coerção será válida, mas se não for, ocorrerá uma `ClassCastException`. Para proteger nosso código dessa incerteza, convém usar o operador especial `instanceof`.

### 9.2.3 O operador instanceof

O operador `instanceof` é usado para determinar, no momento da execução do programa, se um objeto Java é de um determinado tipo. Possui a seguinte sintaxe:

```
variável objeto instanceof Classe
```

Este operador retorna `true` se a variável `objeto` é do tipo da `Classe`, e `false` em caso contrário. Assim, nossa coerção anterior ficaria mais segura se codificada assim:

```
if (p instanceof Filha) {
    Filha f = (Filha) p;
    f.m3();
}
```

O downcast só será válido se, em tempo de execução, o objeto tiver um relacionamento É UM com o tipo dado entre parênteses – ou seja, (`C`) ref só vale se ref É UM `C`.

No exemplo dos Pets, poderíamos por exemplo chamar o método `cavar()` dos objetos

Cachorro que estão no vetor. Para isso, é necessário testar se o objeto é de fato um cachorro:

```
for (int i = 0; i < vetor.length; i++){
    ...
    if(vetor[i] instanceof Cachorro){
        Cachorro c = (Cachorro)vetor[i]; // downcasting
        c.cavar();
    }
}
```

### 9.3 ARGUMENTOS POLIMÓRFICOS

O conceito de *upcasting* também se aplica aos argumentos que são passados aos métodos e aos tipos de retorno dos mesmos. Isto é, se um parâmetro é declarado na assinatura de um método como sendo do tipo da superclasse, então também poderá receber como argumento referências a objetos de suas subclasses.

Veja o exemplo abaixo. A classe `Veterinário` possui um método `cuidar` que possui um parâmetro do tipo `Pet`. Com isso, o método `cuidar` pode receber um cachorro, um gato ou um objeto de qualquer outra possível subclass de `Pet`. Se criarmos uma classe `Papagaio` que estende `Pet`, por exemplo, o método `cuidar` não precisará sofrer nenhuma alteração.

```
public class Veterinario {
    public void cuidar(Pet paciente) {
        System.out.println("Cuidando do "+paciente.getNome());
        if (paciente instanceof Cachorro)
            System.out.println("Tirando as pulgas");
    }
}
```

Vamos testá-la com a seguinte classe:

```
public class PetTesteVeterinario {
    public static void main(String[] args) {
        Cachorro dog = new Cachorro();
        dog.setNome("Rex");

        Gato cat = new Gato();
        cat.setNome("Mimi");

        Veterinario vet = new Veterinario();

        vet.cuidar(dog); // passando argumento do tipo Cachorro
        vet.cuidar(cat); // passando para o mesmo método um argumento do
        tipo Gato
    }
}
```

## 9.4 POLIMORFISMO COM INTERFACES

Podemos fazer uso do polimorfismo também com **interfaces**, afinal, uma classe que implementa uma interface também estabelece a relação "é um" com a interface. No exemplo abaixo, estamos criando um vetor de objetos do tipo **Trabalhador** (interface que criamos no Capítulo 8.3). Isto feito, podemos preencher este vetor com qualquer objeto que implemente a interface Trabalhador, como **Cachorro** e **Cavalo**.

```
class TrabalhadorTeste {
    public static void main(String[] args) {
        Trabalhador vetor[] = new Trabalhador[10]; // o vetor é do tipo da interface

        for (int i = 0; i < vetor.length; i++) {
            if(i%2 == 0){
                vetor[i] = new Cachorro();
            }
            else {
                vetor[i] = new Cavalo();
            }

            vetor[i].trabalhar(); //POLIMORFISMO
            vetor[i].descansar(); //POLIMORFISMO
        }
    }
}
```

## 9.5 SOBRECARGA (OVERLOADING)

Isto não é uma unanimidade: a sobrecarga de métodos ou *overloading*, já vista quando tratamos de construtores (seção 4.6), é considerada por alguns autores uma forma de polimorfismo que ocorre em tempo de compilação ou *ad hoc*.

Na sobrecarga, ao declarar múltiplos métodos que possuam o mesmo nome, o compilador selecionará o método apropriado escolhendo aquele cujo nome, número e tipo de parâmetros coincidir com a chamada de um método com o mesmo nome, número e tipo de argumentos (através da comparação de assinaturas).

Veja o exemplo seguinte:

```
void imprimir(){
    System.out.println("Impressão padrão");
}

void imprimir(String argumento){
    System.out.println(argumento);
}
```

O exemplo anterior demonstra dois métodos **imprimir** sobrecarregados. Os dois métodos possuem o mesmo nome, porém, parâmetros diferentes. Ao se chamar **imprimir()** o primeiro é chamado (executado). Ao se chamar **imprimir("Olá")**, o segundo é chamado. Deste modo, mesmo antes da execução do programa, já saberemos qual versão do método será executada.

Podemos realizar quantas sobrecargas forem necessárias. Porém, é importante lembrar que os argumentos devem ser sempre diferentes. Não basta você modificar o tipo de retorno, mantendo o resto da assinatura igual. Veja o exemplo:

```
int somar(int x, int y){  
    return x + y;  
}  
  
long somar(int x, int y){  
    return x + y;  
}
```

Neste último caso o compilador não tem como decidir qual método chamar, e acusará um erro.

## 9.6 RESUMO DO CAPÍTULO

- O polimorfismo possibilita que um método resulte em diferentes comportamentos, dependendo do objeto para o qual é aplicado.
- Uma referência do tipo da superclasse pode ser vinculada a objetos de suas subclasses, mudando assim seu comportamento conforme o tipo de objeto que lhe foi atribuído. Isso possibilita que objetos das diferentes subclasses sejam vistos como se fossem da superclasse, abstraindo seus detalhes e criando uma espécie de protocolo padronizado para tratamento de objetos de quaisquer das subclasses. Isso gera economia de código, pois evita replicar a mesma implementação para cada classe específica, implicando consequentemente em maior facilidade de manutenção.
- É necessário que exista herança e que o método em questão tenha sido redefinido (*override*).
- A escolha de qual implementação do método redefinido será se dá apenas em tempo de execução, conforme o objeto referenciado naquele momento. (*late binding* ou ligação tardia)
- É indiferente se a superclasse e o método em questão são concretos ou abstratos.
- As principais vantagens do polimorfismo são a simplificação de códigos e a escalabilidade, pois ao tratar objetos de várias subclasses como se fossem da superclasse, torna-se possível adicionar novas subclasses sem a necessidade de modificar ou reescrever códigos que já estão preparados para tratar com o tipo da superclasse.
- A sobrecarga (mesmo método com parâmetros diferentes) é considerada por alguns autores como uma outra forma de polimorfismo, este em tempo de compilação.

## 9.7 BIBLIOGRAFIA DO CAPÍTULO

RUSSELL, John W. M.. **Tutorial 5 – Inheritance and Polymorphism**. Disponível em : <<http://home.cogeco.ca/~ve3ll/jatutor5.htm>>. Acesso em 04/05/2011.

MENDES, Douglas Rocha. **Programação Java com Ênfase em Orientação a Objetos**. São Paulo: Novatec, 2009.

RICARTE, Ivan Luiz Marques. **Polimorfismo**. 2000. Disponível em <<http://www.dca.fee.unicamp.br/courses/PooJava/polimorf/index.html>>. Acesso em 24/11/2016.

## 10 Tratamento de Exceções

O termo exceção refere-se a algo excepcional que aconteceu em determinado contexto, como uma falha de funcionamento. Algumas falhas são plausíveis de acontecer em softwares, como *overflow* em uma soma, esgotamento da memória do sistema, abrir um arquivo inexistente, etc. O programa deve ser capaz de detectar esse erro e, se possível, tomar as medidas cabíveis para voltar a um estado estável e funcional.

O conceito de exceção é simples: se algum erro acontece (ex.: acesso a um arquivo inexistente), uma exceção é lançada (*throw*) e o código que estava sendo executado **para** de ser processado. A exceção lançada (que é por sua vez um objeto) contém informações referentes ao erro, como uma mensagem que o descreve (ex.: "arquivo teste.txt não existe").

Depois que um exceção é lançada, o sistema (no caso a JVM) tenta encontrar algo que seja capaz de manusear e tratar essa exceção. Os possíveis candidatos a essa tarefa serão os métodos que foram chamados anteriormente até que o código que gerou o erro fosse executado. Dessa forma, se a ordem de chamada dos métodos foi `main() → abreCadastro() → abreArquivo()` e o método `abreArquivo()` foi quem gerou a exceção, os métodos analisados serão `abreCadastro()` e `main()` (nessa ordem, que é do mais recente para o mais antigo chamado). Um método se qualifica a capturar a exceção (*catch*) se ele contiver um **código especial** para isso. Se ele não possuir esse código, o próximo método na lista de candidatos será analisado. O processo se repete até que alguém apto seja encontrado.

Se um método for capaz de tratar a exceção, a execução do programa (que havia sido interrompida pela exceção) tem continuidade no exato ponto onde se inicia o código de tratamento da exceção. Se nenhum método for capaz de tratar a exceção, ela irá viajar por toda a lista até chegar na JVM (que foi que iniciou o programa). Nesse caso, a JVM termina o programa e mostra uma mensagem de erro, como a seguinte:

```
Exception in thread "main" java.lang.NoClassDefFoundError: Teste1
Caused by: java.lang.ClassNotFoundException: Teste1
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:248)
```

A primeira linha da mensagem mostra que houve uma exceção na *thread* chamada "main", sendo que o tipo da exceção é `java.lang.NoClassDefFoundError` (veremos adiante sobre tipos de exceções). Abaixo dessa mensagem, há a lista de métodos que foram percorridos, sendo que nenhum deles foi capaz de tratar a exceção. O método que causou a exceção está indicado na terceira linha (método `run()` da classe `URLClassLoader`, que está na linha 202 do arquivo `URLClassLoader.java`). Os métodos abaixo desse foram os métodos testados (na ordem de cima para baixo) em busca de alguém apto a tratar a exceção.

### 10.1 COMO TRATAR EXCEÇÕES

O código especial (chamado tratador de exceção) que a JVM busca para que um método seja capaz de tratar uma exceção é o bloco `try...catch`. O exemplo abaixo ilustra a utilização do bloco:

```
try {
    // código(s) que pode(m) gerar exceção(ões)

} catch () {
    // código para tratar a exceção
}
```

O bloco é composto por duas partes importantes: o `try` (que contém os códigos que podem gerar exceção) e o `catch` (que contém o código que será executado para tratar alguma exceção levantada dentro do `try`). Abaixo há um exemplo de código:

```
private List<Integer> lista;
PrintWriter p = null;

try {
    p = new PrintWriter(new FileWriter("teste.txt"));
    for (int i = 0; i < 5; i++) {
        out.println("Valor em: " + i + " = " + lista.get(i));
    }
} catch (IOException e) {
    System.err.println("Pegamos uma exceção: " + e.getMessage());
}

System.out.println("Continuando...");
```

No código acima, o construtor da classe `PrintWriter` pode levantar uma exceção para indicar algum erro, como impossibilidade de ler do disco. Se esse erro acontecer, o construtor jogará a exceção e a execução do código será interrompida imediatamente. Isso quer dizer que o `for()` (abaixo da instanciação do objeto `p`) não será executado. Como a exceção foi jogada dentro de um bloco `try`, ela será pega pelo `catch` no momento que for jogada, e o código do `catch` será executado (no caso, apenas um `println()` dizendo que pegamos uma exceção); perceba que a JVM não precisa procurar em um método anterior pelo tratador de exceção, porque o código atual está apto a fazer esse tratamento. A exceção poderá ser manipulada através do objeto `e` (declarado dentro do `catch`), que contém as informações da exceção jogada pelo construtor da classe `PrintWriter`.

Se o construtor da classe `PrintWriter` não levantar uma exceção, o código do `try` será executado inteiramente: o objeto `p` será instanciado e o `for()` será executado normalmente. Quando ele terminar, o código do programa seguirá executando (no caso, o `println("Continuando...")` será executado). Nesse caso, o código dentro do `catch` **não** será executado, visto que nenhuma exceção foi jogada.

O bloco `try...catch` é o responsável por conter e tratar uma exceção jogada (o código dentro do `try` pode jogar a exceção, sendo que o `catch` a pega e trata). Se a exceção for jogada e não houver um `try...catch` envolvendo o método que gerou a exceção, ela será jogada novamente para o método anterior (que foi quem chamou o método atual). Se esse método não possuir um `try...catch` também, a exceção será jogada novamente para o próximo método. Se ninguém tratar a exceção, ela cairá no `try...catch` mais externo existente, que é o da JVM (que em seu `catch` apenas imprime o erro e encerra o programa).

### 10.1.1 Tratamento múltiplo

Determinados métodos podem jogar mais de um tipo de exceção, especialmente quando desempenham tarefas complexas. Um exemplo é o método que obtém um elemento de uma lista, que pode gerar as seguintes exceções (exemplo): elemento não encontrado, lista vazia ou

parâmetro de busca incorreto (esperado int e um float foi dado, por exemplo).

Quando o programa deve reagir de forma diferente para cada uma dessas possíveis exceções, é conveniente utilizar um `try` com vários `catches` (um para cada exceção a ser capturada). O código abaixo mostra o mesmo exemplo anterior, com a diferença que há mais de uma exceção sendo capturada.

```
private List<Integer> lista;
PrintWriter p = null;

try {
    p = new PrintWriter(new FileWriter("teste.txt"));
    for (int i = 0; i < 5; i++) {
        out.println("Valor em: " + i + " = " + lista.get(i));
    }
} catch (FileNotFoundException e) {
    System.err.println("Arquivo não encontrado! " + e.getMessage());
}

} catch (IOException e) {
    System.err.println("Pegamos uma exceção: " + e.getMessage());
}

System.out.println("Continuando...");
```

Se o construtor da classe `PrintWriter` não conseguir encontrar o arquivo, ele irá jogar uma exceção do tipo `FileNotFoundException`, que será pega e tratada pelo primeiro `catch`. O primeiro `catch` será utilizado porque a JVM irá comparar o `tipo` do objeto da exceção jogada com o `tipo` informado no `catch` (através do uso de `instanceof`). Se o *objeto da exceção jogada for uma instância da classe indicada no catch*, então esse `catch` será usado, caso contrário o próximo `catch` será testado. Se nenhum `catch` conseguir pegar a exceção, um erro de compilação será gerado.

É importante notar que o teste é feito com `instanceof` e que todos os conceitos de polimorfismo se aplicam aqui. Isso quer dizer que se `FileNotFoundException` herda de `IOException`, por exemplo, então qualquer objeto da classe `FileNotFoundException` também é uma instância de `IOException`. Desse modo, se os `catches` estivessem dispostos da seguinte forma:

```
try {

} catch (IOException e) {

} catch (FileNotFoundException e) {

}
```

o segundo `catch` **nunca** pegaria uma exceção que herda de `IOException`, porque o `catch` anterior pegaria todas elas.

A linguagem Java disponibiliza diversas classes que são exceções prontas para uso, como `FileNotFoundException`. Para poder ser jogada como uma exceção, a classe precisa herdar a classe `Throwable`<sup>2</sup> (que é a superclasse de todas as exceções). Dessa forma, se o programador deseja pegar qualquer exceção levantada dentro do `try` (independente do seu tipo), basta utilizar uma classe muito genérica no `catch`, como é o caso da `Throwable` ou da `Exception` (que herda de `Throwable` também).

2 A tradução do inglês *throwable* para português é "jogável".

## 10.2 O BLOCO FINALLY

O bloco `finally` é utilizado como complemento do `try...catch`. O bloco `finally` é **opcional** e é utilizado da seguinte forma:

```
try {
    ...
} catch (IOException e) {
    ...
} catch (Exception e) {
    ...
} finally {
    // código que será sempre executado, independente
    // de haver ou não exceções
}
```

O bloco `finally` **sempre** é executado, independente de ter havido ou não ter havido alguma exceção. Uma utilidade para esse é bloco é evitar duplicação de código entre o `try` e o `catch`. Por exemplo, se dentro do `try` houver um código para abrir um arquivo (ex.: `file.open()`), esse arquivo deve ser fechado depois que for usado. Se o comando para fechar o arquivo estiver no final do `try`, ele será executado se não houver exceções. Se alguma exceção for jogada, pode ser que o fechamento do arquivo não seja feito, porque a execução do código do `try` será interrompida. Para garantir que o arquivo seja fechado, o comando para fechar o arquivo deverá ser colocado também dentro do `catch`. Dessa forma, o arquivo é fechado se houver e se não houver exceções. O código abaixo exemplifica essa ideia (o código é fictício).

```
File f = new File();
f.open("teste.txt");

try {
    // manipula o arquivo...
    // faz outras coisas...

    f.close();
}

} catch (Exception e) {
    f.close();
}
```

Essa abordagem, porém, cria uma replicação de código. O comando para fechar o arquivo existe em dois locais diferentes, sendo que se o programador adicionar um novo `catch` (para tratar alguma exceção nova, por exemplo), ele terá de lembrar de colocar o `f.close()` dentro desse novo `catch`.

Para evitar isso, o fechamento do arquivo poderia ser feito dentro de um bloco `finally`. O bloco `finally` sempre será executado, independente do `try` e do `catch`. Logo, o código poderia ser refeito dessa forma:

```
File f = new File();
f.open("teste.txt");

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

```

} finally {

    f.close();
}

```

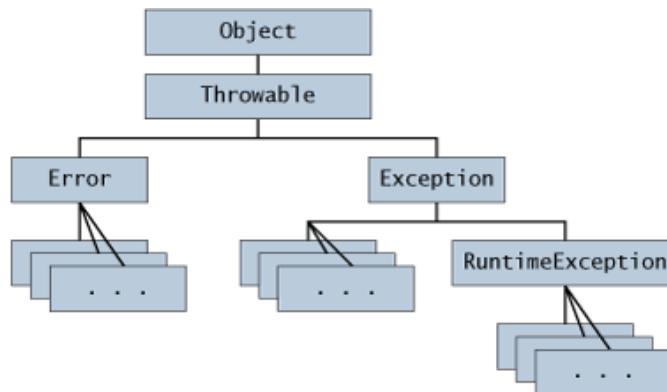
O bloco `finally` **não** é um prolongamento do `catch`, ele é na verdade um *complemento* do `try`; é um bloco de código que **sempre** será executado, seja após os comandos contidos num `try` ou após os comandos contidos num `catch`.

### 10.3 JOGANDO (OU LANÇANDO) EXCEÇÕES

O programador pode jogar uma exceção em qualquer momento do código, basta utilizar o comando `throw` seguido de um objeto jogável (que herde de `Throwable` ou suas subclasses), conforme o exemplo abaixo.

```
throw new Exception();
```

Conforme já mencionado, a linguagem Java possui diversas exceções prontas para serem utilizadas. O diagrama abaixo mostra a hierarquia de classes relacionadas a exceções.



No topo da hierarquia, encontra-se a classe `Object`, superclasse de todas as classes Java. Abaixo encontra-se a classe `Throwable`, que é a superclasse de qualquer classe que pode ser jogada através do comando `throw`.

As subclasses da classe `Error` (à esquerda na hierarquia) geralmente indicam algum erro muito grave e completamente anormal, possivelmente associado ao mau funcionamento da JVM. A classe `RuntimeException` e suas subclasses indicam erros que acontecem em tempo de execução, algo que não poderia ter sido previsto em tempo de compilação. Ambas as classes `Error` e `RuntimeException` (e suas subclasses) são ditas exceções **não checadas** (ou não verificadas), que indicam que o programador **não** é obrigado a utilizar um `try...catch` para conter qualquer código que possa levantar esse tipo de exceção. Além disso, se um método levanta esse tipo de exceção, essa informação não precisa estar especificada na declaração do método.

A classe `Exception` (e suas subclasses, excluindo-se a `RuntimeException` e suas subclasses), por sua vez, é uma exceção **checada** (ou verificada), o que quer dizer que qualquer código que levante esse tipo de exceção **deve** ser envolvido por um `try...catch`. Além disso, na declaração do método que levanta a exceção, é necessário que o programador informe qual o tipo de exceção (ou exceções) que o método levanta.

Se um método jogar uma exceção do tipo `Exception`, ele precisa explicitamente informar

isso, caso contrário o código não compilará. Para informar isso, utiliza-se a palavra-chave `throws` ao lado do método (e antes de seu código), como mostra o exemplo abaixo.

```
public void escreveInfoArquivo() throws IOException {
    PrintWriter out = new PrintWriter(new FileWriter("teste.txt"));
    out.print("Meu arquivo teste");
    out.close();
}
```

No exemplo acima, o construtor da classe `PrintWriter` pode levantar uma exceção do tipo `IOException` (ou derivado) para indicar algum erro. Como a classe `IOException` herda de `Exception`, ela é uma exceção checada, então tem-se duas opções: 1) envolver a instanciação do objeto `out` com um `try...catch` (para pegar e tratar a exceção) ou 2) repassar a exceção adiante para ser tratada pelo método que chamou o método `escreveInfoArquivo()`. Se a solução 1 for escolhida, o método não precisa dizer que joga uma exceção, porque a exceção está sendo pega e tratada dentro do próprio método. Se a solução 2 for utilizada, então o método precisa informar que joga uma exceção; sendo assim, ele precisa utilizar a palavra-chave `throws` e informar qual tipo de exceção joga (`IOException`, no caso do exemplo).

Em determinados casos, um método pode levantar mais de uma exceção. Se elas forem exceções checadas, o método precisa informar que levanta todas essas exceções, como no exemplo abaixo (as exceções levantadas foram colocadas uma embaixo da outra por questões de legibilidade).

```
public void fazAlgo() throws ExcecaoParamErrado,
                           ExcecaoOrdemErrada,
                           ExcecaoImpossivelConverter {
    // código do método...
    if(teste()) {
        throw new ExcecaoParamErrado();
    }

    // ... mais código
    if(a == 1) {
        throw new ExcecaoOrdemErrada();
    }

    // ... mais código
    throw new ExcecaoImpossivelConverter();
}
```

É importante notar que os conceitos de polimorfismo também se aplicam aqui. Da mesma forma que um bloco `catch` pode utilizar uma classe alta na hierarquia para poder pegar todas as exceções (independente do seu tipo), um método pode especificar que joga apenas um tipo de exceção (uma classe genérica, como `Exception`), ao invés de informar que joga diversas exceções.

Utilizando-se o código do exemplo anterior como base e imaginando que `ExcecaoParamErrado`, `ExcecaoOrdemErrada` e `ExcecaoImpossivelConverter` herdam da classe `Exception`, o método poderia ser reescrito da seguinte forma:

```
public void fazAlgo() throws Exception {
    // código do método...
    if(teste()) {
```

```

        throw new ExcecaoParamErrado();
    }

    // ... mais código
    if(a == 1) {
        throw new ExcecaoOrdemErrada();
    }

    // ... mais código
    throw new ExcecaoImpossivelConverter();
}

```

Embora essa seja uma abordagem possível, ela deixa o código confuso e vai contra a ideia de encapsulamento. Um programador deve ser capaz de prever o que um método vai fazer apenas lendo seu nome (e, consequentemente, quais exceções ele levanta). Se um método levanta diversos tipos de exceções, mas informa apenas que levanta uma exceção genérica, ele pode ser facilmente interpretado de maneira errada pelo programador que for utilizá-lo. A palavra-chave `throws` ajuda a identificar quais são os possíveis erros que o método pode encontrar na hora de desempenhar sua tarefa; a classe que for utilizá-lo saberá quais são esses erros, podendo reagir de forma diferente para cada um deles.

## 10.4 RESUMO DO CAPÍTULO

- Exceções são utilizadas para separar a lógica de ação da tarefa da lógica de tratamento de erros. O código de ação fica no `try`, enquanto o código de tratamento de erro fica no `catch`.
- No momento em que uma exceção é jogada, o código para de ser executado e só volta a ser processado quando alguém pegar e tratar a exceção, sendo que a execução continua a partir desse código de tratamento.
- Para jogar uma exceção, basta utilizar o comando `throw` seguido de um objeto jogável (que seja de uma classe que herde de `Throwable`).
- O bloco `finally` é opcional, porém se for utilizado terá seu código executado **sempre**, independente de ter havido ou não exceções.
- Se uma classe de exceção herdar de `Error` ou `RuntimeException`, ela se torna uma exceção não checada; nesse caso, o código que joga essa exceção não precisa ter um `try...catch`; um método que joga essa exceção não precisa informar isso através da palavra-chave `throws`.
- Se uma classe herda de `Exception`, ela é dita uma exceção checada. Nesse caso, o código que levanta a exceção precisa ser envolvido por um `try...catch`. Se um método levantar essa exceção e não tratá-la (com `try...catch`), ele precisa informar que levanta uma exceção através da palavra-chave `throws`.
- Utilizando-se polimorfismo, é possível pegar qualquer exceção jogada através de um `catch` com uma classe genérica.

## 10.5 BIBLIOGRAFIA DO CAPÍTULO

ORACLE. Lesson: Exceptions (The Java Tutorials). Disponível em <<http://download.oracle.com/javase/tutorial/essential/exceptions/index.html>>. Acesso em 18/05/2011.

# 11 Pacotes e Arquivos JAR

## 11.1 PACOTES

Uma aplicação pequena em Java possui algumas poucas classes. Em contra partida, uma aplicação mais complexa pode apresentar dezenas, até centenas de classes. Com o reaproveitamento de classes entre uma aplicação e outra, podem haver conflitos nos nomes das classes, sem mencionar que encontrar uma classe específica num mar de centenas delas não é uma tarefa fácil.

Para solucionar todos esses problemas, programadores podem agregar grupos de classes com tipos relacionados em **pacotes**. Em uma visão prática, pacotes funcionam como bibliotecas: um pacote chamado `grafico` provavelmente contenha diversas classes relacionadas com gráficos; um pacote chamado `matematica` provavelmente contenha classes para cálculos matemáticos diversos.

Pacotes servem para organizar as classes por afinidade e hierarquia, tornando fácil o trabalho de localização de funcionalidades e compartilhamento de código entre programadores. Uma classe pertencente a um pacote tem seu nome prefixado pelo nome do pacote; isso quer dizer que uma classe `Retangulo`, por exemplo, pertencente ao pacote `geometria` na verdade se chamada `geometria.Retangulo` (é a classe `Retangulo` do pacote `geometria`). Esse é o nome qualificado completo da classe (*fully qualified name*). Isso ajuda a evitar o conflito de nomes entre classes. É muito provável que outro programador já tenha criado uma classe chamada `Retangulo`, porém se ela pertencer a outro pacote, ambas as classes podem coexistir, visto que seus nomes são prefixados pelo nome de seus pacotes (`geometria.Retangulo` é uma classe, enquanto `formas.Retangulo` é outra).

A nomenclatura de pacotes geralmente segue algumas convenções. Uma delas é que um pacote geralmente é formado por diversos subpacotes, afim de organizar ainda melhor as classes. Imagine o exemplo do pacote `geometria`: `Retangulo` é uma forma geométrica, porém um pacote chamado `geometria` possui diversas formas. Podemos agrupá-las em formas retas (triângulos, quadrados, etc) e formas curvas (círculos, elipses, etc). Dessa forma, nosso pacote `geometria` poderia ter dois subpacotes: `retos` e `curvos`. A hierarquia de pacotes utiliza a notação do ponto para indicar nível, assim como fazemos com classes e membros (ex.: `System.out.println()`). Nossos pacotes seriam, então, `geometria.retos` e `geometria.curvos`.

Quando uma classe faz parte de um pacote, a primeira linha de código dessa classe deve informar isso. O exemplo abaixo mostra a classe `Retangulo`, do pacote `geometria.retos`:

```
package geometria.retos;

public class Retangulo {
    // código da classe
}
```

A primeira linha do arquivo `Retangulo.java` é a indicação de qual pacote a classe faz parte. O único comando permitido antes da instrução de pacote é um comentário. Em relação à declaração do pacote, utilizamos a palavra-chave `package` seguida do nome do pacote. No caso do exemplo, a classe `Retangulo` faz parte do sub pacote `retos` do pacote `geometria`, por isso o nome do pacote é `geometria.retos`. Se a classe fizesse parte apenas do pacote `geometria`, a primeira linha do código seria `package geometria`. De forma análoga, se a classe fizesse parte do pacote `geometria.retos.quatrolados`, a primeira instrução deveria informar isso.

## 11.2 UTILIZAÇÃO DE CLASSE DE PACOTES

Conforme citado, classes que pertencem a um pacote tem seu nome prefixado pelo nome do pacote. Utilizando-se a classe `Retangulo` do exemplo anterior, o nome dessa classe não é simplesmente `Retangulo`, é `geometria.retos.Retangulo`. Uma classe nomeada simplesmente como `Retangulo` não pertence a pacote algum.

Classes que fazem parte de um mesmo pacote **não** precisam utilizar seu nome completo qualificado quando se referenciarem, visto que fazem parte do mesmo pacote. Se uma classe do pacote `geometria.retos` instanciar a classe `Retangulo` (também desse pacote), ela pode fazê-lo através de `Retangulo r = new Retangulo()`. Se uma classe de **outro** pacote instanciar a classe `Retangulo`, porém, ela não pode simplesmente utilizar o nome `Retangulo`, porque ela estaria se referindo à classe `Retangulo` sem pacote, porém o desejado é a classe `Retangulo` do pacote `geometria.retos`. Para poder utilizar classes de outro pacote, uma classe precisa utilizar um dos três métodos abaixo.

### 11.2.1 Usar o nome qualificado completo da classe

A classe `Retangulo` do exemplo anterior, por exemplo, faz parte do pacote `geometria.retos`, logo seu nome qualificado completo é `geometria.retos.Retangulo`. Esse é o nome que deve ser utilizado para se instanciar a classe, conforme o exemplo abaixo.

```
public class Main {
    public static void main(String args[]) {
        geometria.retos.Retangulo r = new geometria.retos.Retangulo();

        r.metodo1();
        r.metodo2();
    }
}
```

Essa forma é rápida e eficaz, porém se for utilizada algumas vezes no mesmo código tornará o programa difícil de ser lido e entendido. Para contornar esse problema, pode-se importar uma classe específica de um pacote, assim seu nome completo não é necessário.

### 11.2.2 Importar uma classe específica de um pacote

Ao importar uma classe específica de um pacote, pode-se referenciar a classe apenas pelo seu primeiro nome ao invés do nome qualificado completo. A importação de uma classe é feita através do comando `import`, como no exemplo abaixo.

```
import geometria.retos.Retangulo;

public class Main {
    public static void main(String args[]) {
        Retangulo r = new Retangulo();
        r.metodo1();
        r.metodo2();
    }
}
```

O comando `import` indica qual a classe a ser importada. É importante notar que o comando `import` exige que o nome qualificado completo da classe seja fornecido. Depois que a importação é feita, a classe pode ser utilizada pelo seu primeiro nome, sem necessidade de referenciar seu pacote.

Quando faz-se necessário importar mais de uma classe de um determinado pacote, o código tente a ter diversos `imports` (um para cada classe a ser importada). Para evitar esse problema, pode-se importar todas as classes de um determinado pacote.

### 11.2.3 Importar um pacote inteiro

Para realizar a importação de todas as classes de um pacote, utiliza-se o comando `import`, porém informando-se um asterisco (\*) no nome da classe (o que indica algo como "todas as classes"). O código abaixo exemplifica a importação de todas as classes do pacote `geometria.retos`.

```
import geometria.retos.*;

public class Main {
    public static void main(String args[]) {
        Retangulo r = new Retangulo();
        Triangulo t = new Triangulo();
        Quadrado q = new Quadrado();

        r.metodo1();
        r.metodo2();
    }
}
```

Ao importar todas as classes de um pacote, pode-se referenciar qualquer uma delas pelo seu primeiro nome, como se cada uma dessas classes tivesse sido importada através de um `import`.

Uma característica muito importante é que o operador asterisco não importa classes pertencentes a subpacotes, ou seja, o comando `import geometria.*` **NÃO** importa as classes do pacote `retos` e `curvos`. Embora o pacote `retos`, por exemplo, esteja dentro do pacote `geometria`, eles não possuem relação em termos de código. O pacote `retos` está dentro do pacote `geometria` apenas para *demonstrar* uma hierarquia e ajudar a fazer a organização das classes mais explícita.

O comando `import geometria.*` faz a importação de todas as classes que estão no pacote `geometria`. O pacote `geometria.retos` é um pacote diferente, assim como `geometria.curvos`. Para importar todas as classes do pacote `geometria`, o programador deveria fazer o seguinte (assumindo que não existem classes dentro do pacote `geometria` e que somente `retos` e `curvos` são os subpacotes de `geometria`):

```
import geometria.retos.*;
import geometria.curvos.*;

public class Main {
    public static void main(String args[]) {
        // código...
    }
}
```

## 11.3 ORGANIZAÇÃO FÍSICA DOS ARQUIVOS

Em termos de organização física de arquivos, um pacote é descrito como uma hierarquia de diretórios. Dessa forma, as classes do pacote `geometria.retos` estariam dentro da pasta `retos`, que estaria dentro da pasta `geometria`.

Ex.:    <caminho do projeto>\geometria\retos\ no Windows, e  
       <caminho do projeto>/geometria/retos no Linux.

Imaginando-se as classes `geometria.retos.Retangulo`, `geometria.retos.Quadrado`, `geometria.curvos.Circulo` e `geometria.curvos.Ellipse`, a seguinte estrutura seria encontrada no disco (imaginando o sistema Windows):

```
<caminho do projeto>\geometria\retos\Retangulo.java
<caminho do projeto>\geometria\retos\Quadrado.java
<caminho do projeto>\geometria\curvos\Circulo.java
<caminho do projeto>\geometria\curvos\Ellipse.java
```

O local onde residem os diretórios dos pacotes e suas classes chama-se **classpath** (do inglês "class path", caminho das classes). Em um projeto pequeno ou em um programa de testes, geralmente não se faz uso de pacotes. Com isso, todas as classes são criadas e colocadas na pasta do projeto, como o exemplo abaixo

```
<caminho do projeto>\Main.java
<caminho do projeto>\Classe1.java
<caminho do projeto>\Classe2.java
```

Todas as classes da aplicação estão na pasta do projeto, então o **classpath** é essa própria pasta. Por essa razão, a compilação das classes é feita sem que nenhum diretório seja informado:

```
<caminho do projeto>\javac *.java
```

Isso irá compilar todas as classes. Depois da compilação, o diretório do projeto terá os seguintes arquivos:

```
<caminho do projeto>\Main.java
<caminho do projeto>\Classe1.java
<caminho do projeto>\Classe2.java
<caminho do projeto>\Main.class
<caminho do projeto>\Classe1.class
<caminho do projeto>\Classe2.class
```

sendo que cada arquivo `.class` corresponde a uma classe compilada. São esses arquivos que a JVM procura ao tentar executar um programa. Imaginando-se que o método `main()` está na classe Main, roda-se o programa da seguinte forma:

```
<caminho do projeto>\java Main
```

O comando funciona corretamente, porque a JVM fará uma busca pela classe Main compilada (`Main.class`) no diretório corrente (que é `<caminho do projeto>`), e o arquivo será encontrado. Quando pacotes são utilizados, as classes estão em diretórios (e subdiretórios), então o compilador java precisa ser informado disso. Imaginando-se a seguinte estrutura de pastas e arquivos

```
<caminho do projeto>\Main.java
<caminho do projeto>\geometria\retos\Retangulo.java
<caminho do projeto>\geometria\retos\Quadrado.java
<caminho do projeto>\geometria\curvos\Circulo.java
<caminho do projeto>\geometria\curvos\Ellipse.java
```

nota-se que a classe Main não faz parte de qualquer pacote, porém as demais classes fazem parte de outros pacotes (`geometria.retos` e `geometria.curvos`). Para compilar todas as classes, é necessário informar ao compilador java os arquivos (e pastas) a serem compilados:

```
<caminho>\javac *.java .\geometria\retos\*.java .\geometria\curvos\*.java
```

Esse comando faz a compilação de três grupos de classes: as classes no diretório atual

(\*.java), as classes do pacote `geometria.retos` (na pasta `.\geometria\retos`) e as classes do pacote `geometria.curvos` (na pasta `.\geometria\curvos`). Os arquivos em disco depois da compilação seriam os seguintes:

```
<caminho do projeto>\Main.java
<caminho do projeto>\Main.class
<caminho do projeto>\geometria\retos\Retangulo.java
<caminho do projeto>\geometria\retos\Retangulo.class
<caminho do projeto>\geometria\retos\Quadrado.java
<caminho do projeto>\geometria\retos\Quadrado.class
<caminho do projeto>\geometria\curvos\Circulo.java
<caminho do projeto>\geometria\curvos\Circulo.class
<caminho do projeto>\geometria\curvos\Ellipse.java
<caminho do projeto>\geometria\curvos\Ellipse.class
```

É importante notar que os arquivos `.class` de cada uma das classes do pacote foram colocados no mesmo diretório que seu equivalente `.java`. Assim como os arquivos `.java`, os arquivos `.class` de uma classe pertencente a um pacote precisam estar em uma hierarquia de diretórios para poderem ser utilizados e encontrados pela JVM. A única informação que precisa ser passada para a JVM é o local onde está a pasta raiz de todos os pacotes, ou seja, a pasta onde estão as classes e outras pastas dos pacotes (a pasta `geometria`, no caso desse exemplo). No caso do exemplo acima, essa pasta raiz é `<caminho do projeto>`, porque as pastas dos pacotes estão contidas nessa pasta (assim como a classe `Main`).

Imaginando-se o código da classe `Main` como o seguinte

```
import geometria.retos.*;
import geometria.curvos.*;

public class Main {
    public static void main(String args[]) {
        Retangulo t = new Retangulo();
        Circulo c = new Circulo();
    }
}
```

a execução do programa seria também dada pelo seguinte comando

```
<caminho do projeto>\java Main
```

porque a JVM irá procurar na pasta `<caminho do projeto>` pelas classes, e ela irá encontrar todas elas (a classe `geometria.retos.Retangulo`, por exemplo, será procurada em `<caminho do projeto>\geometria\retos\Retangulo.class`, e será encontrada).

O procedimento seria diferente se todas as classes do pacote fossem colocadas em outro local, como na pasta "classes". O exemplo abaixo mostra as classes do pacote colocadas em outra pasta, porém a classe `Main` continua na raiz do projeto:

```
<caminho do projeto>\Main.java
<caminho do projeto>\classes\geometria\retos\Retangulo.java
<caminho do projeto>\classes\geometria\retos\Quadrado.java
<caminho do projeto>\classes\geometria\curvos\Circulo.java
<caminho do projeto>\classes\geometria\curvos\Ellipse.java
```

A compilação seguiria a mesma ideia de antes, com a diferença que as classes do pacote estariam em outra pasta:

```
<caminho>\javac *.java .\classes\geometria\retos\*.java .\classes\geometria\curvos\*.java
```

Essa compilação produziria os seguintes arquivos no disco:

```
<caminho do projeto>\Main.java
<caminho do projeto>\Main.class
<caminho do projeto>\classes\geometria\retos\Retangulo.java
<caminho do projeto>\classes\geometria\retos\Retangulo.class
<caminho do projeto>\classes\geometria\retos\Quadrado.java
<caminho do projeto>\classes\geometria\retos\Quadrado.class
<caminho do projeto>\classes\geometria\curvos\Circulo.java
<caminho do projeto>\classes\geometria\curvos\Circulo.class
<caminho do projeto>\classes\geometria\curvos\Ellipse.java
<caminho do projeto>\classes\geometria\curvos\Ellipse.class
```

A execução do programa, porém, não pode ser feita da mesma forma que antes, porque as classes serão procuradas na pasta atual; a classe Main será encontrada, entretanto as classes dos pacotes não serão encontradas. Elas não serão encontradas porque a pasta raiz de todos os pacotes (pasta geometria) está dentro da pasta classes. Para que o programa funcione corretamente, é necessário informar a JVM que existem **dois classpaths** diferentes: um deles é o diretório atual e o outro é a pasta classes (onde estão as classes dos pacotes).

Para informar classpaths para a JVM, utilizamos o parâmetro -cp, como no exemplo abaixo

```
<caminho do projeto>\java -cp <classpath> Main
```

sendo que **<classpath>** deve ser substituído pelos caminhos que devem ser usados como classpath. No caso do exemplo, esses caminhos são o diretório atual (representado por .) e o diretório classes (representado por ./classes). Para separar os diferentes classpaths, utiliza-se o separador ; (ponto e vírgula) no Windows e : (dois pontos) no Linux. Para rodar o exemplo acima, tem-se o seguinte (imaginando o sistema Windows):

```
<caminho do projeto>\java -cp .;./classes Main
```

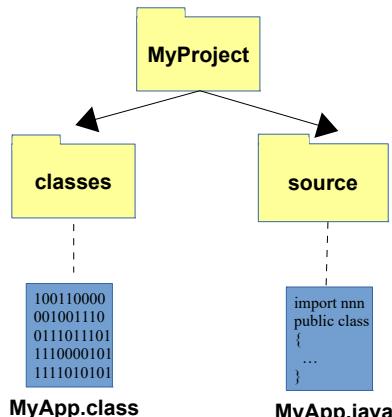
O parâmetro **-cp** (ou **-classpath**) deve informar todos os diretórios que devem ser buscados para que as classes compiladas sejam encontradas.

## 11.4 COMPILANDO COM A FLAG -D

Para facilitar o trabalho de organização de nossos arquivos, além de organizar as classes em pacotes, podemos separar os arquivos-fonte dos arquivos compilados usando a flag **-d** do compilador javac.

```
$ javac arquivo.java -d caminho
```

Vamos separar as classes de nosso projeto em duas pastas: source, para arquivos-fonte e classes para classes compiladas. Teremos a seguinte estrutura:



Coloque todos os seus arquivos-fonte dentro da pasta source. No terminal, entre nesta pasta e compile-os, direcionando o resultado para a pasta classes. Note que para chegarmos a classes a partir de source, temos que voltar um nível na estrutura de pastas ( .. significa "a pasta acima"). Pode-se também compilar todos os arquivos em um mesmo comando usando \*.java.

```
$ cd MyProject/source  
$ javac *.java -d ../classes
```

O aplicativo compilado estará na pasta classes:

```
$ cd MyProject/classes  
$ java MyApp
```

Se o código-fonte estiver organizado em pacotes (packages), a estrutura de pastas referente aos pacotes será criada automaticamente na pasta de destino.

## 11.5 ARQUIVOS JAR

Após concluir o desenvolvimento de um aplicativo, temos 3 opções de implantação / distribuição:

- **Local:** o aplicativo inteiro será executado no computador do usuário final, como um programa autônomo, implantado como um arquivo jar executável;
- **Combinação de local e remota:** o aplicativo é distribuído com uma parte cliente sendo executada no sistema local do usuário, conectada a um servidor onde outras partes do aplicativo são executadas. Isso pode ser implementado através das tecnologias Java Web Start ou por um aplicativo RMI (Remote Method Invocation);
- **Remota:** o aplicativo Java inteiro é executado em um sistema servidor, com o cliente acessando o sistema através de algum meio não relacionado à Java, provavelmente um navegador Web. Temos como exemplo o uso de Servlets.

Examinaremos a seguir a primeira opção: arquivos JAR.

Um arquivo **JAR** é um Java **ARchive**. Ele se baseia no formato de arquivo pkzip e permitirá que possamos empacotar todas as classes pertencentes a um aplicativo em um arquivo único e executável pela JVM.

Por executável, queremos dizer que o usuário final não precisará extrair os arquivos das classes antes de executar o programa. Ele poderá executar o aplicativo com os arquivos de classes ainda no formato JAR. Para isso, é necessário incluir no arquivo JAR um arquivo de declaração (manifesto) que informe à JVM qual das classes tem o método main().

Para criar um arquivo JAR, proceda da seguinte forma:

1. Crie um arquivo chamado **manifest.txt** com o seguinte conteúdo:

```
Main-Class: MyApp
```

2. Pressione ENTER depois de digitar a linha acima, ou seu arquivo pode não funcionar corretamente. Salve-o na pasta onde estão as classes do seu aplicativo.
3. Execute a ferramenta jar para criar um arquivo JAR que contenha suas classes, mais o arquivo manifest.txt:

```
$ jar -cmvf manifest.txt MyApp.jar *.class
```

Se as classes de seu aplicativo estão organizadas em pacotes, inclua também os nomes dos mesmos:

```
$ jar -cmvf manifest.txt MyApp.jar *.class pack1 pack2
```

4. Para executar um arquivo JAR:

```
$ java -jar MyApp.jar
```

## 11.6 RESUMO DO CAPÍTULO

- Pacotes são utilizados para organizar as classes, seja por assunto, por hierarquia, por afinidade, etc.
- Para que uma classe faça parte de um pacote, é necessário o comando `package` no começo do arquivo dessa classe.
- Uma classe que faz parte de um pacote possui um nome qualificado completo, que é a concatenação do nome do pacote mais o nome da classe. Ex.: `geometria.retos.Retangulo`.
- Para utilizar uma classe pertencente a um pacote, deve-se utilizar o comando `import`. Ex.: `import geometria.retos.Retangulo` OU `import geometria.retos.*`.
- O comando `import` não é hierárquico, ou seja, `import geometria.*` não irá importar as classes do pacote `geometria.retos` e `geoemtria.cursos`, por exemplo.
- Ao utilizar pacotes, é importante cuidar os classpaths (caminhos das classes). Se o parâmetro `-cp` não for informado, a JVM não encontrará a classe pertencente ao pacote.
- Se o método `main()` do programa estiver em uma classe que pertence a um pacote, o comando para iniciar o programa deve ter o parâmetro `-cp` e, também, deve usar o nome qualificado completo da classe. Ex.: `java -cp ..\classes nome.pacote.Classe`.
- Para evitar conflitos de nomes, utiliza-se a convenção de nomes inversos de domínios para pacotes. Ex.: `br.edu.uffs.pacote`, `br.com.uol.pacote`, `com.ibm.pacote`.

## 11.7 BIBLIOGRAFIA DO CAPÍTULO

ORACLE. **Lesson:** packages (The Java Tutorials). Disponível em <<http://download.oracle.com/javase/tutorial/java/package/index.html>>. Acesso em 18/05/2011.

## 12 Interface gráfica com o usuário

Uma interface gráfica com o usuário (GUI - *Graphic User Interface*) apresenta um conjunto de componentes visuais (*widgets*) para um programa. Diferentemente das interfaces em modo texto, as GUIs possibilitam ao usuário a interação de forma amigável e intuitiva, evitando necessidade de memorizar longas sequências de teclas e comandos.

*Widget* é um termo sem tradução que designa qualquer componente de uma GUI, como por exemplo: janelas, botões, menus, itens de menu, ícones, caixas de texto, barras de rolagem, etc. Além dos *widgets*, temos também os eventos, que são situações disparadas pelo usuário, como o pressionamento do botão do mouse, de uma tecla, a mudança e estado em um campo, etc. Ao criar elementos gráficos de interface e vinculá-los a eventos, temos a possibilidade de disparar ações específicas do programa.

### 12.1 SWING

A linguagem Java possui 2 pacotes GUI: a original AWT (*Abstract Window Toolkit*) e a nova SWING. A AWT utiliza rotinas nativas do sistema operacional para a construção de janelas, o que contraria a ideia de um modelo virtual. Já o SWING veio como uma extensão à AWT e contém apenas códigos Java. Entretanto, apesar da criação da Swing, a AWT ainda é suportada.

As classes que são utilizadas para criar os componentes GUI do Swing fazem parte do pacote `javax.swing`. Já os componentes AWT estão no pacote `java.awt`. Tratadores de evento estão no pacote `java.awt.event`. Portanto, ao utilizar qualquer componente, devemos primeiramente importar o referido pacote.

### 12.2 COMPONENTES DO SWING

Janelas, campos de texto, botões, listas roláveis, botões de rádio, etc., são todos considerados componentes. Todos eles estendem `javax.swing.JComponent`.

Quase todas as interfaces gráficas são compostas de uma janela principal ou de alto-nível, onde são montados os outros componentes. Estes componentes que agrupam os demais são chamados containers. Podemos ver os containers como o cimento das aplicações em Swing; eles agrupam os componentes que seriam os tijolos (como botões, campos de texto, checkboxes, etc). Por definição, Containers são componentes que podem conter outros componentes.

Há inúmeros componentes Swing, veremos aqui alguns dos principais:

- Componentes container:
  - `JFrame`
  - `JPanel`
  - `JOptionPane`
- Outros componentes:
  - `JLabel`
  - `JButton`
  - `JCheckBox`
  - `JRadioButton`
  - `JTextField`
  - `JTextArea`
  - `JList`
  - `JComboBox`

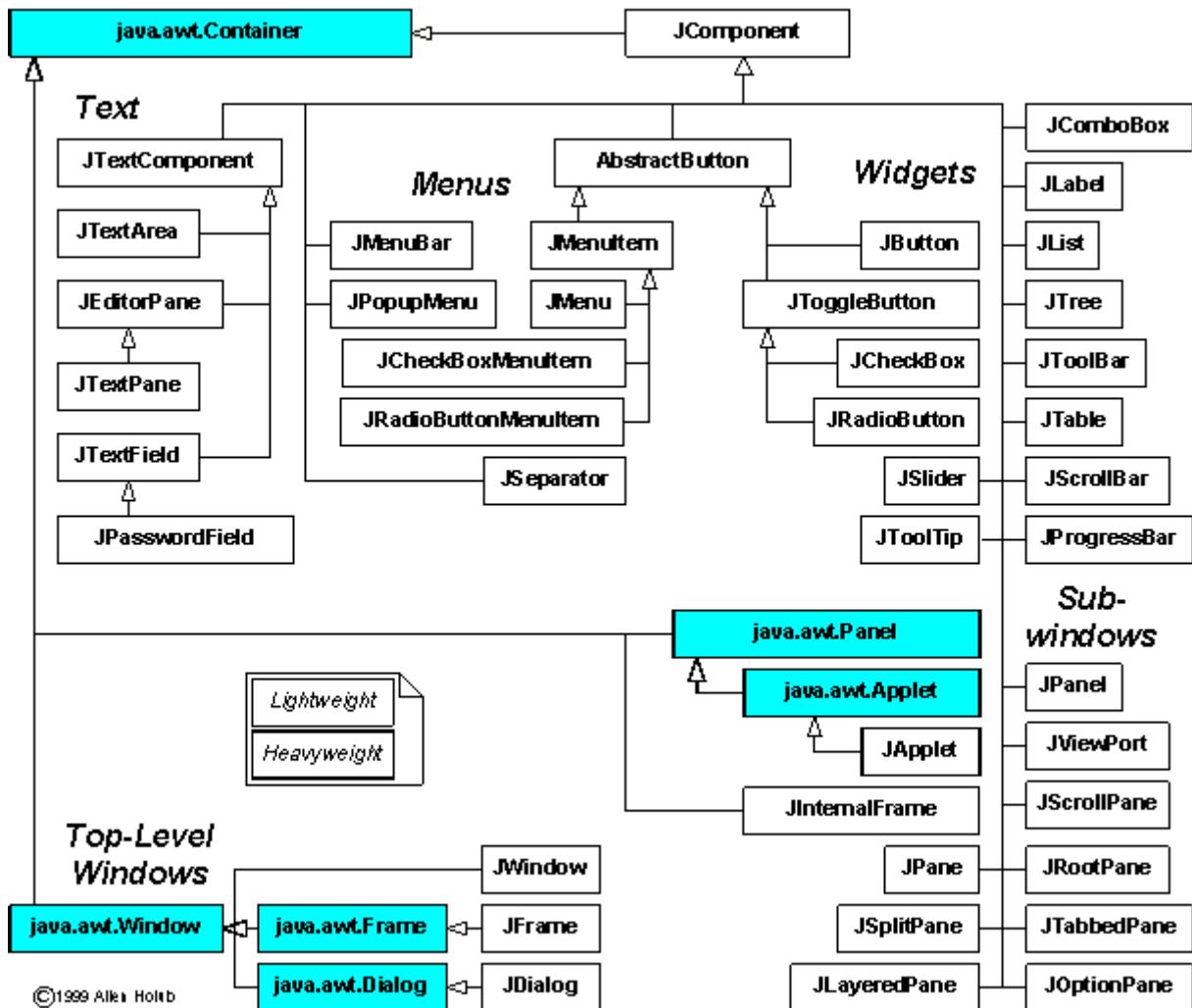


Figura 24: Hierarquia de componentes Swing (fonte: <http://www.holub.com/goodies/java.swing.html>).

### 12.2.1 JFrame

Na Swing, as janelas principais de uma aplicação são geralmente instâncias das classes `JFrame` ou `JWindow`. A diferença entre essas duas classes está na simplicidade: `JWindow` não tem a barra de título e não coloca botões na barra de tarefas, portanto, utilizaremos a `JFrame`.

Um componente `JFrame` será uma janela com as funcionalidades básicas de maximizar, minimizar e fechar, além da barra de título e bordas. Alguns métodos relevantes são `getTitle()`, `setBounds(x,y,w,h)`,  `setLocation(x,y)`,  `setMaximumSize(w,h)`,  `setMinimumSize(w,h)`,  `setPreferredSize(w,h)`,  `setResizable(bool)`,  `setSize(w,h)`,  `setTitle(str)`,  `setVisible(bool)` e  `setDefaultCloseOperation(constant)`, método que controla a ação do ícone fechar (esta constante normalmente é `EXIT_ON_CLOSE`).

Sendo um top-level container, o `JFrame` possui um `contentPane` onde os componentes são adicionados. Uma referência a ele pode ser obtida utilizando-se o método `getContentPane()`.

Veja a janela abaixo. Temos 2 formas de criá-la:



**1<sup>a</sup> opção:** Utilizando um objeto separado da classe da aplicação para a Janela (JFrame).

```
import javax.swing.*;
public class JFrameTest {
    private JFrame f;

    public JFrameTest() {
        f = new JFrame("Exemplo de JFrame");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300,300);
        f.setVisible(true);
    }

    public static void main(String args[]){
        new JFrameTest();
    }
}
```

**2<sup>a</sup> opção:** Utilizando um objeto único para a aplicação e para a Janela (JFrameTest é um JFrame). Desta forma, JFrameTest herdará os atributos e métodos da classe JFrame.

```
import javax.swing.*;
public class JFrameTest extends JFrame {
    public JFrameTest() {
        super("Exemplo de JFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300,300);
        setVisible(true);
    }

    public static void main(String args[]){
        new JFrameTest();
    }
}
```

### 12.2.2 JPanel

O JPanel é um container intermediário que deve sempre ser adicionado a um container do tipo JFrame ou JWindow.

Seu método `add(widgetName)` permite adicionar outros componentes ao painel. A forma como os elementos serão dispostos é controlada por um gerenciador de layout (conforme veremos adiante). O layout padrão de um painel é o FlowLayout.

Para entender a finalidade de um JPanel, vamos primeiramente criar um botão e adicioná-lo diretamente a uma moldura (JFrame), sem o uso de um JPanel.

```
import javax.swing.*;
public class JFrameTest2 {
    public JFrame f;
    public JButton botao;
    public JFrameTest2() {
        f = new JFrame("Exemplo 2: frame com botao ");
        botao = new JButton("pressione");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300,300); // tamanho do frame
        f.add(botao); // adiciona o botão ao frame
        f.setVisible(true);
    }
    public static void main(String args[]){
        new JFrameTest2();
    }
}
```

Teremos o seguinte resultado:



Ou seja, o botão ocupou toda a área disponível da janela. Para evitar este problema, devemos criar um painel (JPanel), adicionar os componentes desejados ao painel e só então adicionar o painel à moldura (JFrame).

Veja o exemplo a seguir:

```
import javax.swing.*;
```

```
public class JFrameTest3 {  
    public JFrame f;  
    public JButton botao;  
    public JPanel p;  
  
    public JFrameTest3(){  
        f = new JFrame("Exemplo 3: frame com painel");  
        botao = new JButton("pressione");  
        p = new JPanel();  
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        f.setSize(300,300); // tamanho do frame  
        p.add(botao); // adiciona botao ao painel  
        f.add(p); // adiciona o painel com botão ao frame  
        f.setVisible(true);  
    }  
    public static void main(String args[]){  
        new JFrameTest3();  
    }  
}
```

Veja o programa em execução:



Os componentes em um JPanel não são reajustados, ou seja, podemos aumentar a janela e o tamanho dos componentes permanecerá o mesmo, o que observamos que não acontece se adicionarmos direto ao JFrame, como no primeiro exemplo desta seção.

Observe que em nenhum momento foi definido o tamanho do botão. Então, por que ele se comporta de forma diferente quando é adicionado ao JFrame e ao JPanel? A resposta para esta pergunta será vista adiante, e diz respeito aos **Gerenciadores de Layout**.

### 12.2.3 JOptionPane

Caixas de diálogo são pequenas janelas popup contendo mensagens curtas, solicitações de confirmação ou campos para a entrada de dados. Elas podem ser modais (requerem uma resposta do usuário antes de continuar) ou não modais (é possível continuar a usar o programa mesmo com a janela aberta).

O componente *JOptionPane* possibilita a criação de caixas de diálogo modais contendo métodos predefinidos para cada tipo de diálogo. Qualquer *JOptionPane* possui um primeiro parâmetro que aponta para sua janela pai (a janela onde deve aparecer) ou null, que aponta para a janela atual. O segundo parâmetro é a mensagem a ser exibida.

Há 3 tipos de caixas de diálogo do tipo *JOptionPane*:

- *showMessageDialog()*

O método *showMessageDialog()* possui mais dois parâmetros opcionais que permitem definir o título da caixa de diálogo e o tipo de ícone a ser exibido. A janela terá apenas um botão OK e não retornará nada.

```
JOptionPane.showMessageDialog(null, "Apenas uma mensagem",
                           "Caixa de mensagem", JOptionPane.WARNING_MESSAGE);
```

- *showConfirmDialog()*

O método *showConfirmDialog()* possui mais três parâmetros adicionais para definir o título da janela, alterar os botões visíveis e selecionar o tipo de ícone. Por padrão, os botões são 'Yes', 'No' e 'Cancel', mas pode-se configurá-los para *JOptionPane.YES\_NO\_OPTION*, por exemplo. Os valores retornados são 0, 1 ou 2 e correspondem respectivamente às constantes *JOptionPane.YES\_OPTION*, *JOptionPane.NO\_OPTION* ou *JOptionPane.CANCEL\_OPTION*.

```
int opcao = JOptionPane.showConfirmDialog(null, "Deseja prosseguir?");
if (opcao == JOptionPane.YES_OPTION) {
    // ação a ser realizada
}
```

- *showInputDialog()*

O método *showInputDialog()* pode conter dois parâmetros adicionais para definir o título e o ícone da caixa de diálogo. Os botões são o OK e o Cancel. Qualquer informação digitada no campo é retornada como uma string.

```
String dados = JOptionPane.showInputDialog(null, "Qual o seu nome?");
String dados = JOptionPane.showInputDialog(null, "Qual o seu nome?",
                                         "Pergunta", JOptionPane.QUESTION_MESSAGE);
```

Os ícones que podem ser utilizados nas janelas são *ERROR\_MESSAGE*, *INFORMATION\_MESSAGE*, *QUESTION\_MESSAGE*, *WARNING\_MESSAGE* e *PLAIN\_MESSAGE* (sem ícone).

### 12.2.4 JLabel

Os rótulos são utilizados para mostrar textos, prompts ou qualquer outra informação em uma GUI. Os rótulos são criados através do construtor *JLabel()*, com o texto do botão como primeiro parâmetro. Um rotulo exibe uma única linha de texto, mas pode exibir também imagens e conteúdo HTML.

Veja como utilizar um JLabel:

- Criar um label simples:

```
JLabel texto = new JLabel("Este texto é um JLabel");
```

- Criar um label com um ícone:

```
Icon bug = new ImageIcon("bug1.gif");
JLabel label1 = new JLabel("Label com texto e ícone", bug, SwingConstants.LEFT);
```

- Colocar uma dica de tela para o label:

```
label1.setToolTipText("Dica de tela do label 1");
```

- Criar um label com o construtor vazio e depois setar suas características:

```
JLabel label1 = new JLabel();
label1.setText("Label com ícone e texto no botão");
Icon bug = new ImageIcon("bug1.gif");
label1.setIcon(bug);
```

- Modificar a posição horizontal / vertical do texto em relação ao ícone (usando as SwingConstants CENTER, LEFT, RIGHT, TOP, BOTTOM):

```
label1.setHorizontalTextPosition(SwingConstants.CENTER);
label1.setVerticalTextPosition(SwingConstants.BOTTOM);
```

- Criar um label com borda de título:

```
JLabel label1 = new JLabel("Texto de um JLabel");
label1.setBorder(BorderFactory.createTitledBorder("Borda do rótulo"));
```

Veja abaixo um exemplo de um programa contendo uma janela, um painel e 2 labels.

```
import javax.swing.*;
public class ExemploJLabel {

    public ExemploJLabel(){
        JFrame f = new JFrame("Título da Janela");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(300,300);
        JLabel texto = new JLabel("Este texto é um JLabel");
        JLabel texto2 = new JLabel("<html>ESTE TEXTO TAMBÉM É UM<u>JLABEL</u></html>");
        JPanel p = new JPanel();
        p.add(texto);
        p.add(texto2);
        f.add(p);
        f.setVisible(true);
    }
}
```

```

    }
    public static void main(String args[]){
        new ExemploJLabel();
    }
}

```

Resultado:



### 12.2.5 JButton

Um botão é um componente em que o usuário clica para acionar uma ação específica. Pode conter um texto ou um ícone, e suas funcionalidades são controladas pelo ActionListener, como veremos adiante.

Eles podem ser desativados com o método `setEnabled(false)` e testados com o método `isEnabled()`. Um método que pode ser útil é o `setMnemonic(char)`, que permite associar uma tecla de atalho ao botão (ao pressionar ALT + a tecla mnemônica, o botão será ativado).

```

import java.awt.*;
import javax.swing.*;
public class Botoes {
    public Botoes() {
        JFrame f = new JFrame("Exemplo JButton");
        f.setSize(400, 200);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel p = new JPanel();
        JButton b1 = new JButton("Botão normal");
        b1.setMnemonic('B');
        p.add(b1);
        ImageIcon brasil = new ImageIcon("brasil.gif");
        JButton b2 = new JButton(brasil);
        b2.setBackground(Color.yellow); // Define a cor de fundo do botão
        p.add(b2);
        f.add(p);
        f.setVisible(true);
    }
    public static void main(String[] args) {

```

```

        new Botoes();
    }
}

```

### 12.2.6 JTextField e JPasswordField

Os JTextField e JPasswordField são áreas de uma única linha em que o texto pode ser inserido via teclado pelo usuário ou onde texto pode simplesmente ser exibido. Quando o usuário digita os dados e pressiona <ENTER>, um evento de ação ocorre.

Um campo de texto é declarado da seguinte forma:

```
JTextField campo = new JTextField(10);
```

No exemplo acima, o campo terá o tamanho de 10 colunas.

Veja como usar um campo de texto:

- Capturar seu conteúdo:

```
String s = campo.getText();
```

- Inserir conteúdo:

```
campo.setText("Texto");
```

```
campo.setText(""); // limpando o texto
```

- Selecionar o texto:

```
campo.selectAll();
```

- Posicionar o cursor sobre o campo:

```
campo.requestFocus();
```

- Capturar um evento de ação quando o usuário pressionar as teclas apagar ou enter:

```
campo.addActionListener(ouvinte);
```

Vejamos abaixo um exemplo funcional, já utilizando gerenciadores de layout para alinhar os componentes e também um exemplo de RadioButton:

```

import java.awt.*;
import javax.swing.*;

public class Textfield {
    public static void main(String[] args) {
        new Textfield();
    }
    public Textfield() {
        JFrame f = new JFrame("Exemplo JTextField");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(500,120);
        f.setVisible(true);
    }
}
```

```
JPanel p = new JPanel(); //painel principal
p.setLayout(new BorderLayout()); // define o layout do container

FlowLayout layout = new FlowLayout(); //instancia um flowlayout
layout.setAlignment(FlowLayout.LEFT); //define alinhamento à esquerda

JPanel painel1 = new JPanel(); //painel para campo nome
painel1.setLayout(layout); //define o layout do painel1

JPanel painel2 = new JPanel(); //painel para campo senha
painel2.setLayout(layout);

JPanel painel3 = new JPanel(); //painel para campo sexo
painel3.setLayout(layout);

JLabel nome = new JLabel("Nome:"); //cria um label
painel1.add(nome); //adiciona o label ao painel 1
JTextField vnome = new JTextField(30);
painel1.add(vnome);

JLabel senha = new JLabel("Senha:");
painel2.add(senha);

JPasswordField vsenha = new JPasswordField(10);
painel2.add(vsenha);

JLabel sexo = new JLabel("Sexo:");
painel3.add(sexo);

ButtonGroup vsexo = new ButtonGroup();
JRadioButton masc = new JRadioButton("Masculino");
vsexo.add(masc); // adiciona ao grupo
painel3.add(masc); // adiciona ao painel
JRadioButton fem = new JRadioButton("Feminino");
vsexo.add(fem);
painel3.add(fem);

p.add(painel1, BorderLayout.NORTH);
p.add(painel2, BorderLayout.CENTER);
p.add(painel3, BorderLayout.SOUTH);
f.add(p); // adiciona o painel ao frame
}

}
```

Veja o resultado. Observe que criamos 3 painéis e adicionamos respectivamente às regiões norte, centro e sul do JFrame (o gerenciador padrão do JFrame é o BorderLayout). Dentro de cada panel, usamos o FlowLayout alinhado à esquerda.



### 12.2.7 JRadioButton

Os botões do tipo `JRadioButton` possibilitam a seleção de apenas um dentre vários elementos. O estado de um radiobutton pode ser obtido através de seu método `isSelected()`.

Grupos de botões são utilizados para garantir a exclusão mútua de botões no grupo (ou seja, apenas um pode estar selecionado em um dado momento). Use `ButtonGroup()` para construir um grupo de botões e então adicione os botões com o método `add()`.

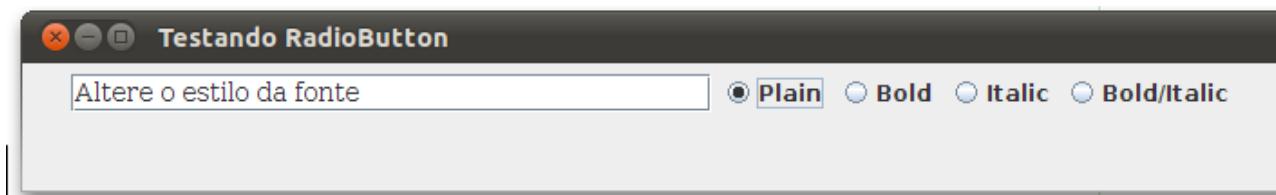
```
...
JRadioButton botaoSomar = new JRadioButton("Somar");
JRadioButton botaoSubtrair = new JRadioButton("Subtrair");
JRadioButton botaoMultiplicar = new JRadioButton("Multiplicar");

ButtonGroup grupo = new ButtonGroup();
grupo.add(botaoSomar);
grupo.add(botaoSubtrair);
grupo.add(botaoMultiplicar);
...
```

É importante destacar que mesmo adicionando os `JRadioButton` a um `ButtonGroup`, não é possível adicionar o grupo diretamente ao painel. Deve-se adicionar ao painel cada `JRadioButton` individualmente. Observe isso no programa do tópico anterior.

Bordas são normalmente adicionadas a grupos de botões para separá-los visualmente. Em Java, para conseguir este efeito você deve criar um painel e então adicionar bordas a ele.

Veja o programa abaixo. O usuário pode alterar o estilo da fonte de um texto de um `JTextField`. O programa utiliza botões de opções que permitem que apenas um único estilo de fonte no grupo seja selecionado de cada vez.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButtonTest implements ItemListener {
```

```
private JFrame j;
private JTextField t;
private JPanel p;
private Font plainFont, boldFont, italicFont, boldItalicFont;
private JRadioButton plain, bold, italic, boldItalic;
private ButtonGroup radioGroup;

public static void main( String args[] )
{
    RadioButtonTest app = new RadioButtonTest();
}

public RadioButtonTest()
{
    j = new JFrame("Testando RadioButton");
    j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    j.setSize(700, 100);
    j.setVisible(true);
    p = new JPanel();
    j.add(p);
    t = new JTextField("Altere o estilo da fonte", 25);
    p.add(t);
    // Cria botões radio
    plain = new JRadioButton("Plain", true);
    p.add(plain);
    bold = new JRadioButton("Bold", false);
    p.add(bold);
    italic = new JRadioButton("Italic", false);
    p.add(italic);
    boldItalic = new JRadioButton("Bold/Italic", false);
    p.add(boldItalic);
    // Cria relacionamentos lógicos entre JRadioButton
    radioGroup = new ButtonGroup();
    radioGroup.add(plain);
    radioGroup.add(bold);
    radioGroup.add(italic);
    radioGroup.add(boldItalic);
    // Cria os listeners para monitorar os eventos
    plain.addItemListener(this);
    bold.addItemListener(this);
    italic.addItemListener(this);
    boldItalic.addItemListener(this);
    //Cria as fontes
    plainFont = new Font("TimesRoman", Font.PLAIN, 14);
    boldFont = new Font("TimesRoman", Font.BOLD, 14);
    italicFont = new Font("TimesRoman", Font.ITALIC, 14);
    boldItalicFont = new Font("TimesRoman", Font.BOLD+Font.ITALIC, 14);
    t.setFont(plainFont);
}

public void itemStateChanged(ItemEvent event)
```

```

    {
        if (event.getSource() == plain)
            t.setFont(plainFont);
        else if (event.getSource() == bold)
            t.setFont(boldFont);
        else if (event.getSource() == italic)
            t.setFont(italicFont);
        else if (event.getSource() == boldItalic)
            t.setFont(boldItalicFont);
    }
}

```

Veja que o exemplo acima já implementa a funcionalidade dos botões através de Listeners, conforme veremos adiante.

### 12.2.8 JCheckBox

Os botões do tipo JCheckBox permitem escolhas do tipo Sim/Não, Verdadeiro/Falso, Ativado/Desativado, etc. Sua principal característica é a possibilidade de selecionar várias opções simultaneamente.

Veja o código abaixo, que apresenta como criamos um JCheckBox.

```
JCheckBox basicoCheckBox = new JCheckBox("Operações Básicas");
JCheckBox cientificoCheckBox = new JCheckBox("Operações Científicas");
JCheckBox financeiroCheckBox = new JCheckBox("Operações Financeiras");
```

No exemplo acima criamos três JCheckBox para o usuário dizer se deseja operações Básicas, Científicas ou Financeiras. Perceba que as opções não são excludentes, ou seja, o usuário pode desejar ver operações Básicas, Científicas e Financeiras ao mesmo tempo.

Para usarmos os CheckBox na tela, basta adicioná-los a um container.

Cada JCheckBox possui dois estados: *checked* e *unchecked*. Para verificar se cada item está *checked* ou *unchecked* basta chamar o método `isSelected()`.

O aplicativo a seguir demonstra o funcionamento do JCheckBox, utilizando dois objetos JCheckBox para alterar o estilo do texto da fonte em um JTextField. Um JCheckBox aplica o estilo negrito se selecionado e o outro aplica o estilo itálico quando selecionado. Se ambos são selecionados o estilo da fonte é negrito e itálico. Inicialmente ambos estão desativados.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CheckBoxTest implements ItemListener {
    private JTextField t;
```

```

private JPanel painel;
private JFrame janela;
private JCheckBox bold, italic;
private int valBold = Font.PLAIN, valItalic = Font.PLAIN;
public static void main(String args[])
{
    CheckBoxTest app = new CheckBoxTest();
}
public CheckBoxTest()
{
    janela = new JFrame("Testando Jcheckbox");
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setSize(300, 100);
    janela.setVisible(true);
    painel = new JPanel();
    janela.add(painel);
    t = new JTextField("Altere o estilo da fonte", 20);
    t.setFont( new Font("TimesRoman", Font.PLAIN, 14) );
    painel.add(t);
    // Cria e adiciona os objetos CheckBox
    bold = new JCheckBox("Bold");
    painel.add(bold);
    italic = new JCheckBox("Italic");
    painel.add(italic);
    //Cria os Listeners para monitorar os eventos
    bold.addItemListener(this);
    italic.addItemListener(this);
}

public void itemStateChanged(ItemEvent event)
{
    if (event.getSource() == bold)
        if (event.getStateChange() == ItemEvent.SELECTED)
            valBold = Font.BOLD;
        else
            valBold = 0;
    if (event.getSource() == italic)
        if (event.getStateChange() == ItemEvent.SELECTED)
            valItalic = Font.ITALIC;
        else
            valItalic = 0;
    t.setFont(new Font("TimesRoman", valBold + valItalic, 14));
}
}

```

### 12.2.9 JTextArea

Diferentemente do JTextField, o JTextArea pode ter mais de uma linha de texto. Como ele não vem com barras de rolagem ou quebras de linha, é preciso um esforço adicional em sua configuração. Para fazer um JTextArea rolar, teremos que incluí-lo em um painel de rolagem

**JScrollPane.**

Veja abaixo como declarar e utilizar o JTextArea.

```
JTextArea texto = new JTextArea (10, 20); // 10 linhas, 20 colunas
```

- Para que haja somente a barra de rolagem vertical:

```
JScrollPane scroller = JScrollPane(texto);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

scroller.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
painel.add(scroller);
IMPORTANTE: devemos adicionar o painel de rolagem ao container, e não o TextArea.
```

- Substituir o texto existente:

```
texto.setText("novo texto");
```

- Acrescentar algo a um texto existente:

```
texto.append("nova linha de texto");
```

- Selecionar o texto:

```
texto.selectAll();
```

- Posicionar o cursor no campo:

```
texto.requestFocus();
```

Veja o programa abaixo. Ele cria um JTextArea usando um painel de rolagem e um botão para adicionar conteúdo.



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TextAreaTest implements ActionListener {
    JTextArea text;
    int count;
    public static void main(String args[])
    {
        TextAreaTest gui = new TextAreaTest();
    }
}
```

```

        gui.go();
    }

    public void go() {
        JFrame janela = new JFrame("Testando JTextArea");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setSize(350, 300);
        janela.setVisible(true);
        JPanel painel = new JPanel();
        JButton botao = new JButton("Clique aqui");
        botao.addActionListener(this);
        text = new JTextArea (10, 20); // 10 linhas, 20 colunas
        text.setLineWrap(true);
        JScrollPane scroller = new JScrollPane(text);
        scroller.setVerticalScrollBarPolicy(
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        scroller.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        painel.add(scroller);
        janela.add(BorderLayout.CENTER, painel);
        janela.add(BorderLayout.SOUTH, botao);
    }
    public void actionPerformed(ActionEvent event)
    {
        this.count++;
        text.append("Botão clicado pela "+count+"ª vez\n");
    }
}

```

### 12.2.10 JComboBox

Uma caixa de combinação ou ComboBox fornece uma lista de itens dentre os quais o usuário pode escolher. As caixas de combinação são implementadas com a classe JComboBox. Os componentes JComboBox geram ItemEvents, da mesma forma que os componentes JCheckBox e JRadioButtons.

```

JComboBox comboOperacao = new JComboBox();
comboOperacao.addItem("Somar");
comboOperacao.addItem("Subtrair");
comboOperacao.addItem("Multiplicar");

```

No código acima criamos um JComboBox e adicionamos três opções a ele. Para colocarmos nosso JComboBox na tela, basta adicioná-lo a um painel ou frame. O código abaixo mostra como descobrimos qual opção o usuário selecionou.

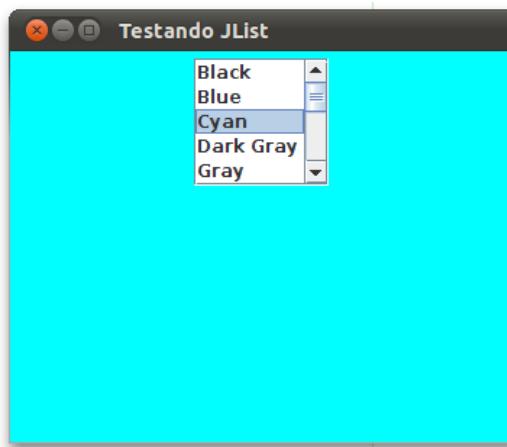
```
String operacaoSelecionada = (String)comboOperacao.getSelectedItem();
```

Na variável operacaoSelecionada estará a String da operação que o usuário selecionou. Nós obtivemos este valor através do método getSelectedItem da classe JComboBox.

### 12.2.11 JList

A classe JList suporta listas de uma única seleção e listas de seleção múltipla. O aplicativo a seguir gera um JList de 13 cores. Quando um nome de cor é selecionado no JList, um ListSelectionEvent ocorre e a cor de fundo do painel de conteúdo da janela altera-se.

Veja o exemplo abaixo:



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class ListTest implements ListSelectionListener{
    private JList colorList;
    private JPanel painel;
    private String colorNames[] = {"Black", "Blue", "Cyan", "Dark Gray", "Gray",
"Green", "Light Gray", "Magenta", "Orange", "Pink", "Red", "White", "Yellow" };
    private Color colors[] = {Color.black, Color.blue, Color.cyan,
Color.darkGray, Color.gray, Color.green, Color.lightGray, Color.magenta, Color.orange,
, Color.pink, Color.red,
    Color.white, Color.yellow };

    public ListTest() {
        JFrame janela = new JFrame("Testando JList");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setSize(350, 300);
        janela.setVisible(true);
        painel = new JPanel();
        janela.add(painel);
        // cria uma lista com os itens de um array de nomes de cores
        colorList = new JList(colorNames);
        colorList.setVisibleRowCount(5);
        // não permite seleções múltiplas
        colorList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        // adicionado um JScrollPane contendo o JList para o painel
        painel.add(new JScrollPane(colorList));
        // estabelece um tratador de eventos
        colorList.addListSelectionListener(this);
    }
}
```

```

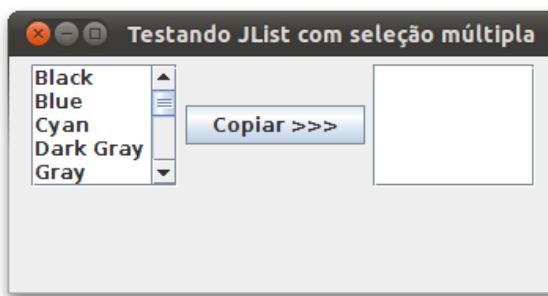
public void valueChanged(ListSelectionEvent event)
{
    painel.setBackground(colors[colorList.getSelectedIndex()]);
}
public static void main( String args[] )
{
    ListTest app = new ListTest();
}
}

```

### 12.2.12 Listas de seleção múltipla

Uma lista de seleção múltipla permite ao usuário selecionar muitos itens de uma JList. Uma lista SINGLE\_INTERVAL\_SELECTION permite seleção de um intervalo contínuo de itens na lista clicando no primeiro item, então mantendo pressionada a tecla Shift enquanto clica no último item para selecionar para selecionar no intervalo. Uma lista MULTIPLE\_INTERVAL\_SELECTION permite seleção contínuo de intervalo como descrito para SINGLE\_INTERVAL\_SELECTION, e permite que itens variados sejam selecionados mantendo pressionada a tecla Ctrl enquanto se clica em cada item para selecionar. Para remover a seleção de um item, mantém-se pressionada a tecla Ctrl ao clicar no item uma segunda vez.

O aplicativo a seguir utiliza listas de seleção múltiplas para copiar itens de um Jlist para outro. Uma lista é uma lista MULTIPLE\_INTERVAL\_SELECTION e a outra é uma lista SINGLE\_INTERVAL\_SELECTION.



```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MultipleSelection implements ActionListener {
    private JList colorList, copyList;
    private JButton copy;
    private String colorNames[] = {"Black", "Blue", "Cyan", "Dark Gray", "Gray",
        "Green", "Light Gray", "Magenta", "Orange", "Pink", "Red", "White", "Yellow" };

    public MultipleSelection()
    {
        JFrame janela = new JFrame("Testando JList com seleção múltipla");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.setSize(350, 300);
        janela.setVisible(true);
        JPanel painel = new JPanel();

```

```
janela.add(painel);
colorList = new JList(colorNames);
colorList.setVisibleRowCount(5);
colorList.setFixedCellHeight(15);
colorList.setSelectionMode(
    ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
painel.add(new JScrollPane(colorList));
// cria um botão de copia
copy = new JButton("Copiar >>>");
painel.add(copy);
copy.addActionListener(this);
copyList = new JList();
copyList.setVisibleRowCount(5);
// conf. largura em pixels de cada item
copyList.setFixedCellWidth(100);
copyList.setFixedCellHeight(15);
// conf. altura em pixels de cada item
copyList.setSelectionMode(
    ListSelectionModel.SINGLE_INTERVAL_SELECTION);
painel.add(new JScrollPane(copyList));
}
public void actionPerformed(ActionEvent event)
{
    copyList.setListData(colorList.getSelectedValues());
}
public static void main( String args[] )
{
    MultipleSelection app = new MultipleSelection();
}
}
```

### 12.3 GERENCIADORES DE LAYOUT

Vimos que um container é um componente que contém outros componentes. Cada container possui um gerenciador de layout, o qual é utilizado para definir como os componentes devem ser distribuídos dentro dele.

Em Java existem diversos gerenciadores de layout: FlowLayout, BorderLayout, BoxLayout e GridLayout.

#### 12.3.1 FlowLayout

O gerenciador FlowLayout é o padrão para o container JPanel. O FlowLayout especifica que os componentes devem ser organizados da esquerda para a direita e de cima para baixo, na ordem em que foram adicionados. Quando um componente não couber horizontalmente, ele será colocado em uma nova linha. Cada componente terá a largura e altura necessárias para comportar seu conteúdo.

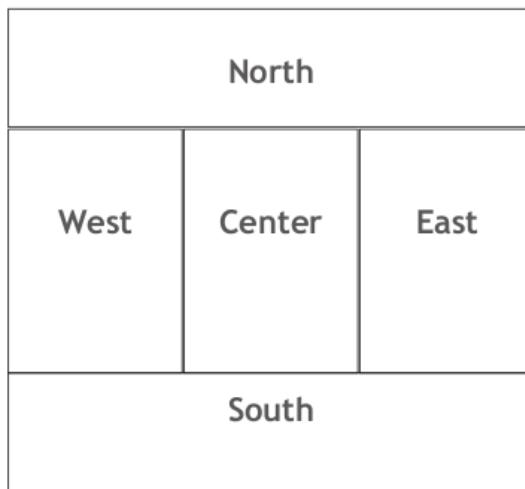
O alinhamento padrão é a esquerda. As constantes que modificam o alinhamento são. FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER

```
JFrame f = new JFrame();
f.setLayout(new FlowLayout());
...
f.add(botao1);
f.add(botao2);
f.add(botao3);
f.add(botao4);
f.add(botao5);
```



### 12.3.2 BorderLayout

O gerenciador BorderLayout divide o componente de plano de fundo em 5 áreas: centro, leste, oeste, norte e sul, conforme a figura abaixo.



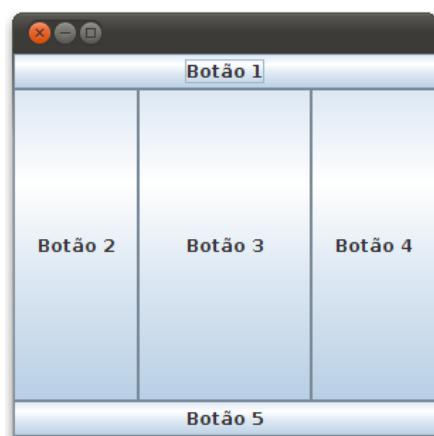
O BorderLayout é o gerenciador padrão do container JFrame. Então quando adicionamos um componente em uma tela, podemos definir em qual região este componente deve ficar, conforme código de exemplo abaixo.

```
tela.add(component, BorderLayout.NORTH);
```

Cada região comporta somente um componente. Portanto, para adicionar vários, é necessário criar um novo painel, adicionar os componentes a ele e depois adicionar o painel à região desejada.

Cada componente adicionado à região norte ou sul terá a largura da janela e a altura preferencial. Já componentes adicionados às regiões leste ou oeste terão sua largura preferencial, mas a altura será esticada para ocupar todo o espaço disponível. A área central fica com o espaço de sobra quando a janela for redimensionada.

```
JFrame f = new JFrame();
f.setLayout(new BorderLayout());
...
f.add(BorderLayout.NORTH, botao1);
f.add(BorderLayout.WEST, botao2);
f.add(BorderLayout.CENTER, botao3);
f.add(BorderLayout.EAST, botao4);
f.add(BorderLayout.SOUTH, botao5);
```



### 12.3.3 BoxLayout

O gerenciador BoxLayout se parece com o FlowLayout: cada componente tem seu próprio tamanho e os mesmos são inseridos no container na ordem em que são adicionados. Entretanto, o BoxLayout pode empilhar os componentes verticalmente (eixo Y) ou horizontalmente (eixo X). Veja o exemplo:

```
JFrame f = new JFrame();
JPanel p = new JPanel();
f.add(p);
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
...
p.add(botao1);
p.add(botao2);
p.add(botao3);
p.add(botao4);
p.add(botao5);
```



### 12.3.4 GridLayout

O gerenciador grid layout irá criar uma grade virtual no container e colocar cada componente dentro de cada célula desta grade. A grade terá um número fixo de linhas e colunas e todos os componentes terão o mesmo tamanho (o tamanho da célula). Veja o exemplo abaixo.

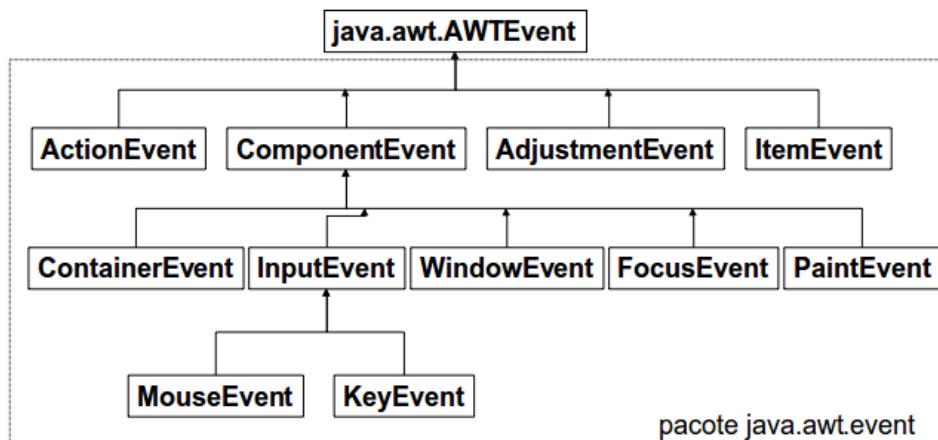
```
p.setLayout(new GridLayout(2, 3));
...
p.add(botao1);
p.add(botao2);
p.add(botao3);
p.add(botao4);
p.add(botao5);
```



### 12.3.5 Modelo de Tratamento de Eventos

As GUIs são baseadas em eventos, isto é, geram eventos quando o usuário do programa interage com a GUI. Algumas interações comuns são mover o mouse, clicar no mouse, clicar em um botão, digitar um campo de texto, etc. Quando ocorre uma interação com o usuário, um evento é automaticamente enviado para o programa.

Informações de evento GUI são armazenados em um objeto de uma classe que estende AWTEvent. Veja abaixo a hierarquia de classes de eventos:

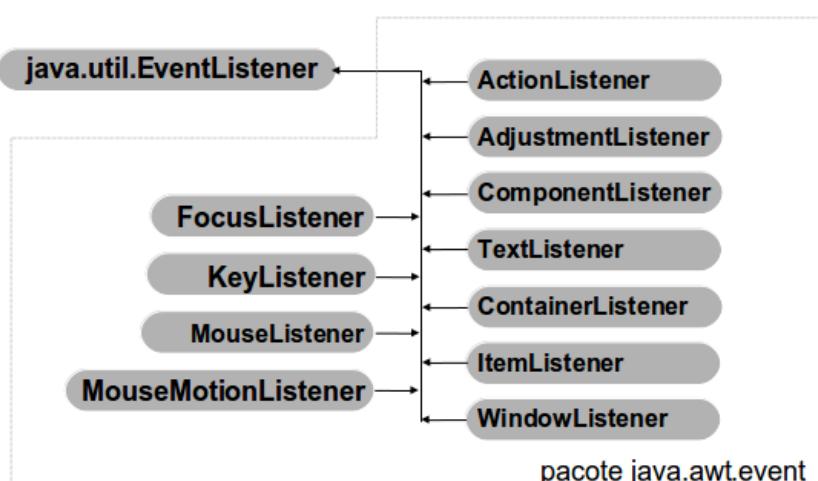


Quando queremos que um componente gráfico realize uma determinada tarefa desenvolvemos um "tratador de eventos" (ou *handler*) para um evento específico do componente.

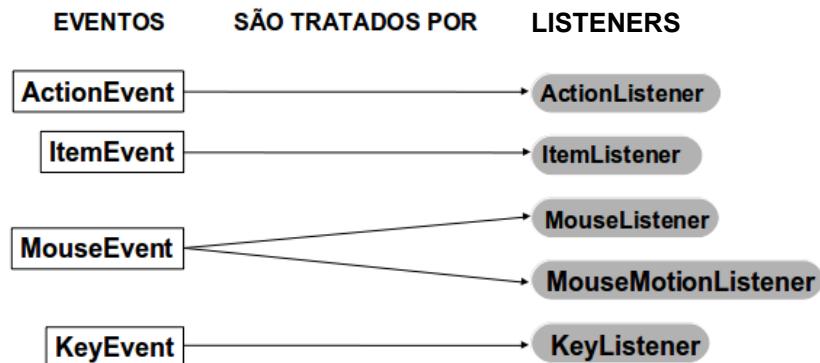
Por exemplo, se o componente gráfico for um botão e desejarmos realizar algo no evento "pressionamento do botão", então desenvolvemos um método tratador para este evento. Neste método dizemos o que deve ser feito quando o botão for pressionado. Neste cenário, o objeto é vinculado ao método handler através de um "event listener" ou "ouvidor de evento", pois ela "ouve" os eventos, ou seja, fica aguardando o acontecimento do evento. O componente que gera o evento (a classe do botão) é chamada de "event source", ou seja, fonte do evento.

Enquanto os eventos são classes, os listeners são interfaces que devem ser implementadas na classe que irá tratar o evento. Criar um observador de eventos para um componente consiste em criar uma classe que implemente a interface observadora do evento a ser tratado.

Veja abaixo alguns *listeners*:



Cada listener destina-se ao tratamento de um tipo de evento:



Por exemplo, o componente botão é implementado na classe JButton, que gera eventos de ação (ActionEvent). Portanto, um event listener para as ações do botão deve implementar a interface ActionListener. Isto significa que ele DEVE implementar todos os métodos desta interface. A interface ActionListener tem o método actionPerformed. Desta forma, nossa classe que irá tratar o evento deverá implementar a interface ActionListener e especificar o método actionPerformed.

Cada uma das interfaces observadoras define um ou mais métodos tratadores de eventos, como veremos a seguir.

### 12.3.6 Interface ActionListener

- Eventos capturados:
  - `ActionEvent`
- Método(s) definidos:
  - `void actionPerformed(ActionEvent e)`
- Componentes que registram:
  - `JComboBox, JButton, JMenuItem, JTextField, JPasswordField`
- Método usado para registrar:
  - `addActionListener(ActionListener al)`

### 12.3.7 Interface ItemListener

- Eventos capturados:
  - `ItemEvent`
- Método(s) definidos:
  - `void itemStateChanged(ItemEvent e)`
- Componentes que registram:
  - `JCheckBox, JRadioButton, JComboBox`
- Método usado para registrar:
  - `addItemListener(ItemListener il)`

### 12.3.8 Interface ListSelectionListener

- Eventos capturados:
  - `ListSelectionEvent`
- Método(s) definidos:

- `void valueChanged(ListSelectionEvent e)`
- Componentes que registram:
  - `JList`
- Método usado para registrar:
  - `addListSelectionListener(ListSelectionListener lsl)`

### 12.3.9 Interface MouseListener

- Eventos capturados:
  - `MouseEvent`
- Método(s) definidos:
  - `void mousePressed(MouseEvent e)`
  - `void mouseClicked(MouseEvent e)`
  - `void mouseReleased(MouseEvent e)`
  - `void mouseEntered(MouseEvent e)`
  - `void mouseExited(MouseEvent e)`
- Componentes que registram:
  - Vários
- Método usado para registrar:
  - `addMouseListener(MouseListener ml)`

### 12.3.10 Interface MouseMotionListener

- Eventos capturados:
  - `MouseEvent`
- Método(s) definidos:
  - `void mouseDragged(MouseEvent e)`
  - `void mouseMoved(MouseEvent e)`
- Componentes que registram:
  - Vários
- Método usado para registrar:
  - `addMouseMotionListener(MouseMotionListener mml)`

### 12.3.11 Interface KeyListener

- Eventos capturados:
  - `KeyEvent`
- Método(s) definidos:
  - `void keyPressed(KeyEvent e)`
  - `void keyReleased(KeyEvent e)`
  - `void keyTyped(KeyEvent e)`
- Componentes que registram:
  - Vários
- Método usado para registrar:

- `addKeyListener(KeyListener mml)`
- As teclas são divididas em 3 categorias:
  - Teclas de ação
    - Setas, Home, End, PageUp, PageDn, teclas de funções, Num Lock, Print Screen, Scroll Lock, Caps Lock e Pause.
  - Teclas comuns
    - Letras, Números, Símbolos, Espaço, Enter, Shift, Alt, Ctrl
  - Modificadoras
    - Shift, Alt e Ctrl
- `keyTyped()` só é chamado para as teclas comuns
- `keyPressed()` e `keyReleased()` são chamados para todos os tipos de teclas.

#### 12.3.12 Classe MouseEvent

- Alguns métodos:
  - `int getX(): Devolve a coordenada x onde o MouseEvent ocorreu.`
  - `int getY(): Devolve a coordenada y onde o MouseEvent ocorreu.`
  - `int getClickCount(): informa quantas vezes o mouse foi clicado.`
  - `boolean isMetaDown(): Botão da direita do mouse foi usado para clicar.`
  - `boolean isAltDown(): Botão do meio do mouse foi usado para clicar.`

## 12.4 RESUMO DO CAPÍTULO

Passos para criar uma GUI:

- Declarar um componente JFrame (o *top-level container*);
- Criar e adicionar os componentes ao content pane do frame ou a algum outro container intermediário ( JPanel );
- Definir o gerenciador de layout;
- Definir a operação padrão de fechamento;
- Definir o tamanho do frame com setSize().
- Tornar a janela visível com setVisible(true).
- Registrar listeners para os componentes que querem tratar eventos.
- Criar os event handlers.

## 12.5 BIBLIOGRAFIA DO CAPÍTULO

SIERRA, Kathy; BATES, Bert. **Use a Cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2007.

SILVA, Paulo Fernando. **Programação**. Univali. 2007.

PEREIRA, Frederico Costa Guedes. **Java: Swing**. 2000.

## 13 Persistência de objetos

Persistência é a capacidade de um programa guardar dados ou objetos por um tempo maior do que o tempo em que o mesmo permanece em execução. Até o momento, todos os objetos que criamos permanecem na memória RAM e são perdidos quando a aplicação é encerrada. A persistência consiste em salvar dados por tempo indeterminado, de modo que estes possam, em outro momento, ser recuperados.

Exemplos de objetos que poderiam ser persistidos:

- Clientes, produtos, fornecedores e funcionários, em um sistema de loja;
- O estado de um jogo;
- Um *log* de mensagens;
- As preferências do usuário em um aplicativo.

Já vimos que objetos possuem estado e comportamento. O comportamento reside na classe (são os métodos). O estado encontra-se nos atributos ou variáveis de instância. Para guardar o estado de um objeto, podemos utilizar as seguintes técnicas:

- **serialização/desserialização**: consiste em “congelar” o estado dos objetos e depois reconstituí-los;
- **arquivos**: podemos gravar os valores de cada atributo em um arquivo de texto simples (ou de bytes);
- **banco de dados**: podemos utilizar bancos de dados relacionais ou orientados a objetos, mas não abordaremos este assunto aqui.

### 13.1 SERIALIZAÇÃO

Serializar um objeto significa capturar seu estado e transformá-lo em uma cadeia de bytes, os quais podem ser armazenados em um arquivo ou transmitidos em uma rede. O processo inverso, chamado de desserialização, consiste em recuperar o conjunto de bytes recompor o objeto original.

A serialização envolve o estabelecimento de um *stream* (ou fluxo) de cadeia capaz de converter objetos em dados que podem ser gravados ou transmitidos. Mas, para que a gravação ou transmissão se efetive, é necessário o auxílio de um *stream* (ou fluxo) de conexão. Um fluxo de conexão é um objeto de transmissão de dados que conecta uma origem e um destino (pode ser um arquivo ou soquete de rede).

Para gravar e recuperar objetos serializados em arquivos, vamos utilizar como fluxo de conexão as classes:

- **FileOutputStream**: *stream* de arquivo que permite a gravação em disco;
- **FileInputStream**: *stream* que permite a leitura de um arquivo em disco.

Para converter objetos em um formato de dados compatível com o fluxo de conexão descrito acima, vamos utilizar as seguintes classes:

- **ObjectOutputStream**: através do método **writeObject**, cria uma cadeia de bytes a partir de um objeto (serialização) e a insere em um fluxo de conexão;
- **ObjectInputStream**: recebe bytes de um fluxo de conexão e recupera um objeto, por meio do método **readObject** (desserialização).

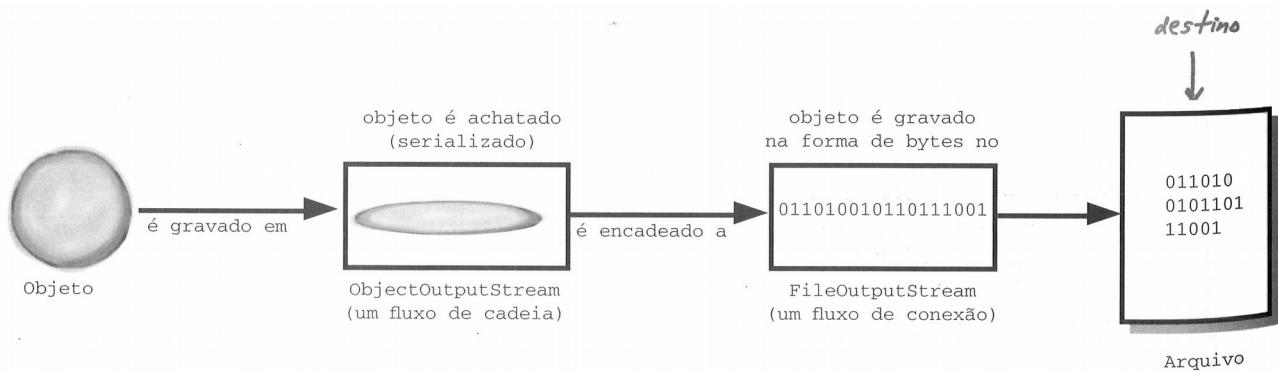


Figura 25: Processo de serialização (SIERRA & BATES, 2007).

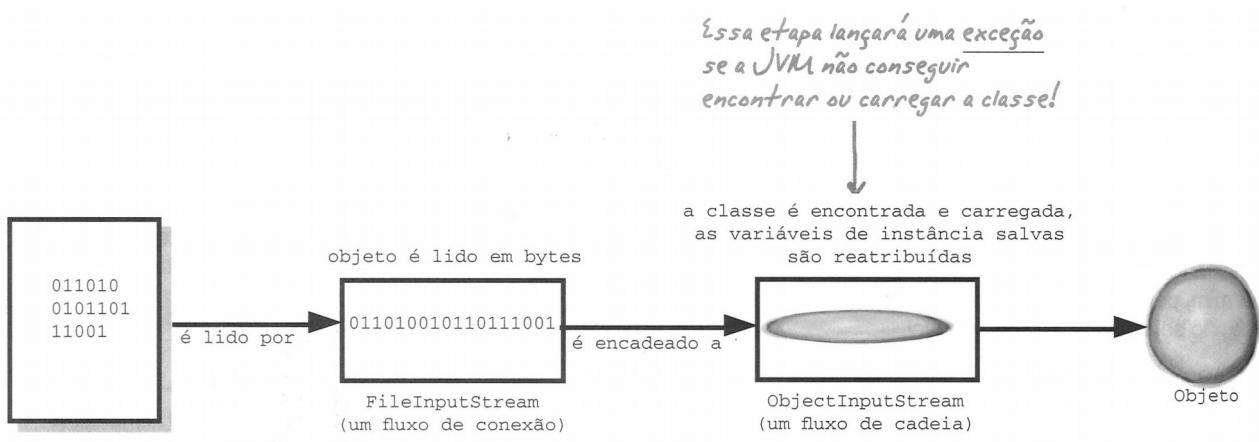


Figura 26: Processo de desserialização (SIERRA & BATES, 2007).

### 13.1.1 Gravando um objeto serializado em disco

Para gravar um objeto serializado em disco, devemos seguir estas etapas:

1. Criar um objeto FileOutputStream:

```
FileOutputStream outFile = new FileOutputStream("file.ser");
```

Se o arquivo **file.ser** não existe, ele será criado.

2. Criar um objeto ObjectOutputStream:

```
ObjectOutputStream os = new ObjectOutputStream(outFile);
```

Estamos encadeando um fluxo a outro.

3. Gravar o (s) objeto(s):

```
os.writeObject(obj1);
os.writeObject(obj2);
```

Serializa o objeto referenciado por obj1 e obj2 e grava no arquivo file.ser. A classe de obj1 e obj2 deve implementar a interface Serializable.

4. Fechar o ObjectOutputStream:

```
os.close();
```

Fechar o fluxo mais abrangente fechará os fluxos de nível inferior, portanto, o objeto FileOutputStream e o arquivo serão fechados automaticamente.

### 13.1.2 Recuperando um objeto serializado

Para recuperar um objeto serializado que se encontra em um arquivo em disco, devemos seguir estas etapas:

1. Criar um objeto FileInputStream:

```
FileInputStream inFile = new FileInputStream("file.ser");
```

Se o arquivo **file.ser** não existe, será lançada uma **FileNotFoundException**.

2. Criar um objeto ObjectOutputStream:

```
ObjectInputStream os = new ObjectInputStream(inFile);
```

Estamos encadeando um fluxo a outro.

3. Ler o(s) objeto(s):

```
TipoObjeto obj1 = (TipoObjeto)os.readObject();
TipoObjeto obj2 = (TipoObjeto)os.readObject();
```

O método `readObject` retorna um Object. Por esta razão, temos que fazer o downcast para o tipo real do objeto.

Cada execução de `readObject` lerá o próximo objeto do fluxo. Se tentarmos ler mais objetos do que existem, teremos uma exceção.

4. Fechar o ObjectOutputStream:

```
os.close();
```

Fechar o fluxo mais abrangente fechará os fluxos de nível inferior, portanto, o objeto `FileInputStream` e o arquivo serão fechados automaticamente.

### 13.1.3 Interface Serializable

Para que um objeto possa ser serializado em Java, sua classe deve ser declarada como serializável. Isso é feito implementando-se a interface **java.io.Serializable**.

Esta interface não possui nenhum método, funcionando apenas como uma marcação para informar à JVM que a referida classe pode ser serializada.

```
import java.io.Serializable;
public class Aluno implements Serializable {
    ...
}
```

É importante destacar que quando uma superclasse implementa a interface “Serializable”, todas as suas subclasses estarão implicitamente marcadas como serializáveis, mesmo que não declarem isso explicitamente.

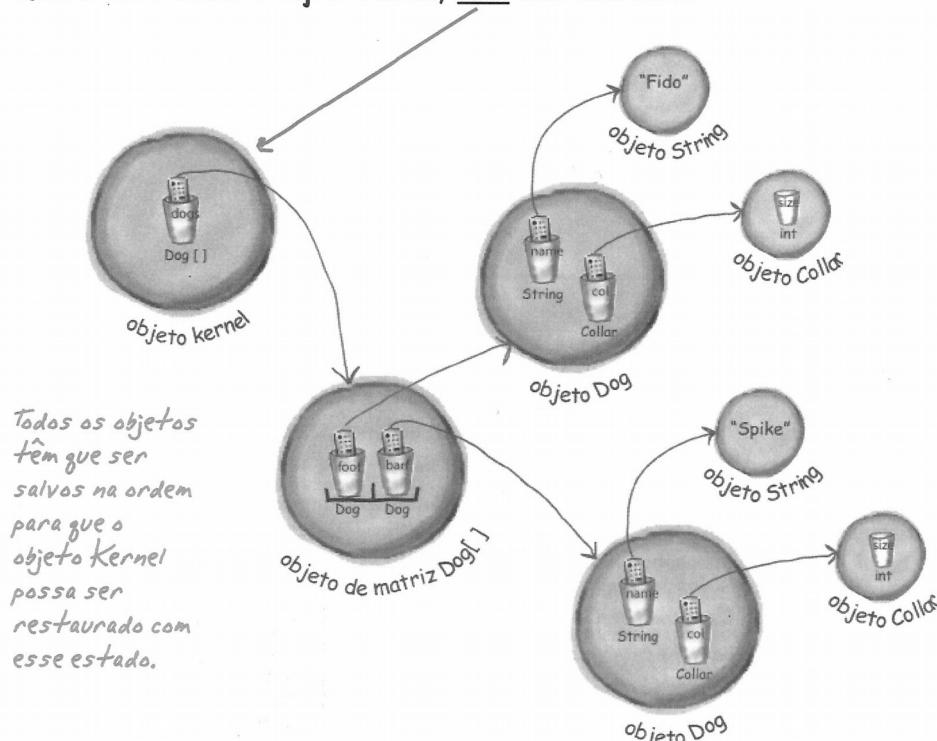
### 13.1.4 Serialização e associações/composições

Quando um objeto é serializado, todos os objetos que ele referencia em suas variáveis de instância também serão serializados. Por sua vez, se estes objetos referenciam outros, estes serão também serializados, e assim sucessivamente.

Observe a fig. 27. Suponha um objeto Kernel que possua uma referência para uma matriz

de Dog (lembre-se, matrizes são também objetos). Esta matriz contém 2 referências a objetos Dog. Cada objeto Dog possui um atributo nome (que é uma referência a um objeto String) e um objeto Collar, o qual possui um int. Ao serializar o objeto Kernel, a matriz, os objetos Dog, os String e os objetos Collar serão serializados automaticamente.

**Quando você salvar o objeto Kernel, tudo isso será salvo**



**A serialização salva a ramificação inteira do objeto. Todos os objetos são referenciados pelas variáveis de instância, a partir do objeto que está sendo serializado.**

Figura 27: Na serialização, objetos envolvidos em relações de composição e associação também serão serializados (SIERRA & BATES, 2007).

No entanto, diferente da herança, no caso de associações as classes relacionadas não serão automaticamente serializáveis. É necessário então que todas elas implementem a interface Serializável, ou teremos o problema reportado na fig. 28.

### 13.1.5 Atributos *transient*

Para evitar que um atributo específico seja serializado, basta marcá-lo como `transient`. Assim, no momento da chamada ao método `writeObject`, o objeto será serializado sem aquele atributo.

```
public class Carro implements Serializable{
    private String modelo;
    private transient String cor;
    private PortaMalas pm;
    ...
}
```

No momento da desserialização, atributos transientes serão restaurados com valores null (quando se tratar de atributo do tipo objeto) ou com valores padrão (quando se tratar de tipo primitivo).

```

import java.io.*;

public class Pond implements Serializable {
    private Duck duck = new Duck(); ← os objetos Pond podem ser serializados
    public static void main (String[] args) {
        Pond myPond = new Pond();
        try {
            FileOutputStream fs = new FileOutputStream("Pond.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);

            os.writeObject(myPond); ← A classe Pond tem uma variável de instância, um objeto Duck.
            os.close();

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

Nossa!! Duck não pode ser serializada! Ela não implementa Serializable, portanto, quando você tentar serializar um objeto Pond, não conseguirá, porque a variável de instância Duck desse objeto não pode ser salva. ← Quando você tentar executar o método main na classe Pond:
public class Duck {
    // o código de duck entra aqui
}

```

Quando você serializar myPond (um objeto Pond), sua variável de instância Duck será serializada automaticamente.

Figura 28: Problemas quando uma variável de instância se refere a uma classe não serializável (SIERRA & BATES, 2007).

### 13.1.6 Variáveis estáticas

Variáveis estáticas não são serializadas, pois pertencem à classe e não às suas instâncias. Ao ser desserializado, o objeto terá o valor das variáveis estáticas de sua classe naquele momento. Portanto, objetos que podem ser serializados não devem depender de valores de variáveis estáticas que sejam alteradas dinamicamente, tendo em vista que seus valores podem não ser os mesmos quando o objeto for reconstituído.

### 13.1.7 Um exemplo de serialização

Abaixo, apresentamos um exemplo onde objetos da classe Carro são serializados. A classe Carro possui um atributo do tipo PortaMalas. Os trechos mais relevantes são destacados em vermelho.

Classe Carro:

```

import java.io.Serializable;

public class Carro implements Serializable{
    private String modelo;
    private String cor;
    private PortaMalas pm;

    public Carro() { }

    public Carro(String modelo, String cor, int litros) {

```

```

        this.modelo = modelo;
        this.cor = cor;
        this.pm = new PortaMalas(litros);
    }

    public String getCor() { return cor; }
    public String getModelo() { return modelo; }
    public void setCor(String cor) { this.cor = cor; }
    public void setModelo(String modelo) { this.modelo = modelo; }
    public PortaMalas getPm() { return pm; }
    public void setPm(PortaMalas pm) { this.pm = pm; }
}

```

Classe PortaMalas:

```

import java.io.Serializable;

public class PortaMalas implements Serializable {
    private int litros;

    public PortaMalas() { }
    public PortaMalas(int litros) { this.litros = litros; }
    public int getLitros() { return litros; }
    public void setLitros(int litros) { this.litros = litros; }
}

```

Classe de teste 1 (serializa 3 carros):

```

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Principal1 {
    public static void main(String args[]){
        Carro carro1, carro2, carro3;
        carro1 = new Carro("Fusca", "verde", 200);
        carro2 = new Carro("Gol", "azul", 300);
        carro3 = new Carro("Kombi", "branca", 400);
        try {
            FileOutputStream outFile = new FileOutputStream("carros.ser");
            ObjectOutputStream os = new ObjectOutputStream(outFile);
            os.writeObject(carro1);
            os.writeObject(carro2);
            os.writeObject(carro3);
            os.close();
        }
    }
}

```

```
        System.out.println("Serialização concluída!");
    } catch (IOException e) {
        System.out.println("Problemas na serialização!");
        e.printStackTrace();
    }
}
```

Classe de teste 2 (dessaializa 3 carros):

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Principal2 {
    public static void main(String args[]){
        Carro carro1, carro2, carro3;
        try {
            FileInputStream inFile = new FileInputStream("carros.ser");
            ObjectInputStream in = new ObjectInputStream(inFile);
            carro1 = (Carro) in.readObject();
            carro2 = (Carro) in.readObject();
            carro3 = (Carro) in.readObject();
            in.close();
            System.out.println("Carro 1: "+carro1.getModelo()+
                "+carro1.getCor()+" - "+carro1.getPm().getLitros());
            System.out.println("Carro 2: "+carro2.getModelo()+
                "+carro2.getCor()+" - "+carro2.getPm().getLitros());
            System.out.println("Carro 3: "+carro3.getModelo()+
                "+carro3.getCor()+" - "+carro3.getPm().getLitros());
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado!");
        } catch (IOException e) {
            System.out.println("Erro na leitura dos dados!");
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            System.out.println("Classe não encontrada!");
            e.printStackTrace();
        }
    }
}
```

### 13.1.8      Serializando objetos em XML

O XML (eXtensible Markup Language) é largamente utilizado como formato de intercâmbio de dados entre sistemas. Nele os dados são organizados de forma hierárquica e identificados com tags.

Serializando em XML:

```
import java.beans.XMLEncoder;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class PrincipalXML1 {
    public static void main(String args[]){
        Carro carro1, carro2, carro3;
        carro1 = new Carro("Fusca", "verde", 200);
        carro2 = new Carro("Gol", "azul", 300);
        carro3 = new Carro("Kombi", "branca", 400);
        try {
            XMLEncoder encoder = new XMLEncoder(new
                BufferedOutputStream(new FileOutputStream("carros1.xml")));
            encoder.writeObject(carro1);
            encoder.writeObject(carro2);
            encoder.writeObject(carro3);
            encoder.close();
            System.out.println("Serialização concluída!");
        } catch (IOException e) {
            System.out.println("Problemas na serialização!");
            e.printStackTrace();
        }
    }
}
```

Desserializando um XML:

```
import java.beans.XMLDecoder;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class PrincipalXML2 {
    public static void main(String args[]){
        Carro carro1, carro2, carro3;
```

```
try {  
    XMLDecoder decoder = new XMLDecoder(new  
        BufferedInputStream(new FileInputStream("carros1.xml")));  
    carro1 = (Carro) decoder.readObject();  
    carro2 = (Carro) decoder.readObject();  
    carro3 = (Carro) decoder.readObject();  
    decoder.close();  
    System.out.println("Carro 1: "+carro1.getModelo()+" -  
        "+carro1.getCor()+" - "+carro1.getPm().getLitros());  
    System.out.println("Carro 2: "+carro2.getModelo()+" -  
        "+carro2.getCor()+" - "+carro2.getPm().getLitros());  
    System.out.println("Carro 3: "+carro3.getModelo()+" -  
        "+carro3.getCor()+" - "+carro3.getPm().getLitros());  
} catch (FileNotFoundException e) {  
    System.out.println("Arquivo não encontrado!");  
} catch (IOException e) {  
    System.out.println("Erro na leitura dos dados!");  
    e.printStackTrace();  
}  
}  
}  
}
```

## 13.2 ARQUIVOS DE TEXTO

Diferentemente dos arquivos serializados, que estão em um formato específico, arquivos de texto guardam caracteres ASCII que podem ser lidos em qualquer editor de textos.

Há diversas classes para manipular arquivos na API Java. Veremos a seguir apenas algumas, a título de exemplo. Recomenda-se uma busca na documentação da API Java a fim de identificar a classe mais adequada às necessidades do sistema que o programador estiver desenvolvendo.

### 13.2.1 **FileWriter** e **FileReader**

Um forma simples de ler e gravar fluxos de caracteres em arquivos de texto é utilizando as classes **FileWriter** (para escrita) e **FileReader** (para leitura). Elas assumem que a codificação e o tamanho de buffer padrão são suficientes.

O exemplo abaixo cria um arquivo chamado **teste.txt** por meio de um objeto **FileWriter** e grava, usando o método **write**, as letras maiúsculas de A a Z (converte os códigos ASCII de 65 a 91 em char). Em seguida, usa um objeto **FileReader** para ler os caracteres do arquivo, usando o método **read** (que retorna um inteiro). Os trechos mais relevantes estão destacados em vermelho.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploCaracteres {
    public static void main(String[] args) throws IOException {
        // gravando arquivo
        System.out.println("Gravando...");
        FileWriter arqGravacao = new FileWriter("teste.txt");
        for(int i = 65 ; i < 91; i++){
            arqGravacao.write((char)i);
        }
        arqGravacao.close();
        System.out.println("Gravação concluída!");

        //lendo arquivo
        System.out.println("*****");
        System.out.println("Lendo:");
        FileReader arqLeitura = new FileReader("teste.txt");
        int byteLido = 0;
        while(arqLeitura.ready()){
            byteLido = arqLeitura.read();
            System.out.println((char)byteLido);
        }
        arqLeitura.close();
    }
}
```

Algumas considerações importantes sobre o código acima:

- A declaração `new FileWriter("teste.txt")` faz com que a cada execução os dados anteriormente gravados sejam perdidos. Para permitir acrescentar dados a um arquivo existente, basta incluir o parâmetro true: `new FileWriter("teste.txt", true)`;
- Ao término de qualquer operação, o arquivo deve ser fechado para evitar problemas de inconsistência;
- O caminho para o arquivo pode ser fornecido como um String (no exemplo acima, "teste.txt") ou pode ser associado a um objeto previamente criado do tipo `File` ou `FileDescriptor`;
- Ao término do arquivo o método `read` retorna -1. Deste modo, poderíamos ter um while no seguinte formato: `while((byteLido = arqLeitura.read()) != -1)`;
- Observe a declaração `throws IOException` no método `main`. Como as classes e métodos que fazem entrada e saída (IO) lançam exceções do tipo `IOException`, e como esta classe representa exceções verificadas (*checked*), somos obrigados a dar o devido tratamento, capturando-as e tratando-as com `try/catch` ou propagando para o chamador com `throws`.

Neste exemplo, a fim de deixá-lo mais simples, optamos pelo throws. Mas como main é o primeiro método na pilha de execução, sabemos que não deveríamos fazer isto em uma situação real, pois a exceção não estaria sendo tratada por ninguém e encerraria o programa;

- As classes `FileWriter` e `FileReader` gravam os bytes no formato de caracteres. É possível gravar e ler em formato binário usando `FileOutputStream` e `FileInputStream`.

### 13.2.2 BufferedWriter e BufferedReader

As classes `BufferedWriter` e `BufferedReader` também servem para gravar e ler caracteres, porém, além de ter um desempenho superior, possuem alguns métodos como `newLine` e `readLine`. Porém, um `BufferedWriter` não se conecta diretamente a um arquivo físico, motivo pelo qual usa os servidores de um `FileWriter`, visto anteriormente. O mesmo ocorre com um `BufferedReader`, que utiliza um `FileReader`.

Para o exemplificar o uso destas classes, vamos utilizar uma classe `Produto` que possui como atributos nome (String), preco (double) e quantidade (int). Desta vez, faremos o tratamento das exceções com try/catch.

Vamos criar quatro produtos e gravar os valores de seus 3 atributos em um arquivo de texto, separados por ponto e vírgula, uma linha por produto (este formato é conhecido como CSV, de *comma separated values* ou valores separados por vírgula e pode ser importado facilmente em outros aplicativos, como planilhas eletrônicas).

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploBuffer {
    public static void main(String[] args) {
        Produto[] prod = new Produto[4];
        prod[0] = new Produto("Milho", 1.75, 36);
        prod[1] = new Produto("Pipoca", 2.99, 15);
        prod[2] = new Produto("Arroz", 3.50, 20);
        prod[3] = new Produto("Feijão preto", 9.99, 10);

        try{
            System.out.println("Gravando...");
            FileWriter arqGravacao = new FileWriter("produtos.csv");
            BufferedWriter buffer1 = new BufferedWriter(arqGravacao);

            for (int i = 0; i < 4; i++){
                String linha = prod[i].getNome() + ";" + prod[i].getPreco()
                             + ";" + prod[i].getQuantidade();
                buffer1.write(linha);
            }
            buffer1.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        buffer1.newLine();
        buffer1.flush();
    }

    buffer1.close();
    arqGravacao.close();
    System.out.println("Gravação concluída!");
} catch(IOException e){
    e.printStackTrace();
}

System.out.println("*****");
System.out.println("Lendo:");
try{
    FileReader arqLeitura = new FileReader("produtos.csv");
    BufferedReader buffer2 = new BufferedReader(arqLeitura);
    while(buffer2.ready()){
        String linha = buffer2.readLine();
        System.out.println(linha);
    }
    buffer2.close();
    arqLeitura.close();
} catch (IOException e){
    e.printStackTrace();
}
}
```

Algumas considerações sobre o código acima:

- O método `ready()` informa se o arquivo possui ou não linhas para a leitura ou se arquivo ainda está pronto para leitura;
- O método `close()` deve ser chamado quando a leitura for encerrada. Assim o arquivo é fechado e salvo (se ouver alguma alteração);
- O método `flush()` (descarga) força que os dados que estão no buffer sejam efetivamente gravados em disco. Se ele não estiver presente, o próprio `close()` faz o flush de todos os dados antes de fechar o arquivo.
- O método `readLine()` retorna uma String com a uma linha completa do arquivo sem o \n e avança uma posição. Quando retornar nulo, também quer dizer que não há mais linhas a serem lidas;
- Se quisermos ler o valor de cada atributo individualmente, podemos usar um Scanner com ";" como delimitador. Para isto, basta substituir o while do código anterior pelas linhas destacadas em vermelho no recorte de código abaixo:

```

try{
    FileReader arqLeitura = new FileReader("produtos.csv");
    BufferedReader buffer2 = new BufferedReader(arqLeitura);
    Scanner sc = new Scanner(buffer2).useDelimiter(",");
    while(sc.hasNext()) {
        if(sc.hasNextDouble())
            System.out.print(sc.nextDouble()+"\t");
        else if(sc.hasNextInt())
            System.out.print(sc.nextInt()+"\t");
        else
            System.out.print(sc.next()+"\t");
    }
}

```

A figura a seguir mostra a saída deste programa com `readLine()`:

---

```

Gravando...
Gravacao concluida!
*****
Lendo:
Milho;1.75;36
Pipoca;2.99;15
Arroz;3.5;20
Feijao preto;9.99;10

```

Já a figura a seguir mostra a saída formatada com `Scanner`. O caractere especial `\t` cria uma tabulação:

---

```

Gravando...
Gravacao concluida!
*****
Lendo:
Milho  1.75   36
Pipoca 2.99   15
Arroz   3.5    20
Feijao  9.99   10

```

### 13.3 BIBLIOGRAFIA DO CAPÍTULO

SIERRA, Kathy; BATES, Bert. **Use a Cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2007.

ASCENIO, Ana Fernanda Gomes; CAMPOS, Edilene A. V. **Fundamentos da Programação de Computadores**. 2. ed. São Paulo: Pearson Prentice Hall, 2007.