



# Linguagem de Programação II

Gustavo Campos Menezes

Ygor Colen Morato

Hilário Seibel Júnior

Curso Técnico em Planejamento e  
Gestão em Tecnologia da Informação





# Linguagem de Programação II

*Gustavo Campos Menezes*

*Ygor Colen Morato*

*Hilário Seibel Júnior*



Belo Horizonte - MG

2013

Presidência da República Federativa do Brasil  
Ministério da Educação  
Secretaria de Educação Profissional e Tecnológica

© Centro Federal de Educação Tecnológica de Minas Gerais  
Este Caderno foi elaborado em parceria entre o Centro Federal de Educação  
Tecnológica de Minas Gerais e a Universidade Federal de Santa Catarina para a  
Rede e-Tec Brasil.

**Equipe de Elaboração**

Centro Federal de Educação Tecnológica de  
Minas Gerais – CEFET-MG

**Coordenação Institucional**

José Wilson da Costa/CEFET-MG

**Coordenação do Curso**

Adelson de Paula Silvia/CEFET-MG

**Professores-autores**

Gustavo Campos Menezes/CEFET-MG  
Ygor Colen Morato/CEFET-MG  
Hilário Seibel Júnior/CEFET-MG

**Comissão de Acompanhamento e Validação**

Universidade Federal de Santa Catarina – UFSC

**Coordenação Institucional**

Araci Hack Catapan/UFSC

**Coordenação do Projeto**

Silvia Modesto Nassar/UFSC

**Coordenação de Design Instrucional**

Beatriz Helena Dal Molin/UNIOESTE e UFSC

**Coordenação de Design Gráfico**

Juliana Tonietto/UFSC

**Design Instrucional**

Renato Cislaghi/UFSC

**Web Master**

Rafaela Lunardi Comarella/UFSC

**Web Design**

Beatriz Wilges/UFSC  
Mônica Nassar Machuca/UFSC

**Diagramação**

Liana Domeneghini Chiaradia/UFSC  
Marília Ceriolli Hermoso/UFSC

**Revisão**

Júlio César Ramos/UFSC

**Projeto Gráfico**

e-Tec/MEC

**Catalogação na fonte pela Biblioteca Universitária da  
Universidade Federal de Santa Catarina**

**M543I Menezes, Gustavo Campos  
Linguagem de programação II / Gustavo Campos  
Menezes, Ygor Colen Morato, Hilário Seibel  
Júnior. – Belo Horizonte : Centro Federal de  
Educação Tecnológica de Minas Gerais, 2013.  
84 p., il., tabs.**

**Inclui bibliografia**

- 1. Linguagem de programação (computadores).**
- 2. Java (Linguagem de programação de computador).**
- I. Morato, Ygor Colen. II. Seibel Júnior, Hilário.**
- III. Título.**

# Apresentação e-Tec Brasil

Bem-vindo a Rede e-Tec Brasil!

Você faz parte de uma rede nacional de ensino, que por sua vez constitui uma das ações do Pronatec - Programa Nacional de Acesso ao Ensino Técnico e Emprego. O Pronatec, instituído pela Lei nº 12.513/2011, tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira proporcionando caminho de acesso mais rápido ao emprego.

É neste âmbito que as ações da Rede e-Tec Brasil promovem a parceria entre a Secretaria de Educação Profissional e Tecnológica (SETEC) e as instâncias promotoras de ensino técnico como os Institutos Federais, as Secretarias de Educação dos Estados, as Universidades, as Escolas e Colégios Tecnológicos e o Sistema S.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

A Rede e-Tec Brasil leva diversos cursos técnicos a todas as regiões do país, incentivando os estudantes a concluir o ensino médio e realizar uma formação e atualização contínuas. Os cursos são ofertados pelas instituições de educação profissional e o atendimento ao estudante é realizado tanto nas sedes das instituições quanto em suas unidades remotas, os polos.

Os parceiros da Rede e-Tec Brasil acreditam em uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação  
Março de 2013

Nossa contato

[etecbrasil@mec.gov.br](mailto:etecbrasil@mec.gov.br)



# Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.



**Atenção:** indica pontos de maior relevância no texto.



**Saiba mais:** oferece novas informações que enriquecem o assunto ou “curiosidades” e notícias recentes relacionadas ao tema estudado.



**Glossário:** indica a definição de um termo, palavra ou expressão utilizada no texto.



**Mídias integradas:** sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.



**Atividades de aprendizagem:** apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.



# Sumário

<b>Palavra dos professores-autores.....</b>	<b>9</b>
<b>Apresentação da disciplina.....</b>	<b>11</b>
<b>Projeto instrucional.....</b>	<b>13</b>
<b>Aula 1 – Linguagem Java – primeiros passos.....</b>	<b>15</b>
1.1 Sobre a linguagem Java.....	15
1.2 Criação de programas em Java.....	15
1.3 Primeiro exemplo de programa em Java.....	21
1.4 Segundo exemplo de programa em Java.....	25
1.5 Incluindo componentes nos painéis.....	29
1.6 Adicionando funcionalidades ao programa.....	32
1.7 Tipos de dados e definição de variáveis.....	37
1.8 <i>Strings</i> .....	39
1.9 <i>Arrays</i> .....	40
1.10 Operadores.....	41
1.11 Funções matemáticas.....	42
1.12 Controle de fluxo.....	42
1.13 Escopo das variáveis.....	43
<b>Aula 2 – Orientação a objetos: classes, objetos e métodos.....</b>	<b>49</b>
2.1 Introdução à orientação a objetos.....	49
2.2 Classe e objeto.....	49
2.3 Classes e objetos em Java.....	50
2.4 Métodos.....	52
<b>Aula 3 – Orientação a objetos: métodos e encapsulamento.....</b>	<b>57</b>
3.1 Construtores.....	57
3.2 Encapsulamento.....	60

<b>Aula 4 – Polimorfismo (herança, sobrecarga e reescrita)</b>	<b>67</b>
4.1 Herança.....	67
4.2 Sobrecarga de métodos.....	70
4.3 Reescrita de métodos.....	71
4.4 Pacotes.....	72
<b>Aula 5 – Controle de erros: exceções</b>	<b>75</b>
5.1 <i>Try-catch</i> .....	75
5.2 <i>Try-finally</i> .....	76
5.3 Tipos de erros.....	78
5.4 Gerando uma exceção.....	78
5.5 Criando suas próprias exceções.....	81
<b>Referências</b>	<b>83</b>
<b>Curriculum dos professores-autores</b>	<b>84</b>

# Palavra dos professores-autores

Prezado estudante!

É um prazer tê-lo conosco!

O CEFET-MG oferece a você, juntamente com as prefeituras e o Governo Federal, o Curso Técnico em Planejamento e Gestão em Tecnologia da Informação (PGTI). Embora o curso seja oferecido na modalidade a distância, esperamos que haja uma grande interação entre nós. Atualmente, com os recursos na área de Tecnologia da Informação (*e-mail, chat, videoconferência etc.*), podemos manter uma comunicação bastante produtiva.

O conteúdo de Linguagens de Programação II, como você já deve saber (uma vez que já fez as disciplinas de Algoritmos e Lógica de Programação e Linguagens de Programação I), exige bastante esforço e dedicação. Portanto, esperamos contar sempre com a sua presença nos *chats*, ressaltando que você pode também enviar suas dúvidas por *e-mail* para os tutores e/ou professores.

Nunca se esqueça que em qualquer instituição de ensino (seja o curso presencial ou a distância), o principal responsável pelo sucesso é você, estudante. E esse sucesso só pode ser obtido mediante o esforço, a dedicação e disciplina!

Então, desejamos a você SUCESSO!!!



# Apresentação da disciplina

Prezado estudante,

Produzimos este material com o objetivo de servir como um guia básico de referência para a disciplina de Linguagens de Programação II. Nesta disciplina vamos estudar os principais conceitos referentes à Programação Orientada a Objetos utilizando como ferramenta para demonstração de exemplos e realização de exercícios a linguagem Java.

Além disso, no final do caderno é apresentada a bibliografia complementar que pode ajudar na compreensão de cada conteúdo abordado.

Bons estudos!



# Projeto instrucional

**Disciplina:** Linguagem de Programação II (carga horária: 60 h).

**Ementa:** Conceitos básicos e ambiente de desenvolvimento integrado. Comandos básicos da linguagem Java. Orientação a objeto em Java. Programação Java com interface gráfica.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Linguagem Java – primeiros passos	Aprender as principais características da linguagem Java. Compreender os procedimentos necessários para a instalação de um conjunto de ferramentas necessárias para o desenvolvimento dos primeiros programas em Java.	Fórum de discussões, <i>chat</i> .	20
2. Orientação a objetos	Compreender o que é orientação a objetos e o que é e como criamos uma classe e um objeto em Java.	Fórum de discussões, <i>chat</i> , ferramenta para desenvolvimento de programas.	10
3. Métodos e encapsulamento	Desenvolver a habilidade de encapsular os dados de uma classe e de criar métodos e construtores que manipulem esses dados.	Fórum de discussões, <i>chat</i> , ferramenta para desenvolvimento de programas.	10
4. Polimorfismo e herança	Desenvolver a habilidade de montar uma hierarquia de classes através do conceito de herança, sobrecarga e reescrita.	Fórum de discussões, <i>chat</i> , ferramenta para desenvolvimento de programas.	10
5. Tratamento de erros: exceções	Criar tratamento para os diversos erros que podem ocorrer durante a execução de um programa.	Fórum de discussões, <i>chat</i> , ferramenta para desenvolvimento de programas.	10



# Aula 1 – Linguagem Java – primeiros passos

## Objetivos

Aprender as principais características da linguagem Java.

Compreender os procedimentos necessários para a instalação de um conjunto de ferramentas necessárias para o desenvolvimento dos primeiros programas em Java.

### 1.1 Sobre a linguagem Java

A linguagem Java foi desenvolvida para ser utilizada em praticamente qualquer dispositivo eletrônico (controle remoto, eletrodomésticos, computadores, carros etc.). Infelizmente, com a falta de financiamento na época (por volta de 1993), ela só foi se popularizar em meados de 1995 e 1996 com o surgimento e popularização da internet; desde então, não parou de crescer. Atualmente ela é utilizada em desde pequenos dispositivos (telefones celulares, GPS, livros eletrônicos etc.) até no controle de grandes aplicações corporativas e, evidentemente, na internet.

### 1.2 Criação de programas em Java

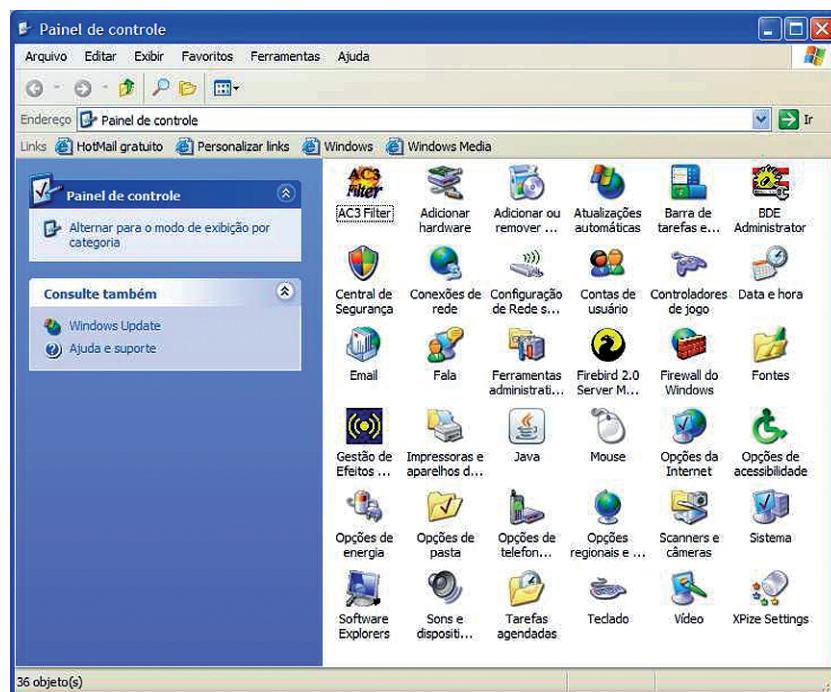
A ferramenta Java é constituída de diversos produtos. O grande diferencial dessa ferramenta é a capacidade de rodar em diferentes máquinas e dispositivos. Como em qualquer linguagem de programação, Java necessita de um compilador. O **compilador** traduz o programa escrito em Java para uma linguagem intermediária chamada **Java bytewords** – um código independente de plataforma que é interpretado por um interpretador Java. Para que um programa em Java seja executado, portanto, é necessário possuir outra ferramenta chamada **interpretador**, que estará emulando a *Java Virtual Machine (JVM)*, Máquina Virtual do Java, em qualquer computador e sistema operacional que esteja usando.

Para instalar o *kit* de desenvolvimento da Sun, é necessário realizar os seguintes passos:

- a) realizar o *download* do *kit* de desenvolvimento Java 6 SDK (*Standard Edition – J2SE*), popularmente conhecido como JDK (*Java Development Kit*). Para isso, acesse o site <http://java.sun.com> e escolha a versão de acordo com o sistema operacional que será utilizado.

Após instalar a ferramenta, é necessário criar uma variável de ambiente chamada JAVA\_HOME (que deve guardar o caminho do diretório onde o Java foi instalado) e adicioná-la ao PATH do seu S.O. Veja a seguir a sintaxe no Windows (para a versão 1.5.0 do JDK, por exemplo), com os passos necessários para alterar as variáveis de ambiente no Windows XP.

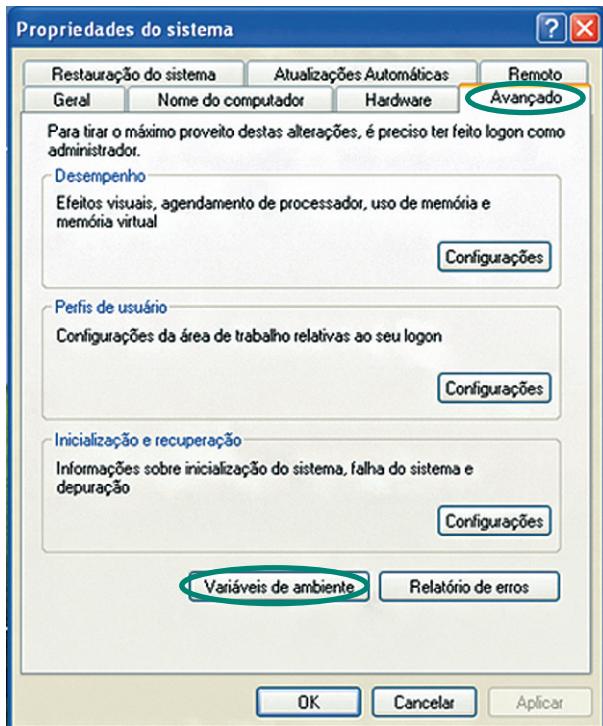
O primeiro passo é abrir o painel de controle do Windows, como pode ser visto na Figura 1.1:



**Figura 1.1: Visão do painel de controle do Windows**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 1

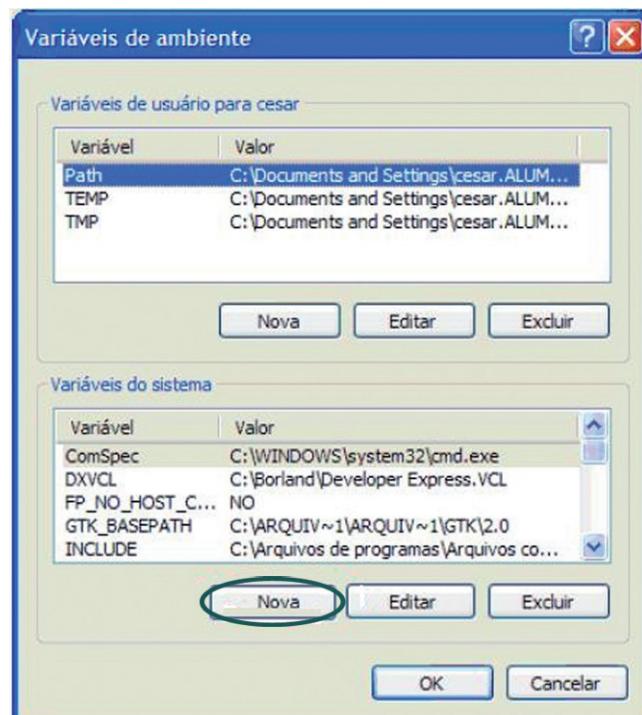
Feito isso, dê duplo clique no ícone Sistemas. A tela de "Propriedades do Sistema" será exibida, como mostra a Figura 1.2:



**Figura 1.2: Visão das propriedades de sistema do Windows**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 2

Selecione a guia “Avançado”, depois clique em “Variáveis de ambiente”. Será exibida a tela mostrada na Figura 1.3:

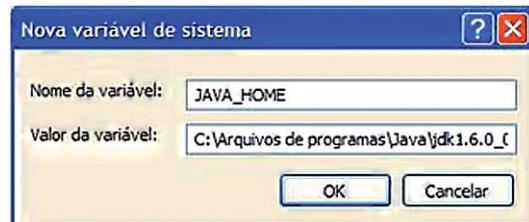


**Figura 1.3: Visão das variáveis de ambiente do Windows**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 3

Depois de abrir a tela de Variáveis de Ambiente, o nosso próximo passo é encontrar nosso JDK, saber onde está instalado o “pacote Java”.

Normalmente ele fica no diretório: “C:\Arquivos de programas\Java”; caso não encontre nesse diretório, localize em seu sistema a pasta JAVA. Com o diretório Java localizado; vamos continuar. Clique no botão “Nova”, localizado na GroupBox “Variáveis do sistema”.



**Figura 1.4: Criando nova variável de ambiente do Windows**

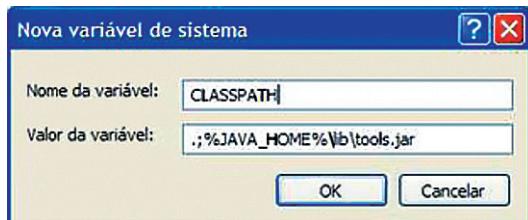
Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br) p. 3

Nessa tela vamos informar:

- nome de variável: JAVA\_HOME
- valor da variável: C:\Arquivos de programas\Java\jdk1.6.0\_02

Atente para o fato de que a versão do seu JDK pode ser diferente. Confirme, então, se o nome do diretório é jdk1.6.0\_02. Feito isso, pode clicar em “OK”!

Vamos clicar novamente no mesmo botão “Nova”, localizado na *GroupBox* “Variáveis do sistema”. E agora vamos informar esses valores:



**Figura 1.5: Criando nova variável de ambiente do Windows**

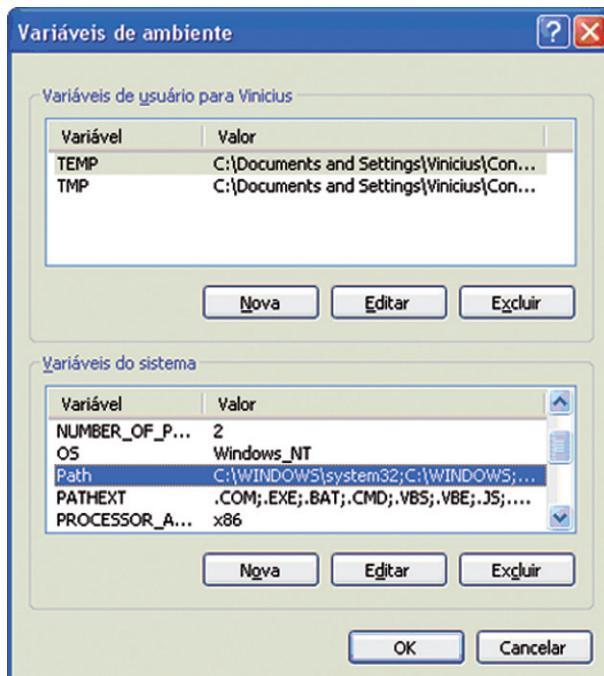
Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 4

Nessa tela vamos informar:

- nome de variável: **CLASSPATH**
- valor da variável: **.;%JAVA\_HOME%\lib\tools.jar**

Feito isso, pode clicar no “OK”!

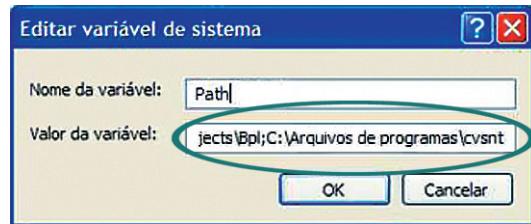
Agora vamos localizar a variável PATH, selecionar e depois clicar em “Editar”.



**Figura 1.6: Editando a variável PATH**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br) p. 5

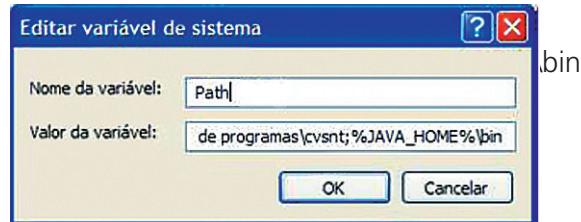
A nova tela irá se abrir, como mostra a Figura 1.7:



**Figura 1.7: Detalhe editando a variável PATH**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 5

Nessa tela, iremos acrescentar a seguinte informação no campo “Valor da Variável”:



**Figura 1.8: Detalhe editando a variável PATH**

Fonte: Apostila de variáveis de ambiente, [www.cseg.eng.br](http://www.cseg.eng.br), p. 6

Feito isso, você já pode clicar no “OK”! Clique em “OK” da tela de seguin-  
te e depois feche a tela. Com isso você terá as variáveis de Ambiente Java  
configuradas!

Lembre-se de reinicializar o PC!



- a) para testar a instalação, entre no *prompt* do DOS (menu Iniciar > Progra-  
mas > Acessórios > *Prompt* de comando), digite javac e pressione a tecla  
Enter. Se aparecer uma tela contendo informações de *help*, significa que a  
instalação foi realizada com sucesso. Caso contrário, aparecerá uma men-  
sagem informando que o programa ainda não é reconhecido pelo sistema.

Existem diversos IDEs (do inglês *Integrated Development Environment*, que  
significa Ambiente Integrado de Desenvolvimento) com ferramentas de  
apoio ao desenvolvimento de softwares em Java, com o objetivo de agilizar  
esse processo.

Nesta disciplina utilizaremos o “*NetBeans IDE*”. Para baixá-lo, acesse o site [http://www.netbeans.org/index\\_pt\\_BR.html](http://www.netbeans.org/index_pt_BR.html) com a versão mais recente (a versão utilizada neste material é a 6.7.1). Na página de *downloads* você também pode baixar diretamente o JDK com o *NetBeans IDE Java SE* já embutido. Nesse caso não será necessário executar os passos de instalação e configuração do Java descritos anteriormente.

Antes de realizar o *download*, não se esqueça de confirmar que o programa será baixado em português.

## 1.3 Primeiro exemplo de programa em Java

Para começar nossos estudos, criaremos um programa em Java no qual, ao clicar em um botão, será mostrada uma mensagem dizendo “Olá Mundo!”.

O primeiro passo é abrirmos o *NetBeans* e criarmos um novo projeto.

Para isso, realize os passos a seguir:

- a) clique em “**Arquivo**” > “**Novo Projeto**”;
- b) entre as opções “Categorias” da janela aberta, escolha “**Java**”. Dentro de “**Projetos**”, escolha “**Aplicativo Java**”. Clique em “**Próximo**”;
- c) escolha um nome para o projeto (por exemplo, “**Ola\_Mundo**”) e certifique-se de que apenas a opção “**Definir como projeto principal**” esteja selecionada. Clique em “**Finalizar**”.

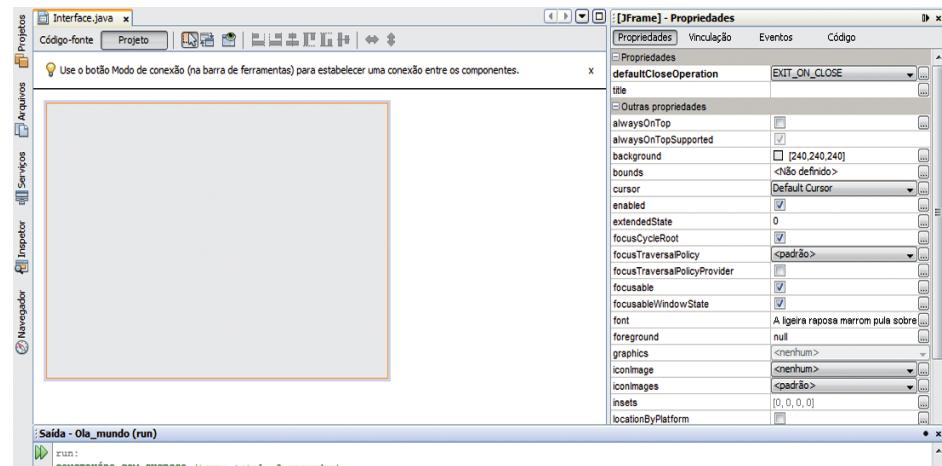
Após criarmos o projeto, ele passa a ser exibido na aba “**Projetos**”, que fica no canto esquerdo do *NetBeans*. Para facilitar seu aprendizado, começaremos o projeto desenvolvendo a interface gráfica de nosso programa.

Para isso, precisamos criar um “quadro” (*ou frame*) que é o espaço onde os componentes de nossa interface serão exibidos. Esse quadro pode ser criado da seguinte forma:

- a) na aba de “Projeto”, clique com o botão direito do mouse no nome de nosso projeto;
- b) escolha as opções “**Novo**” > “**Formulário JFrame**”;

- c) escolha um nome para o *frame* e outro para o pacote em que ele estará inserido (por exemplo, “**Interface**” para o nome do *frame* e “**pacoteInterface**” para o nome do pacote). Mais adiante explicaremos detalhadamente para que servem os “pacotes” em Java. Clique em “**Finalizar**”.

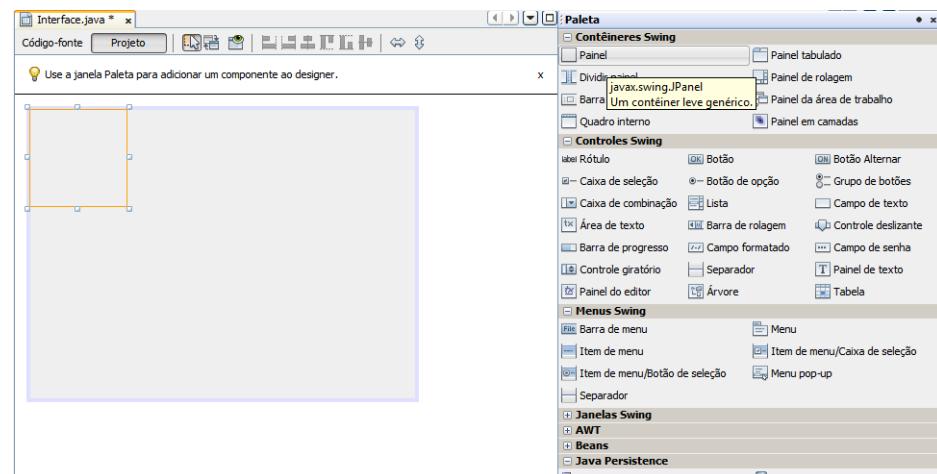
A Figura 1.9 apresenta um trecho da tela mostrada após a realização dos passos anteriormente:



**Figura 1.9: Criando um *frame* para a interface do programa**

Fonte: Elaborada pelo autor Ygor Morato

O quadro cinza à esquerda da Figura 1.9 é o *frame* onde os componentes da interface irão aparecer. Na aba “**Paleta**”, no canto direito da Figura 1.9, estão os componentes que iremos inserir. Para começar, criaremos um “**Painel**” onde o usuário irá digitar o nome e a matrícula de um estudante. Para isso, clique em “**Painel**” na opção “**Contêineres Swing**” da aba “**Paleta**”. Arraste o ponteiro do *mouse* para a área do *frame* e você verá uma tela como a mostrada na Figura 1.10.

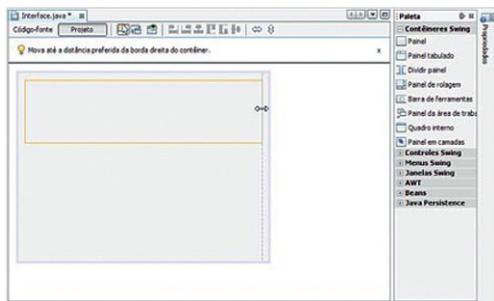


**Figura 1.10: Inserindo um painel na interface do programa**

Fonte: Elaborada pelo autor Ygor Morato

A área quadrada com uma borda laranja vista na Figura 1.10 indica o local onde o painel será inserido no *frame*. A linha pontilhada mostra uma margem dentro do *frame*, útil para alinharmos os componentes. Devemos clicar com o *mouse* no local indicado para definir a área onde será inserido o painel. Em seguida, iremos redimensionar o painel para que caibam os componentes que criaremos mais adiante.

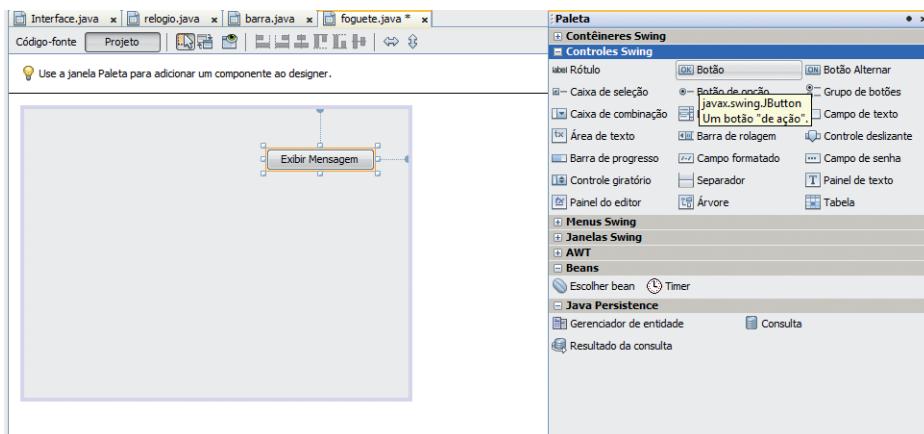
A Figura 1.11 mostra o redimensionamento do painel.



**Figura 1.11: Redimensionando um painel**

Fonte: Elaborada pelo autor Ygor Morato

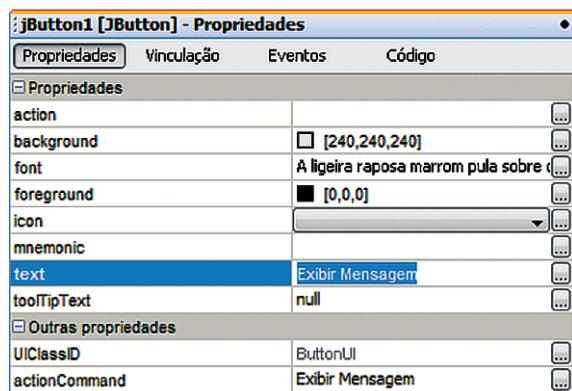
Agora iremos inserir um botão de ação dentro do JPanel. Deveremos mudar o texto que aparece no botão para "Exibir Mensagem", como na Figura 1.12:



**Figura 1.12: Inserindo botão de ação**

Fonte: Elaborada pelo autor Ygor Morato

Para efetuar a mudança desse texto, você precisa clicar no botão e abrir as propriedades que ficam à direita; na propriedade *text*, como na Figura 1.13, basta você colocar o valor desejado.



**Figura 1.13: Janela de propriedades**

Fonte: Elaborada pelo autor Ygor Morato

Devemos agora colocar um título na janela, que é nosso JFrame. Para isso devemos clicar no JFrame e chamar suas propriedades e alterar a propriedade “Title” para o valor desejado, como na Figura 1.14.



**Figura 1.14: Janela de propriedades.**

Fonte: Elaborada pelo autor Ygor Morato

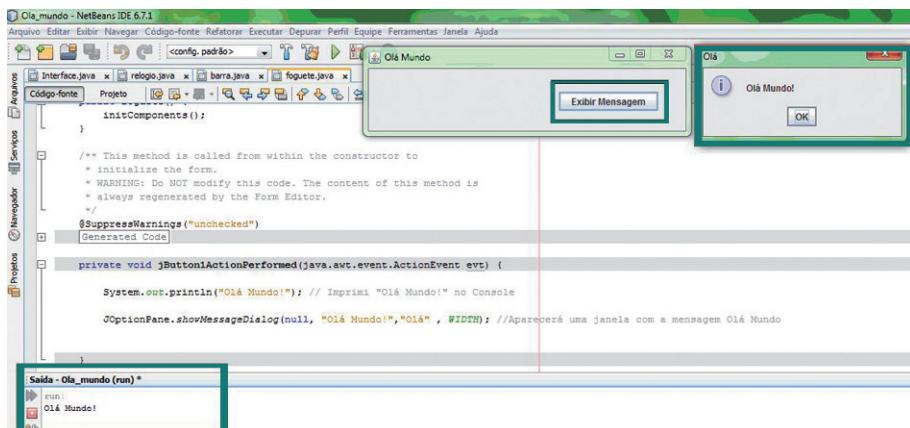
Agora, iremos fazer nossa primeira codificação em Java; para isso, dê dois cliques no botão de ação e irá aparecer a janela de codificação, como na Figura 1.15. Digite o código como o da Figura 1.15 e rode o programa através de “SHIFT + F6”.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    System.out.println("Olá Mundo!"); // Imprimi "Olá Mundo!" no Console
    JOptionPane.showMessageDialog(null, "Olá Mundo!", "Olá", WIDTH); //Aparecerá uma janela com a mensagem Olá Mundo
}
```

**Figura 1.15: Área de codificação**

Fonte: Elaborada pelo autor Ygor Morato.

Irá aparecer uma janela com o seu programa que foi desenhado e, ao clicar no botão de ação, deverá aparecer uma mensagem que está ao lado do programa. Lá embaixo, como saída via console, aparece nossa segunda mensagem:



**Figura 1.16: Resultado de saída do primeiro programa**

Fonte: Elaborada pelo autor Ygor Morato

É extremamente importante que você tente executar os passos apresentados antes de continuar a leitura do material. Portanto, tente desenvolver um projeto com a interface semelhante à que fizemos até aqui. Apresente seu trabalho no fórum criado para a postagem deste trabalho.



## 1.4 Segundo exemplo de programa em Java

Criaremos um programa em Java que calcula as áreas de três figuras geométricas: triângulo, quadrado e retângulo. No programa você deverá marcar qual área deseja calcular. Com base nisso, deverão aparecer as caixas que informam os valores necessários para o cálculo de determinada área e, ao clicar no botão, calcular, o valor encontrado deve ser informado em uma caixa de resultado.

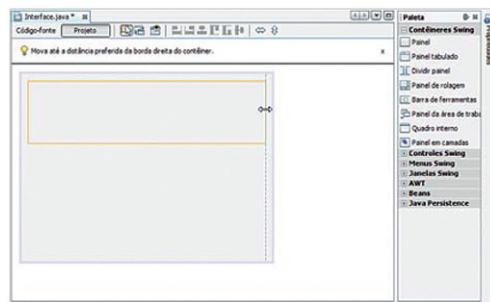
O primeiro passo é abrirmos o *NetBeans* e criarmos um novo projeto.

Para isso, realize os seguintes passos:

- clique em “Arquivo” > “Novo Projeto”;
- dentre as opções “Categorias” da janela aberta, escolha “Java”. Dentro de “Projetos”, escolha “Aplicativo Java”. Clique em “Próximo”;
- escolha um nome para o projeto (por exemplo, “Área”) e certifique-se de que apenas a opção “Definir como projeto principal” esteja selecionada. Clique em “Finalizar”.

Após criarmos o projeto, ele passa a ser exibido na aba “**Projetos**”, que fica no canto esquerdo do *NetBeans*. Para facilitar seu aprendizado, começaremos o projeto desenvolvendo a interface gráfica de nosso programa. Para isso precisamos criar um “quadro” (ou *frame*) que é o espaço onde os componentes de nossa interface serão exibidos. Esse quadro pode ser criado da seguinte forma:

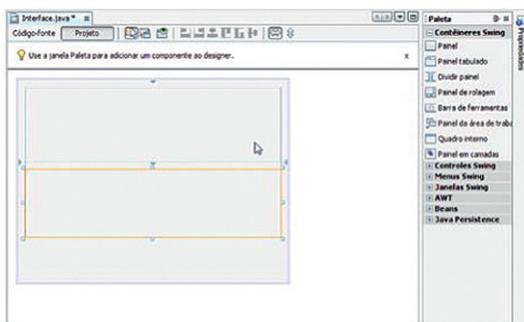
- a) na aba de “Projeto”, clique com o botão direito do *mouse* no nome de nosso projeto;
- b) escolha as opções “**Novo**” > “**Formulário JFrame**”;
- c) escolha um nome para o *frame* e outro para o pacote em que ele estará inserido (por exemplo, “**Interface**” para o nome do *frame* e “**pacotel-interface**” para o nome do pacote). Mais adiante explicaremos detalhadamente para que servem os “pacotes” em Java. Clique em “**Finalizar**”.



**Figura 1.17: Interface gerada**

Fonte: Elaborada pelo autor Ygor Morato

Lembre-se que na aba “**Paleta**”, no canto direito da Figura 1.17, estão os componentes que iremos inserir. Para começar, criaremos um painel onde o usuário irá escolher a opção que deseja calcular. Para isso clique em “**Painel**” na opção “**Contêineres Swing**” da aba “**Paleta**”. Arraste o ponteiro do *mouse* para a área do *frame* e você verá uma tela como a mostrada na Figura 1.18. A área quadrada com uma borda laranja vista na figura indica o local onde o painel será inserido no *frame*. A linha pontilhada mostra uma margem dentro do *frame*, útil para alinharmos os componentes. Devemos clicar com o *mouse* no local indicado para definir a área onde será inserido o painel. Em seguida iremos redimensionar o painel para que caibam os componentes que criaremos mais adiante. A Figura 1.18 mostra o redimensionamento do painel.

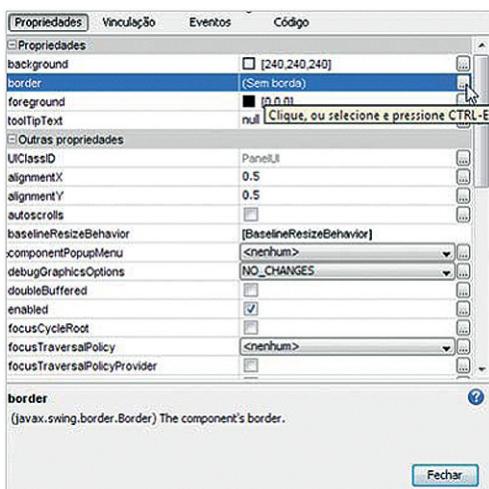


**Figura 1.18: Contêineres**

Fonte: Elaborada pelo autor Hilário Júnior

Nosso programa conterá dois painéis: um para escolha de qual área deverá ser calculada e outro para informar os dados dessa área. Execute os passos apresentados e, em seguida, insira outro painel em seu programa.

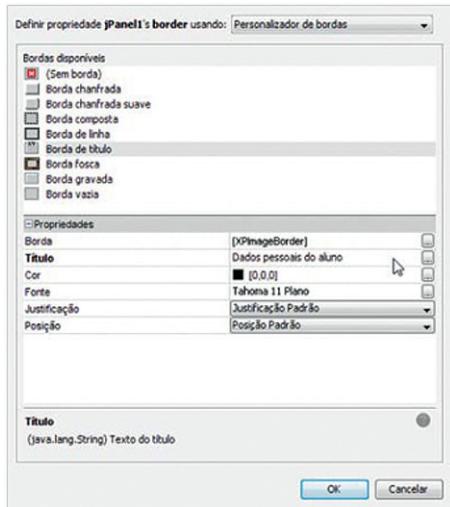
Como queremos distinguir as opções de área e o local de entrada dos dados, adicionaremos uma borda e um título aos nossos painéis. Primeiro clicamos com o botão direito sobre o painel de cima e escolhemos a opção “**Propriedades**”. Será exibida uma tela como a mostrada na Figura 1.19.



**Figura 1.19: Janela de propriedades**

Fonte: Elaborada pelo autor Hilário Júnior

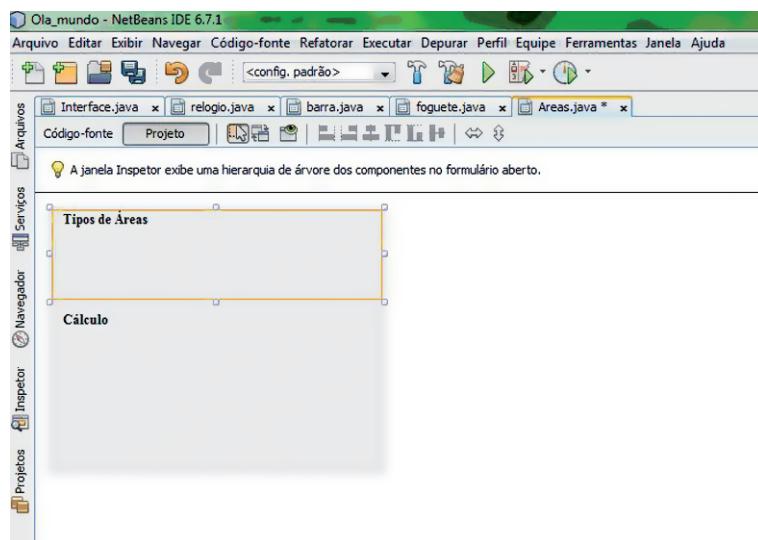
O item “**border**” que aparece em destaque na Figura 1.19 define como será essa borda que queremos alterar no painel. Para modificar esse atributo, devemos clicar no botão do item. Clicando nele será exibida uma tela semelhante à mostrada na Figura 1.20.



**Figura 1.20: Modificando a borda do painel**

Fonte: Elaborada pelo autor Ygor Morato

Na tela mostrada na Figura 1.20 escolhemos o tipo da borda como “**Borda de título**” e definimos o título como “**Tipos de Áreas**”. Podemos também alterar a fonte com a qual o título será escrito, clicando no botão ao lado do item “**Fonte**”. Após clicar no botão do item “**Fonte**”, definimos a fonte com estilo “**Negrito**” e de tamanho “**12**”, e clicamos em seguida em “**OK**”. Voltando à tela da Figura 1.20, clicamos novamente em “**OK**”, depois no botão “**Fechar**”, e visualizamos o painel com a borda e o título definidos nos passos anteriores. A Figura 1.21 mostra a modificação na borda e no título do painel:



**Figura 1.21: Bordas dos painéis modificadas**

Fonte: Elaborada pelo autor Hilário Júnior

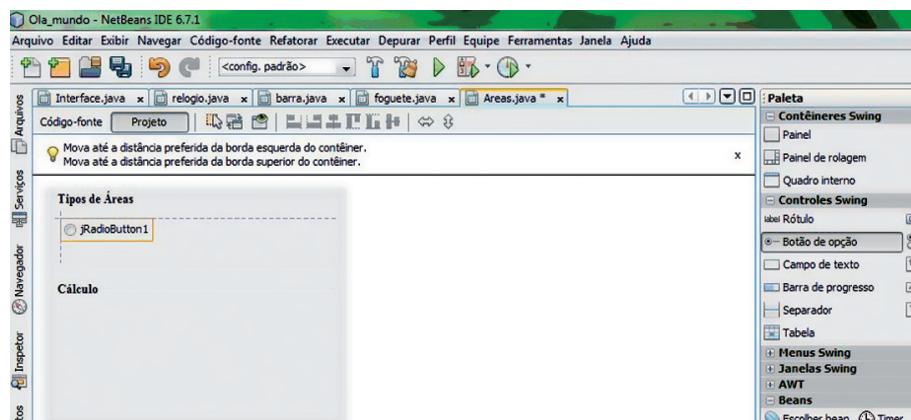
Na tela mostrada na Figura 1.21 estão a borda e o título do painel onde ficarão as opções que já foram modificados (repita os passos anteriores para que isso seja feito em seu programa).

É extremamente importante que você tente executar os passos apresentados antes de continuar a leitura do material. Portanto, tente desenvolver um projeto com a interface semelhante à que fizemos até aqui.

## 1.5 Incluindo componentes nos painéis

Expandindo a opção “**Controles Swing**” da aba “**Paleta**”, encontramos os componentes que iremos inserir em nossos painéis. O primeiro item a ser preenchido pelo usuário será a escolha de uma das opções de cálculo. Para isso iremos inserir um “Botão de Opção”, onde deverá ser marcada uma das opções que servem para explicação do que deve ser inserido nesse campo.

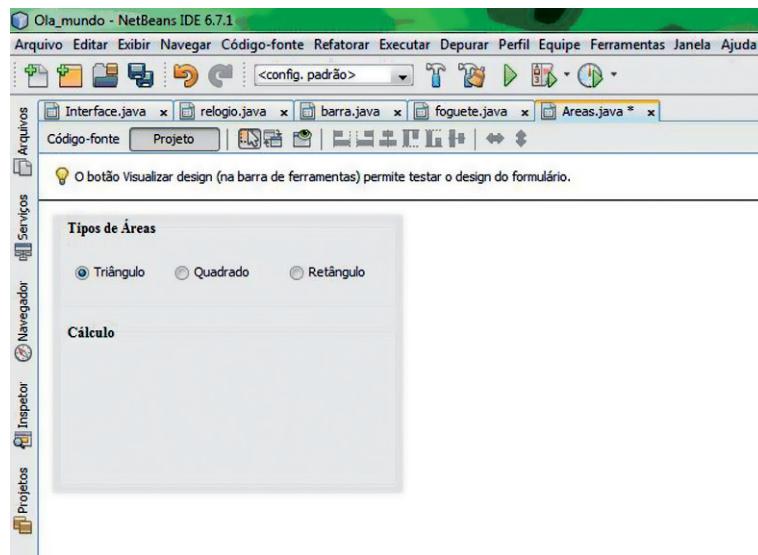
Para isso clicamos primeiro em “Botão de Opção” e definimos o local onde o botão será inserido, como mostrado na Figura 1.22:



**Figura 1.22: Inserindo um botão de opção**

Fonte: Elaborada pelo autor Ygor Morato

Para definirmos o texto que aparecerá no botão de opção, clicamos com o botão direito nele e escolhemos a opção “**Editar texto**”, inserindo o texto “Triângulo”. Devemos repetir esse processo três vezes, pois precisamos ter os três “botões de opção”, como mostra a Figura 1.23:

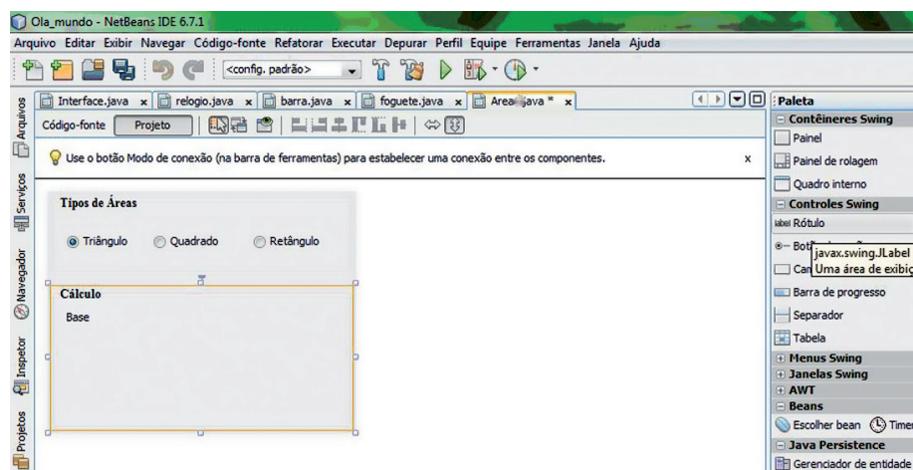


**Figura 1.23: Detalhes botões de opção**

Fonte: Elaborada pelo autor Hilário Júnior

Expandindo a opção “**Controles Swing**” da aba “**Paleta**”, encontramos os componentes que iremos inserir no nosso painel Cálculo. O primeiro item a ser preenchido pelo usuário nessa área, se o cálculo for da área do triângulo, será o valor da base desse triângulo. Para isso iremos inserir um “**Campo de Texto**”, onde esse valor será digitado, e um “**Rótulo**”, que serve para expli-cação do que deve ser inserido nesse campo.

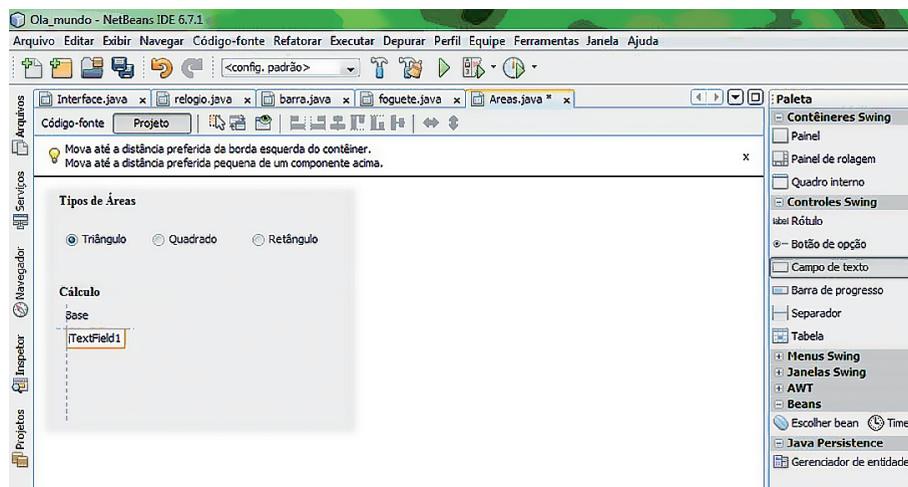
Para isso clicamos primeiro em “Rótulo” e definimos o local onde o rótulo será inserido, como mostrado na Figura 1.24.



**Figura 1.24: Inserindo um rótulo**

Fonte: Elaborada pelo autor Hilário Júnior

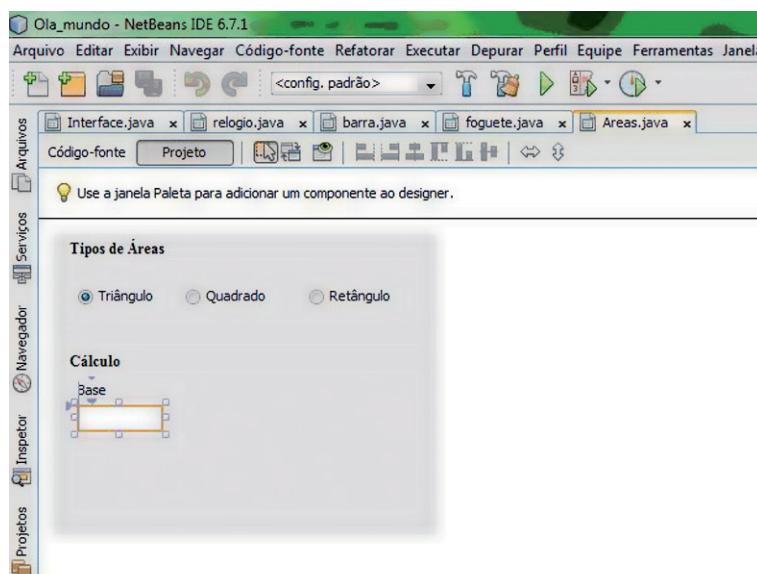
Para definirmos o texto que aparecerá no rótulo, clicamos com o botão direito nele e escolhemos a opção “**Editar texto**”, inserindo o texto “Base”. Ao lado do rótulo incluímos um “Campo de Texto”, como mostrado na Figura 1.25.



**Figura 1.25: Inserindo um campo de texto**

Fonte: Elaborada pelo autor Ygor Morato

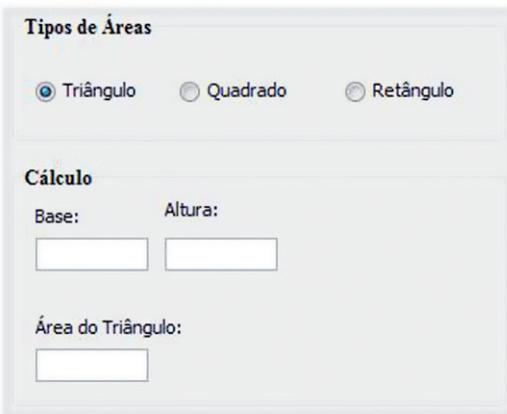
Em seguida editamos o texto do “Campo de texto” criado anteriormente (nesse caso, deixamos o campo em branco para que o usuário digite o valor da base) e redimensionamos o campo, produzindo uma tela semelhante à mostrada na Figura 1.26.



**Figura 1.26: Limpando e redimensionando o campo de texto**

Fonte: Elaborada pelo autor Ygor Morato

Continue desenvolvendo a interface de nosso programa. Para isso crie uma tela semelhante à mostrada na Figura 1.27.



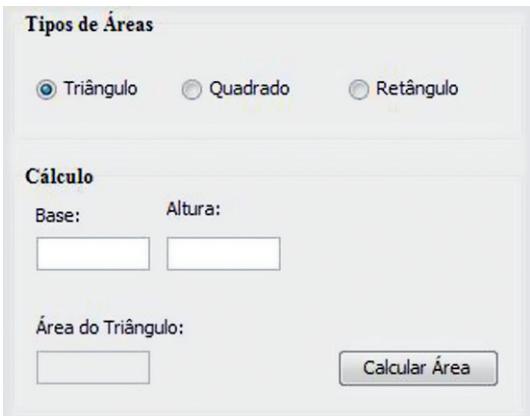
**Figura 1.27: Tela do exercício**

Fonte: Elaborada pelo autor Ygor Morato

A próxima seção apresenta a continuação da interface de nosso programa. Por isso, realize o Exercício 2 antes de continuar a leitura do material. Para implantar nosso programa no *NetBeans*, basta executarmos o projeto no qual o estamos criando. Para isso, clique com o botão direito sobre o nome do projeto na aba “**Projetos**” (no canto esquerdo do *NetBeans*), e depois em “**Executar**”. Se preferir, basta clicar a tecla “**F6**”. Ao executá-lo, o *NetBeans* pedirá para escolhermos a classe principal a ser executada (mais adiante explicaremos o que é uma classe principal). Por enquanto escolha a classe “**pacoteInterface.Interface**” que é a única opção da lista.

## 1.6 Adicionando funcionalidades ao programa

Nas últimas seções começamos a desenvolver a interface gráfica de nosso primeiro programa em Java. Entretanto, o programa ainda não possui funcionalidades. Para isso, vamos incluir um botão para que sejam calculadas as áreas. Um botão é inserido por meio do item “**Botão**” da opção “**Controles Swing**” na aba “**Paleta**” (no mesmo local onde encontramos o “**Rótulo**” e o “**Campo de texto**” anteriormente). Incluímos então um botão ao lado do campo de texto com a área e, em seguida, editamos o texto do botão (de forma que seja exibida a frase “**Calcular Área**”). Temos que ressaltar que nosso programa irá calcular três áreas diferentes. Veja na Figura 1.28 o botão inserido em nossa interface.



**Figura 1.28: Tornando um campo de texto não editável**

Fonte: Elaborada pelo autor Ygor Morato

Além disso, fizemos com que os campos de texto com o resultado da área deixassem de estar disponíveis para ser editados pelo usuário. Para isso, clicamos com o botão direito em cada um dos campos e escolhemos a opção “**Propriedades**”. Em seguida, desmarcamos a opção “**editable**” (que faz com que eles não sejam editáveis). Agora vamos criar o código que irá calcular a área do triângulo. Para isso iremos verificar se o “`jRadioButton1`”, que é a opção de triângulo, está marcado. Se estiver, vamos efetuar o cálculo que é:

$$A = (B * H)/2$$

Área (A) é igual à base (B) vezes altura (H), dividido por 2.

Para isso, clicamos com o botão direito do *mouse* sobre o botão que acabamos de criar, depois em “**Eventos**”, depois em “**Action**” (que significa “ação” em português) e depois na única opção disponível (**actionPerformed [jButton1ActionPerformed]**). Repare que agora o *NetBeans* está exibindo o código-fonte da interface de nosso programa.

Na verdade o código-fonte já podia ser exibido desde quando criamos nosso *frame*. Para isso, bastava alternarmos entre os botões “**Código-Fonte**” e “**Projeto**” acima do código (mostrados no topo das telas). O código-fonte exibido pelo *NetBeans* contém vários detalhes da linguagem Java que ainda não foram explicados neste material. Não se preocupe com isso agora. Mais adiante explicaremos todos esses detalhes.

Por enquanto, nossas atenções devem ser voltadas ao trecho de código mostrado a seguir:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
}
```

**Figura 1.29: Trecho de código**

Fonte: Elaborada pelo autor Hilário Júnior

A seguir mostramos o trecho de código necessário para efetuarmos o cálculo da área do triângulo e, em seguida, explicamos o que o código faz:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    double base = 0, altura =0;  
    double result = 0;  
    //verificamos se o campo JRadioButton1 esta selecionado  
    if (jRadioButton1.isSelected() == true) {  
  
        //Transformando os campos digitados para float  
        base = Double.parseDouble(this.jTextField1.getText());  
        altura = Double.parseDouble(this.jTextField2.getText());  
  
        //Calculando a área  
        result = (base * altura) / 2;  
    }  
    //Atualizando o campo de texto com o resultado da área  
    this.jTextField3.setText(Double.toString(result));  
}
```

**Figura 1.30: Trecho de código**

Fonte: Elaborada pelo autor Hilário Júnior

Como podemos observar no trecho de código da Figura 1.30, Java possui o tipo “*double*”, assim como C, que é utilizado para representar um número real com precisão dupla. Entretanto, os campos de texto digitados pelo usuário são tratados sempre como *string*. Para calcularmos a área, portanto, precisamos encontrar os valores digitados e transformá-los em *double*.

O método “**this.JTextField1.getText()**”, por exemplo, recupera o texto digitado no campo “**jTextField**” (que corresponde à base do triângulo). O método “**Double.parseDouble**” foi utilizado, então, para transformar o texto em *double*. Por fim, “**Double.toString(media)**” transforma a média final (que é um *double*) em *String*, para que ela seja definida como o texto do campo com a área através do método “**this.JTextField3.setText(Double.toString(result))**”.

Agora desenvolva seu programa com calma, execute-o e verifique se o cálculo está sendo feito corretamente.

Não se preocupe muito com aspectos da linguagem Java que você não conhece, pois eles serão explicados em detalhes a partir da próxima aula. No mais, seja bem-vindo ao mundo dos programadores Java, que é a linguagem mais utilizada no mercado de trabalho atualmente!

Agora é necessário que você faça seu programa calcular as demais áreas e isso fica como desafio!

Se precisar acessar os arquivos de seu projeto, eles ficam armazenados dentro da pasta “NetBeans Projects”, que fica na pasta “Meus Documentos”.

Antes de nos preocuparmos com os conceitos de orientação a objetos e com a criação de interfaces gráficas, vamos focar na criação de programas em Java semelhantes aos que criávamos em C. Veja a seguir um exemplo simples de código na linguagem Java.

```
public class Um {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main (String[] args) {  
        System.out.print ("Programa UM!!!");  
        // TODO code application logic here  
    }  
}
```

**Figura 1.31: Trecho de código**

Fonte: Elaborada pelo autor Hilário Júnior

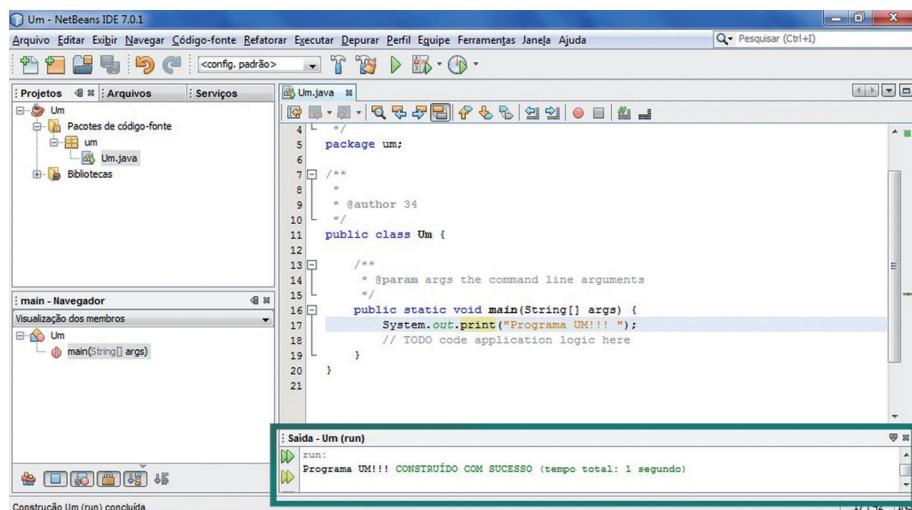
Todo programa em Java inicia-se com a definição de uma classe. Uma classe é definida através da palavra reservada **class**, seguida pelo nome da classe (nesse caso, o nome da classe é “Um”). Todo nome de classe deve iniciar com uma letra maiúscula. Uma observação importante é que a linguagem Java é *case sensitive* (sensível ao caso), ou seja, o compilador diferencia letras maiúsculas de minúsculas. A classe principal (aquele que será executada inicialmente pelo programa) deve possuir um método chamado **main**, assim como em C, que é invocado quando a classe é executada.

Para compilar e executar o programa, primeiro é preciso salvá-lo com o nome “**Um.java**”. Você pode fazê-lo até mesmo em um editor de texto simples, mas continuaremos a mostrar nossos exemplos no *NetBeans*.

Para não haver confusão com códigos anteriores, crie um novo projeto no *NetBeans*. Deixe marcada a opção “Criar classe principal” (defina o nome da classe principal como “**Um**”) e “Definir como projeto principal”. A classe “Um” criada já irá possuir um método “**main**”. Neste caso, apenas insira no método o comando “**System.out.println (“Programa UM!!!!!”)**”.

Em Java, o nome do arquivo deve ser sempre igual ao nome da classe, seguida da extensão “.java”. No caso do nosso exemplo, o nome do arquivo precisa ser “Um.java” porque o nome da classe é “Um”.

Em seguida, execute o projeto. Como não criamos uma interface para o programa, a saída será exibida na aba “Saída”, localizada a seguir aos códigos fontes no *NetBeans*. Veja em destaque na Figura 1.32 essa região com a saída do programa.



**Figura 1.32: Saída de um programa Java no NetBeans**

Fonte: Elaborada pelo autor Hilário Júnior

Se você for executar o programa fora do *NetBeans* é necessário abrir o *prompt* na pasta em que salvou o arquivo e digitar os seguintes comandos:

```
javac Exemplo01.java  
Java Exemplo01
```

O primeiro comando gera o *bytecode* do programa, chamado “*Um.class*”. O segundo executa o programa a partir do *bytecode* gerado no primeiro comando. Se for necessário incluir mais de uma classe em um mesmo arquivo, só uma delas poderá ser *public*. Esta será a classe principal do arquivo. Caso nenhuma classe seja *public*, o compilador entenderá que a classe principal é aquela com o mesmo nome do arquivo.

## 1.7 Tipos de dados e definição de variáveis

O Quadro 1.1 apresenta os principais tipos de dados disponíveis na linguagem Java.

Quadro 1.1 Tipos de dados primitivos em Java		
Tipos de Dados	Definição	Tipo
Caractere	Letras, números, símbolos.	char
Inteiro	Números inteiros positivos ou negativos	int
Real	Números com casas decimais.	float
Lógico	Verdadeiro(1) ou Falso(0)	Boolean

Fonte: Elaborado pelo autor Gustavo Menezes

Para uma variável ser utilizada, ela precisa ser declarada. O código abaixo fornece exemplos de manipulação de variáveis em Java:

```
public Class Dois {
    public static void main(string args[]) {
        int n1; /* Declaração de um inteiro. */
        int n2 = 10; /* Declaração e inicialização de outro inteiro. */
        char c = 'x'; /* caractere */
        n1 = n2 +8; // Atribuição.
        System.out.println("Primeiro valor: "+ n1);
        System.out.println("Segundo valor: "+ n2);
        System.out.println("Terceiro valor: "+ c);
    }
}
```

**Figura 1.33: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Um comentário em Java pode ser escrito com // (para comentar apenas até o final da linha) ou com /\* \*/ (para comentar tudo o que estiver entre o /\* e o \*/). Para identificar a documentação, utilizamos /\*\* \*/

A saída desse programa será:

Primeiro valor: 18

Segundo valor: 10

Terceiro valor: T

Em Java existem duas formas de convertermos valores de um tipo para outro:

a) **conversão implícita**, na qual os dados são convertidos automaticamente, sem a preocupação do programador. Ela ocorre, por exemplo, quando convertemos um número inteiro para um número real. Nesse caso a conversão é implícita porque é óbvio para o compilador que um número inteiro pode ser representado também como um número real. Veja um exemplo a seguir:

```
int t = 7;
float y = t; //convertendo inteiro para float
double z = y; //convertendo float para double
int t = 7;
float y = t; // Convertendo inteiro para float.
double z = y; //Convertendo float para double.
```

**Figura 1.34: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

b) **conversão explícita**, quando o programador precisa explicitar no código que um valor será convertido de um tipo para outro. No caso de um número real para um inteiro, por exemplo, pode haver perda na precisão do número (por isso é necessário que o usuário deixe explícito que ele quer realizar a conversão). Veja um exemplo na Figura 1.35.

```
float a = 9;
float b = a/8; //b = 1.125
int c = (int)b; /* Aqui estamos forçando a conversão para
um numero inteiro. Neste caso, a variável c armazenará
apenas a parte inteira da variável b, ou seja, 1 */
System.out.println(b);
System.out.println(c);
```

**Figura 1.35: Exemplo de código**

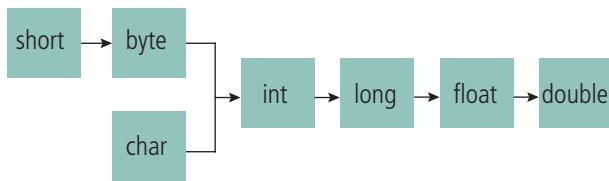
Fonte: Elaborada pelo autor Hilário Júnior

O resultado da execução do trecho de código mostrado na Figura 1.35 é:

1.125

 O tipo *boolean* não pode ser convertido para nenhum outro tipo.

Seguindo o sentido das setas da Figura 1.36, vemos os tipos que podem ser implicitamente convertidos em outros. Seguindo o sentido contrário, vemos os tipos que precisam ser convertidos explicitamente:



**Figura 1.36: Conversões possíveis entre tipos primitivos em Java**

Fonte: Elaborada pelo autor Hilário Júnior

## 1.8 Strings

O termo **string** representa uma sequência de caracteres. *String* em Java é uma classe que oferece diversos métodos para manipular sequências de caracteres. A Figura 1.37 ilustra a utilização do método de concatenar com o operador “+”.

```

String nome = "Gustavo";
String sobrenome = "Menezes";
System.out.print(nome + sobrenome);
System.out.print(nome +" "+sobrenome);

```

**Figura 1.37: Concatenar com o operador “+”**

Fonte: Elaborada pelo autor Ygor Morato

A saída do trecho de código anterior é:

Gustavo Menezes

Gustavo Menezes

Para compararmos se o valor de duas *strings* são iguais, utilizamos o método **equals** (o operador “==” não compara *Strings*). Veja no exemplo a seguir:

```

if ( nome.equalsIgnoreCase(sobrenome) ) {
    System.out.println("Nome e Sobrenome são iguais!");
}

```

**Figura 1.38: Exemplo de código**

Fonte: Elaborada pelo autor Ygor Morato

O trecho de código apresentado na Figura 1.39 mostra outros métodos úteis da classe *String*.

```

// O método length retorna o tamanho da String
System.out.println("Tamanho da String: " + nome.length());
// O método substring retorna um pedaço da String
System.out.println("Substring: " + nome.substring(0, 5));
// O método charAt retorna o caracter em dado índice da String
System.out.println("Caracter na posição 5: " + nome.charAt(5));
// O método split quebra a String em várias outras,
// pelo separador desejado */
String frase = "Isto é uma frase";
String palavras[] = frase.split(" ");
System.out.println(frase.indexOf("uma"));
// Replace troca todas as ocorrências de um caracter por outro
System.out.println("Paulo".replace('o', 'a'));

```

**Figura 1.39: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

A Figura 1.40 mostra a saída do trecho de código da Figura 1.39.

```

Tamanho da String: 7
SubString: Hilar
Caracter na posição 5: i
7
Paula

```

**Figura 1.40: Saída de código**

Fonte: Elaborada pelo autor Hilário Júnior

## 1.9 Arrays

Arrays (vetores) em Java podem conter qualquer tipo de elemento valorado (tipos primitivos ou objetos), mas você não pode armazenar diferentes tipos em um simples array (por exemplo, você pode criar um array de inteiros, ou um array de strings, ou um array de array, mas você não pode criar um array que contenha ambos os objetos strings e inteiros). A seguir algumas declarações possíveis de arrays:

```

String palavra[]; // Array contendo Strings
int numeros[]; // Array contendo inteiros

```

Uma das formas de criarmos um objeto array é usando o operador new, que cria uma nova instância de um array, por exemplo:

```
int numeros[] = new int[30];
```

Neste caso, será criado um array de 30 inteiros (inicializados com o valor 0). Também é possível criarmos um array ao mesmo tempo em que o declaramos.

```
String frutas[] = { "tomate", "abacaxi", "manga" }
```

Uma vez que o *array* já foi devidamente criado, podemos acessar e/ou alterar o valor dos elementos em cada um de seus índices. Os *arrays* em Java sempre iniciam com elementos na posição 0. Por exemplo:

```
int numeros[] = new int[2];
numeros[0] = 10;
numeros[1] = 1;
numeros[2] = 5; // Esta linha gera um erro em tempo de execução.
```

A última linha do trecho de código acima causa um erro de compilação, pois o índice 2 não existe em um *array* com apenas dois elementos. *Arrays* podem ter mais de uma dimensão. Um *array* bidimensional, por exemplo, é um *array* de *arrays*. Podemos simular uma matriz de números, por exemplo, usando *arrays* bidimensionais. Veja um exemplo a seguir:

```
int matriz[][] = new int[10][10];
matriz[0][0] = 57;
matriz[0][1] = -3;
```

Para sabermos o tamanho de um *array*, podemos utilizar o atributo “*length*”, assim como em uma *string*. A partir do momento em que um *array* foi criado, ele não pode mudar de tamanho. Se você precisar de mais espaço, será necessário criar um novo *array* e, antes de se referir a ele, copiar os elementos do *array* antigo.

## 1.10 Operadores

Os operadores aritméticos para manipularmos as variáveis em Java são os mesmos da linguagem C. Utilizamos “+” para somar dois valores, “-” para subtrair, “\*” para multiplicar, “/” para dividir, “%” para calcular o resto da divisão inteira, “++” para incrementar um valor e “--” para decrementar. Java também possui vários operadores para testar igualdade e magnitude. Utilizamos “==” para verificar se dois valores são iguais, “!=” para verificar se são diferentes, “<” para verificar se o primeiro é menor que o segundo”, “>” para verificar se ele é maior, “<=” para menor e igual, e “>=” para maior ou igual.

Para combinarmos valores ou expressões lógicas utilizamos “&&” (operação E), “||” (operação OU) e “!” para negar um valor.

## 1.11 Funções matemáticas

Na classe *Math* existe uma série de métodos estáticos que fazem operações com números como, por exemplo, arredondar (*round*), obter o valor absoluto (*abs*), obter a raiz (*sqrt*), calcular o seno (*sin*) e outros.

```
double d = 4.6;
long i= Math.round(d);
int x = -4;
int y = Math.abs(x);
double area = Math.PI * d^2;
```

**Figura 1.41: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

A documentação do Java apresenta outros métodos que podem ser úteis, como números aleatórios, tangentes, logaritmo, etc.

## 1.12 Controle de fluxo

Comandos condicionais em Java são semelhantes aos de C. O trecho de código a seguir, por exemplo, verifica se um estudante foi aprovado.

```
int x = 60;
if ( x < 60 ) {
    System.out.println("estudante Reprovado")
}
else {
    System.out.println("estudante Aprovado")
}
```

Não é necessário que todo *IF* seja acompanhado de um *ELSE*, mas todo *ELSE* só pode existir após um *IF*. Podemos também concatenar expressões booleanas com os operadores lógicos “E” e “OU”. O primeiro é representado por “*&&*”, e o segundo por “*| |*”. No exemplo a seguir o programa verifica se um estudante pode fazer recuperação ou se passou direto de ano. Para fazer recuperação, a nota deve ser superior a 39 e inferior a 60 pontos.

**Obs.:** Estamos considerando aqui que as notas são sempre valores inteiros.

```

if ( x > 39 && < 60) {
    System.out.println(" Estudante selecionado para recuperação ");
}
else {
    System.out.println(" Estudante aprovado ");
}

```

A estrutura *SWITCH-CASE* equivale a um conjunto de cláusulas *IF* encadeadas, deixando o código mais legível e eficiente no caso de grandes desvios condicionais. Veja o exemplo na Figura 1.43.

```

switch (x) {
    case 0: System.out.println("zero"); break;
    case 1: System.out.println("um"); break;
    case 2: System.out.println("dois"); break;
    case 3: System.out.println("tres"); break;
    case 4: System.out.println("quatro"); break;
    case 5: System.out.println("cinco"); break;
    case 6: System.out.println("seis"); break;
    case 7: System.out.println("sete"); break;

    case 8: System.out.println("oito"); break;
    case 9: System.out.println("nove"); break;
    default: System.out.println("Número desconhecido");
}

```

**Figura 1.42: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

## 1.13 Escopo das variáveis

Em Java podemos declarar variáveis a qualquer momento. Entretanto, o trecho de código em que a variável será válida dependerá de onde ela foi declarada. **Escopo da variável** é o nome dado ao trecho de código em que a variável existe e no qual é possível acessá-la. Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro só existirão até o fim daquele bloco.

```

// Aqui a variável x ainda não existe
int x = 1;
// Aqui a variável x já existe
while ( <condição> ) {
    // Aqui a variável x continua existindo
    int y = 0;
    // Aqui a variável y já existe
}
// Aqui a variável j não existe mais. O x continua
existindo

```

**Figura 1.43: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Também é possível declararmos um bloco dentro de outro. Podemos inserir um *IF* dentro de um laço, um laço dentro de um *IF*, um laço dentro de outro e assim sucessivamente. Veja um exemplo na Figura 1.45.

```
for ( int x = 1; x < 100; x++ ) {  
    if ( par(x) ) {  
        System.out.println(x);  
    }  
}
```

**Figura 1.44: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

## Resumo

Nesta aula estudamos os principais conceitos relacionados à linguagem Java. Aprendemos como fazer o *download* e configurar o ambiente de desenvolvimento *NetBeans*. Além disso, vimos alguns dos principais comandos do Java.

## Atividades de aprendizagem

1. Crie um novo programa que armazene um vetor *float* com as notas de 10 estudantes e imprima a média dessas notas.
2. Crie no mesmo programa da questão anterior um *array* de *strings* com o nome dos meses do ano. Crie uma variável inteira com um valor entre 1 e 12 e imprima o nome do mês correspondente ao valor da variável.
3. Crie um número inteiro de três dígitos. Utilize os operadores "/" e "%" para obter o número com os algarismos na ordem inversa.

Exemplo: "x = 356 e y = 653"

4. Crie uma variável inteira contendo um número de segundos e imprima o número equivalente de horas, minutos e segundos.
5. Avalie o resultado da expressão a seguir:

$((3<4) \&\& ((5<=7 \parallel 8==10 \parallel 9>3) \&\& !(4<5)) \parallel (2>0))>$ .

**6.** O que faz o trecho de código a seguir?

```
If (a > b) {  
    aux = a;  
    a = b;  
    b = a;  
}
```

```
If (a > c) {  
    aux = a;  
    a = c;  
    c = aux;  
}
```

```
If (b > c) {  
    aux = b;  
    b = c;  
    c = aux;  
}
```

**7.** Crie três variáveis reais, defina valores quaisquer para elas e defina um código que verifique se elas mesmas podem ser lados de um triângulo (ou seja, nenhuma pode ser maior que a soma das outras duas).

**8.** Crie uma variável contendo a idade de uma pessoa e verifique sua condição eleitoral:

- até 16 anos não pode votar;
- entre 16 e 18 anos ou mais que 65 é facultativo;
- entre 18 e 65 anos é obrigatório.

**9.** Crie uma variável inteira contendo um dia do ano e outra contendo um mês. Verifique se elas formam uma data válida (considere que fevereiro sempre possui 28 dias).

**10.** Verifique se um número inteiro qualquer corresponde a um ano bissexto (anos bissextos são os múltiplos de 4 que não são múltiplos de 100) e ainda os múltiplos de 400. Exemplo: 1996 e 2000 são bissextos, enquanto 1998 e 1900 são comuns.

**11.** Crie dois números inteiros e imprima:

- a)** a soma dos dois, caso ambos sejam ímpares;
- b)** o produto, caso sejam pares;
- c)** o número ímpar, caso um seja par e outro não.

**12.** Escreva um programa que verifique se uma nota é péssima (nota=1), ruim (2), regular (3), boa (4), ótima (5) ou nenhuma delas (nota inválida). Utilizamos um *WHILE* para criarmos um **laço (loop)**, ou seja, repetir um trecho de código algumas vezes enquanto uma determinada condição for verdadeira. O exemplo abaixo imprime os cinco primeiros múltiplos de 9:

```
int x = 1;
while (x <=5) {
    System.out.println(9*5);
    X++;}
```

**13.** O trecho de código dentro do *WHILE* será executado enquanto a condição ( $x \leq 5$ ) for verdadeira. Isso deixará de acontecer no momento em que ( $x > 5$ ). O comando *FOR* também é utilizado para criarmos *loops*. A ideia é a mesma que a do *WHILE*, mas existe um espaço próprio para inicializar e modificar a variável de controle do laço, deixando-o mais legível. O exemplo abaixo gera o mesmo resultado do *WHILE* acima.

```
for (int x = 1; x <=5; x++) {
    System.out.println(9*5);
}
```

**14.**O *FOR* e o *WHILE* podem ser utilizados para o mesmo propósito. Porém, o código do *FOR* indica claramente que a variável **i** serve, em especial, para controlar a quantidade de laços executados. Use cada um quando achar mais conveniente. O *FOR* também é muito útil para percorrermos um *array*. Para isso, basta usarmos o atributo “*length*”, que retorna o tamanho do *array*. Veja um exemplo:

```
for (int x = 0; x <= nome_do_array.length; x++)  
    System.out.println(nome_do_array[i]);
```

**Use WHILE ou FOR nos exercícios a seguir:**

**15.**Imprima todos os números de 100 a 300.

**16.**Imprima a soma de todos os números de 1 a 999.

**17.**Crie um número inteiro e verifique se ele é primo.

**18.**Qual o valor de *n* no final do código a seguir?

```
int n =7;  
for (int x = 1; x <=5; x++) {  
    n = n * x;  
}
```

**19.**Calcule e imprima o fatorial de um número inteiro qualquer.

**20.**Calcule e imprima o fatorial dos números de 1 a 15. Crie um *FOR* que comece imprimindo o fatorial de 1, e a cada passo utilize o último resultado para o cálculo do fatorial seguinte.

**21.**Calcule e imprima os 30 primeiros elementos da série de Fibonacci. A série é a seguinte: (0, 1, 1, 2, 3, 5, 8, 13, 21, ...). Para calculá-la, o primeiro elemento vale “zero” e o segundo elemento vale 1; daí por diante, cada elemento vale a soma dos dois elementos anteriores (ex.: 8 = 5 + 3).

Registre suas respostas num arquivo digital e poste-o no ambiente virtual de ensino-aprendizagem (AVEA) do curso.



# Aula 2 – Orientação a objetos: classes, objetos e métodos

## Objetivos

Compreender o que é orientação a objetos e o que é e como criamos uma classe e um objeto em Java.

## 2.1 Introdução à orientação a objetos

Para entendermos melhor o que vem a ser orientação a objetos, precisamos compreender os seguintes itens:

- a) qualquer coisa é um objeto;
- b) objetos realizam tarefas através da requisição de serviços a outros objetos;
- c) cada objeto pertence a uma determinada *classe*. Uma classe agrupa objetos similares;
- d) classes são organizadas em hierarquias.

A principal dificuldade neste paradigma é conseguir diferenciar o que é **classe** e o que é **objeto**. Nas próximas seções desta e das próximas aulas mostraremos algumas das principais terminologias utilizadas na programação orientada a objetos (POO) que serão utilizadas em todo este material, como **classes, objetos, herança, métodos, polimorfismo e encapsulamento**.

## 2.2 Classe e objeto

**Classe**: é um molde para objetos. Diz-se que um objeto é uma instância de uma classe. É uma *abstração* das características importantes de um grupo de coisas do mundo real. Classe possui membros (atributos e métodos).

**Objeto**: instância de uma classe. Composto por:

- a) **atributos**: memória onde valores podem ser armazenados e modificados ao longo da vida do objeto;

- b) comportamento:** conjunto de ações predefinidas (métodos) através das quais o objeto responderá às mensagens e desempenhará papéis (funções).

## OBJETO = DADOS + PROCESSOS

Um objeto é criado por alguém, tem um tempo de duração e se desgasta, quebra, fica inutilizado ou é destruído por alguém.

### Objeto como abstrações:

Dependendo do contexto, um mesmo conceito do mundo real pode ser representado por diferentes abstrações:

- a)** Carro (para uma transportadora de cargas);
- b)** Carro (para uma fábrica de automóveis);
- c)** Carro (para um colecionador);
- d)** Carro (para uma empresa de *kart*);
- e)** Carro (para um mecânico).

**Objetos** são abstrações de entidades que existem no mundo real.

**Classes** são definições estáticas que possibilitam o entendimento de um grupo de objetos.

## 2.3 Classes e objetos em Java

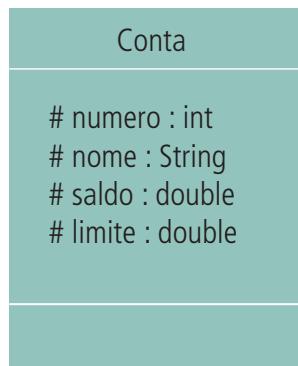
Para exemplificar a implementação de classes em Java, vamos criar uma classe que armazena os dados da conta de um usuário no banco. Por enquanto vamos escrever apenas o que a conta possui (atributos) e não o que ela faz (métodos).

```
class Conta {  
    int numero;  
    String nome;  
    double saldo;  
    double limite;  
}
```

**Figura 2.1: Exemplo de classe**

Fonte: Elaborada pelo autor Hilário Júnior

Esses são os **atributos** que toda conta, quando criada, irá possuir. Repare que essas variáveis foram declaradas fora de um bloco, diferentemente do que fizemos na classe que possuía um método **main**. Quando uma variável é declarada diretamente dentro do escopo da classe, ela é chamada de variável de objeto ou **atributo**. A Figura 2.2 mostra o que temos em nossa classe até agora.



**Figura 2.2: Atributos da classe Conta**

Fonte: Elaborada pelo autor Hilário Júnior

A classe **Conta** especifica o que todos os objetos dessa classe devem conter. Nesse caso ela não possuirá um método chamado **main** porque ela não será a classe principal do nosso programa. Para isso criaremos uma classe **Programa** que criará um objeto da classe **Conta**.

```
class Programa {  
    public static void main(String[] args) {  
        Conta minhaConta; /* Declarando a variável minhaConta,  
        que conterá um objeto da classe Conta */  
        minhaConta = new Conta(); /* Criando efetivamente um  
        objeto da classe Conta e atribuindo-o à variável minha-  
        Conta */  
    }  
}
```

**Figura 2.3: Exemplo de utilização da classe Conta**

Fonte: Elaborada pelo autor Hilário Júnior

```
minhaConta.nome = "Hilário"; /* Atribuindo o valor de uma  
String à variável nome do objeto minhaConta */  
minhaConta.saldo = 1000.0; /* Atribuindo o valor de um  
double à variável saldo do objeto minhaConta */  
System.out.println("Saldo atual: "+  
                    minhaConta.saldo);  
}  
}
```

**Figura 2.4: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Preste atenção no **ponto** que foi utilizado em “minhaConta.nome” e “minhaConta.saldo”. É assim que acessamos uma variável de objeto. Agora, sempre que o programa for iniciado, a conta pertencerá ao usuário “Hilário” e seu saldo será de R\$ 1.000,00.

Para criar um objeto a partir de uma classe, basta utilizar o operador *new*, conforme exemplificado na Figura 2.3.

## 2.4 Métodos

As operações que um objeto oferece são chamadas de **métodos**. **Andar**, **correr e comer**, por exemplo, podem ser métodos da classe **Pessoa**. **Métodos** são comportamentos que os objetos de uma classe podem possuir. A operação de ligar um objeto da classe **Moto**, por exemplo, pode ser feita com o método **ligar()**.

No código apresentado na Figura 2.5, a classe Conta será alterada para inclusão do método “sacar”.

```
class Conta {  
    int numero;  
    /* Deixei aqui os demais atributos que já criamos para a  
    classe Conta */  
  
    void sacar (double quantidade) {  
        this.saldo = this.saldo - quantidade; /* É o mesmo que  
        fazer this.saldo = quantidade */  
    }  
}
```

**Figura 2.5: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

O método “sacar” recebe como argumento (ou parâmetro) um valor do tipo “*double*” contendo a quantidade de dinheiro a ser sacada da conta do usuário. Sem o argumento é impossível sabermos quanto devemos subtrair do saldo da conta. Essa variável chamada “quantidade” pode ser acessada durante todo o método e deixa de existir ao final dele. A palavra “*void*” indica que nenhum valor será retornado por esse método.

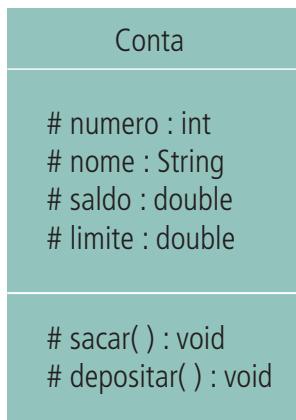
A palavra “*this*” permite que acessemos o valor de um atributo da própria classe em questão, no nosso caso a classe **Conta**. Na Figura 2.6 é mostrada a implementação de outro método para a classe Conta.

```
class Conta {  
    // Deixe aqui os atributos e métodos anteriores  
  
    void depositar (double quantidade) {  
        this.saldo += quantidade;  
    }  
}
```

**Figura 2.6: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

O “`+=`” soma o valor de “`quantidade`” ao valor antigo do “`saldo`” e guarda o resultado no próprio “`saldo`”. Para fazer com que um objeto execute algum método, também colocamos um “ponto”.



**Figura 2.7: Atributos e métodos da classe Conta**

Fonte: Elaborada pelo autor Hilário Júnior

A Figura 2.7 mostra o diagrama da classe Conta com os métodos criados anteriormente.

No código apresentado na Figura 2.8, alteramos a classe “Programa” para sacarmos dinheiro da conta e depois depositarmos.

```

class Programa {
    public static void main (String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();

        minhaConta.nome = "Hilário";
        minhaConta.saldo = 1000.0;

        minhaConta.sacar(200); // Saca 200 reais
        minhaConta.depositar(500); // Deposita 200 reais

        System.out.println("Saldo atual:" +
                           minhaConta.saldo);
    }
}

```

**Figura 2.8: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Em todo método devemos especificar o tipo de retorno. Nos casos anteriores, utilizamos a palavra “*void*”, que significa que não há um valor de retorno do método.

Quando um método possui um valor de retorno, esse valor é devolvido para o código que o chamou. O método “*sacar*”, por exemplo, poderia retornar um valor booleano indicando se a operação foi bem-sucedida (o que acontece se a quantidade a ser sacada não for maior que o valor do saldo mais o limite da conta). Veja o exemplo apresentado na Figura 2.9.

```

boolean sacar (double quantidade) {
    if (quantidade > (this.saldo + this.limite)) {
        return false;
    }
    else {
        this.saldo = this.saldo - quantidade;
        return true;
    }
}

```

**Figura 2.9: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

A Figura 2.10 ilustra a utilização do valor de retorno do método “sacar”.

```
minhaConta.saldo = 1000.0;
boolean consegui = minhaConta.sacar(2000);

if (consegui) {
    System.out.println("Operação realizada com sucesso.");
}
else {
    System.out.println("ERRO: Saldo insuficiente.");
}
```

**Figura 2.10: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Para simplificar o código, podemos obter o mesmo resultado da forma apresentada na Figura 2.11.

```
minhaConta.saldo = 1000.0;
boolean consegui = minhaConta.sacar(2000);

if (consegui) {
    System.out.println("Operação realizada com sucesso.");
}
else {
    System.out.println("ERRO: Saldo insuficiente.");
}
```

**Figura 2.11: Exemplo de código**

Fonte: d Elaborada pelo autor Hilário Júnior

## Resumo

Nesta aula começamos a compreender o que é orientação a objetos. Aprendemos o que é uma classe e um objeto. Além disso, começamos a construir classes em Java com atributos e métodos.

## Atividades de aprendizagem

1. Implemente a classe **Pessoa**, contendo os atributos “nome”, “altura” e “peso”, pelo menos. Crie alguns objetos dessa classe e defina valores diferentes para os atributos de cada um deles.
2. Crie 10 objetos da classe **Pessoa** e insira-os em um *array* de objetos dessa classe. Utilize um *for* para imprimir todos os atributos dos objetos do *array*.
3. Crie uma classe para um “**Funcionario**”. Ela deve ter o nome do funcionário, o departamento onde trabalha, o salário dele (*double*), a data de entrada no banco (*string*), o RG dele (*string*) e um valor booleano que indique se o funcionário está na empresa no momento ou se já foi embora.
4. Crie uma classe **Empresa** que possua “nome”, “cnpj”, “qtde\_de\_funcionario” e um *array* de objetos da classe **Funcionario** (o *array* pode armazenar até 100 funcionários).
5. Crie os seguintes métodos para a classe **Funcionario**, criada anteriormente:
  - a) método “bonificar”, que aumenta o salário do funcionário de acordo com o parâmetro passado como argumento;
  - b) método “demitir”, que não recebe parâmetro algum, apenas modifica o valor booleano indicando que o funcionário não trabalha mais na empresa;
  - c) método “mostrarDados”, que simplesmente imprime todos os atributos de um funcionário.
6. Crie dois objetos da classe **Funcionario** com os mesmos atributos. Compare os dois com o operador “*=*” e veja se o resultado é *true* ou *false*.
7. Insira na classe **Pessoa**, também criada anteriormente, o atributo “idade” e o método “aniversario”, que incrementa sua “idade”. Crie um objeto desta classe, defina a “idade”, chame o método “aniversario” algumas vezes e depois imprima novamente a “idade”. Poste todas as atividades nos fóruns específicos do curso.

Registre suas respostas num arquivo digital e poste-o no ambiente virtual de ensino-aprendizagem (AVEA) do curso.

# Aula 3 – Orientação a objetos: métodos e encapsulamento

## Objetivo

Desenvolver a habilidade de encapsular os dados de uma classe e de criar métodos e construtores que manipulem esses dados.

Antes de continuarmos o nosso estudo de métodos e encapsulamento, vamos fazer uma breve revisão de alguns conceitos da aula anterior:

- a) **Classe:** é um tipo de objeto que pode ser definido pelo programador para descrever uma entidade real ou abstrata.
- b) **Objeto:** é uma instância de uma classe. Por exemplo, vamos imaginar uma classe chamada **Moto**. Cada moto existente é uma instância (ou um objeto) dessa classe. Apesar de parecidas, cada moto possui características próprias, como cor, chassi, arranhões etc.
- c) **Métodos:** são comportamentos que os objetos de uma classe podem possuir. Eles definem as operações que podemos realizar com os objetos de uma classe.

## 3.1 Construtores

Outro conceito importante sobre orientação a objetos é sobre métodos construtores e destrutores. Quando usamos a palavra “*new*”, como fizemos no código da classe “Programa”, estamos construindo um novo objeto. Sempre que isso é feito, o **construtor** da classe é executado. O construtor da classe é um bloco declarado com o mesmo nome que a classe. Veja um exemplo no código apresentado na Figura 3.1.

```
class Conta {  
    int numero;  
    String nome;  
    double saldo;  
    double limite;  
  
    Conta () { //Aqui estamos declarando o contrutor  
        System.out.println("Nova conta criada.");  
    }  
}
```

**Figura 3.1: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

No caso do exemplo mostrado na Figura 3.1, sempre que um novo objeto da classe Conta for criado, a mensagem “Nova conta criada” aparecerá na tela. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. O construtor pode receber parâmetros assim como um método. Para aprimorar o construtor da classe Conta, por exemplo, poderíamos receber o valor do “nome” do titular da conta como parâmetro.

Veja no trecho de código a seguir:

```
Conta (String nome) {  
  
    this.nome = nome;  
  
}
```

Agora não precisamos mais atribuir os valores a esses parâmetros após instanciarmos o objeto “minhaConta” da classe “Programa”. Ao invés disso, passamos os valores como parâmetro para o construtor da classe Conta ao criarmos o objeto. Veja a nova classe “Programa” no código apresentado na Figura 3.2.

```
class Programa {  
    public static void main (String[] args) {  
        Conta minhaConta;  
        minhaConta = new Conta("Hilário");  
  
        minhaConta.saldo = 1000.0;  
        /* Deixe aqui as demais operações que foram inseridas  
no programa */  
    }  
}
```

**Figura 3.2: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Esse exemplo é importante, pois ilustra uma das principais vantagens do construtor, já que toda conta precisa obrigatoriamente de um titular; basta criar um construtor que receba como parâmetro uma *string* com o nome do titular.

O construtor se resume a isso: dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação deste.

Também é possível criar mais de um construtor para uma mesma classe. Para isso basta que cada construtor tenha um número diferente de parâmetros. Observe o código da Figura 3.3.

```

class Conta {
    int numero;
    String nome;
    double saldo;
    double limite;

    Conta (String nome) {
        this.nome = nome;
    }

    Conta (String nome, double saldo) {
        this.nome = nome;
        this.saldo = saldo;
    }
}

```

### Figura 3.3: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

A seguir temos o exemplo de um trecho de código que cria dois objetos para a classe **Conta**. O primeiro passa uma *string* e um *float* como parâmetros na criação do objeto, utilizando o segundo método construtor definido na classe mostrada na Figura 3.3. O segundo passa apenas uma *string* como parâmetro, utilizando o primeiro método construtor definido na classe:

```

Conta minhaConta = new Conta("Hilário", 1000.0);
Conta outraConta = new Conta("Joãozinho");
System.out.print("Usuário: " + minhaConta.nome);
System.out.println(" / Saldo: " + minhaConta.saldo);
System.out.print("Usuário: " + outraConta.nome);
System.out.println(" / Saldo: " + outraConta.saldo);

```

### Figura 3.4: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

A saída do trecho de código mostrado na Figura 3.4 é:

```

Usuário: Hilário / Saldo: 1000.0
Usuário: Joãozinho / Saldo: 0

```

Além dos construtores que atuam na inicialização dos objetos, existem também os **destrutores**, que são executados antes de o objeto ser eliminado da memória. O método destrutor é chamado de “`finalize()`”. Cada objeto possui um método destrutor padrão, que não faz nada. Para criar um método destrutor para suas próprias classes, basta sobrepor o método “`finalize()`” com o seguinte cabeçalho:

```
Protected void finalize() {  
    // código necessário para limpeza do objeto;  
}
```

## 3.2 Encapsulamento

Em Java existem atributos que modificam o nível de acesso às variáveis definidas em uma classe, aos métodos, aos construtores e às próprias classes. São eles: **protected**, **private** e **public**.

- a) **Public**: indica que uma classe, variável, método ou construtor podem ser usados externamente ao pacote em que foram definidos. Sem ele o uso só pode ser feito internamente ao pacote no qual ocorre a definição.
- b) **Private**: indica que uma variável, método ou construtor só podem ser utilizados internamente à classe em que foram declarados.
- c) **Protected**: indica que o uso só pode ser feito na classe ou em uma subclasse.

Esses modificadores permitem a implementação do encapsulamento no Java. O **encapsulamento** faz com que não seja permitido acessarmos diretamente as propriedades de um objeto. Nesse caso, precisamos operar sempre por meio dos métodos pertencentes a ele.

Um problema da classe **Conta** criada na aula anterior é que não há qualquer impedimento com relação a um saque que deixe a conta com saldo menor que o limite permitido. Na verdade o método “sacar” já foi alterado de forma que essa verificação seja feita. Entretanto, nada garante que o usuário da classe **Conta** irá utilizar esse método **sempre** que precisar o valor do atributo “saldo”. O código a seguir ultrapassa o limite diretamente:

```
Conta minhaConta = new Conta();  
minhaconta.limite = 1000.0;  
minhaconta.saldo = -3000.0; // Saldo abaixo do limite.
```

A melhor forma de impedir que isso aconteça é forçar que o desenvolvedor seja sempre obrigado, por exemplo, a utilizar o método “sacar”. Para fazer isso no Java, basta declararmos que os atributos não podem ser acessados fora da classe, utilizando o modificador de acesso **private**, da seguinte forma:

```
class Conta {  
    int numero;  
    String nome;  
    private double saldo;  
    private double limite;  
    //métodos da classe...
```

### Figura 3.5: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

Dessa forma o exemplo anterior (que acessa diretamente o valor do atributo “saldo”) não iria compilar. Cada classe é responsável por controlar seus atributos; portanto ela deve julgar se aquele novo valor é válido ou não. Essa validação não deve ser controlada por quem está utilizando a classe, e sim por ela mesma (centralizando essa responsabilidade e facilitando futuras mudanças no sistema).

Repare que, agora, quem chama o método “sacar” não faz a menor ideia de que existe um limite que está sendo checado. Para quem for usar essa classe, basta saber **o que** o método faz e não exatamente **como** ele o faz (**“o que”** um método faz é sempre mais importante do que **“como”** ele faz: mudar a implementação é fácil; já mudar a assinatura de um método vai gerar problemas).

Veja outras aplicações de encapsulamento no nosso dia a dia.

Todos os celulares fazem a mesma coisa: eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem cada funcionalidade; mas, repare que para o usuário comum pouco importa se o celular é GSM ou CDMA; isso fica encapsulado.

### 3.2.1 Exemplo de código

Segue o código de um programa que guarda os dados digitados por um usuário e retorna os valores:

#### a) Pessoa.java

Nessa classe definimos os atributos como **privados** usando **private**. Depois criamos os **métodos set** e **get** para cada atributo da classe, lembrando que o primeiro nos permite definir um valor para o atributo, enquanto o segundo retorna o valor do atributo.

Da mesma forma que nos métodos construtores mais simples usados nos códigos das aulas anteriores, o **método set** usa o **this** para diferenciar o atributo de classe do parâmetro que será passado. Já o **método get** usa o **return**, que serve justamente para **retornar o valor**.

Podemos também observar que no **método set** foi usado o **void**, já que o **método set** não retornará nenhum valor, apenas servirá para definir um. O **método get**, no exemplo abaixo, foi construído usando **string**, já que é o atributo usado, e, por isso, retornará **string**.

```
public class Pessoa {  
    private String nome;  
    private String telefone;  
    private String endereco;  
  
    public void setNome(String nome) { this.nome = nome; }  
    public String getName() { return nome; }  
  
    public void setTelefone(String telefone) { this.telefone = telefone; }  
    public String getTelefone() { return telefone; }  
  
    public void setEndereco(String endereco) { this.endereco = endereco; }  
    public String getEndereco() { return endereco; }  
}
```

**Figura 3.6: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

### b) Main.java

O código a seguir é da **classe principal**, que será a primeira a ser executada. Como podemos perceber, ela é feita praticamente da mesma forma que as classes principais dos exemplos mostrados nas aulas anteriores. A diferença aqui é o **uso dos métodos set e get para definirmos e lermos os valores**.

Começamos dando o *import* necessário para usarmos o *JOptionPane* e depois **declararamos e instanciamos o objeto** da classe *Pessoas* para podermos usar seus **métodos set e get**.

Podemos ver que usamos o **método set** da seguinte forma:

```
objPessoa.setNome(JOptionPane.showInputDialog("Digite o nome"));
```

Isso fez com que aparecesse uma caixa de diálogo (*JOptionPane.showInputDialog*) pedindo para o usuário digitar um nome e depois esse nome foi passado pelo método *set* para o atributo *nome* da classe *Pessoa*. Ou seja, o

valor dentro dos parentes do método set é o que foi definido. Uma forma de passar o valor direto no código, sem a **interação do usuário**, seria:

```
objPessoa.setNome("Marcos Penha");
```

Depois usamos o **método get** sem parâmetros adicionais para retornar o valor. O *JOptionPane.showMessageDialog* mostra uma caixa de diálogo com uma mensagem específica. No código ele foi colocado com o método get justamente para **mostrar a caixa de mensagens com os valores digitados pelo usuário**.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        Pessoa objPessoa = new Pessoa(); //declaração e instanciação de objeto

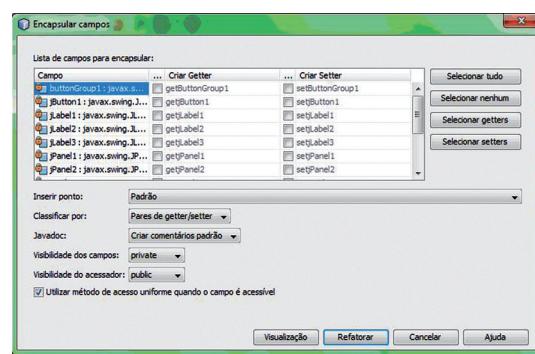
        objPessoa.setNome(JOptionPane.showInputDialog("Digite o nome"));
        objPessoa.setTelefone(JOptionPane.showInputDialog("Digite o telefone"));
        objPessoa.setEndereco(JOptionPane.showInputDialog("Digite o endereço"));

        JOptionPane.showMessageDialog(null, "Nome: " + objPessoa.getNome() +
            "\nTelefone: " + objPessoa.getTelefone() +
            "\nEndereço: " + objPessoa.getEndereco());
    }
}
```

**Figura 3.7: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

O *NetBeans* possui uma funcionalidade que realiza o encapsulamento dos dados para nós. Para isso, clique com o botão direito sobre a classe, escolha a opção “Refatorar” e depois clique em “Encapsular campos”. Aparecerá uma janela semelhante à mostrada na Figura 3.8:



**Figura 3.8: Encapsulamento dos dados**

Fonte: Elaborada pelo autor Hilário Júnior

Clique no botão “Selecionar tudo” para que sejam criados todos os métodos que modificam e os que retornam os atributos da classe. Em seguida, clique em “Refatorar”. Veja que o código da classe foi automaticamente modificado pelo *NetBeans*. Os atributos agora possuem o modificador “private” e já foram criados os métodos para manipular nossos dados.

## Resumo

Ao final desta aula, temos condições de encapsular os dados de uma classe e de criar métodos construtores e destrutores capazes de manipular esses dados.

## Atividades de aprendizagem

1. Crie um construtor para a classe **Empresa** que recebe como parâmetros um “nome” e um “cnpj” e o tamanho do array de funcionários. O atributo “qtde\_de\_funcionario” deve começar com o valor zero. Crie também um construtor para a classe **Empresa** que não receba o tamanho do array de funcionários. Nesse caso defina que o padrão é criá-lo com tamanho 100.
2. Crie um construtor para a classe **Funcionario** que receba os atributos “nome”, “departamento”, “salario” e RG. No atributo booleano considere que o funcionário está empregado. Crie também um construtor para a classe **Funcionario** que receba apenas nome e RG, e outro que não receba parâmetros.
3. Crie um objeto da classe Empresa. Em seguida crie vários objetos da classe **Funcionario** e inclua-os na <Empresa. Demita alguns e depois imprima a quantidade de funcionários da empresa utilizando o método “numeroDeFuncionarios”.
4. Adicione o modificador de visibilidade (*private*, se necessário) para cada atributo e método da classe **Funcionario**. Tente criar um **Funcionario** no “main” e modificar ou ler um de seus atributos privados. O que acontece?

- 5.** Crie métodos que alteram e que retornam cada atributo da classe **Funcionario**. Exemplo:

```
void setSalario (double salario) {  
    this.salario = salario;  
}  
void getSalario() {  
    return this.salario;  
}
```

- 6.** Modifique os códigos que acessavam atributos da classe **Funcionario** e faça com que eles passem a utilizar as funções acima.

Exemplo:

```
f.salario = 100;  
System.out.println(f.salario);
```

passa a ser:

```
f.setSalario(100);  
System.out.println(f.getSalario());
```

Registre suas respostas num arquivo digital e poste-o no ambiente virtual de ensino-aprendizagem (AVEA) do curso.



# Aula 4 – Polimorfismo (herança, sobrecarga e reescrita)

## Objetivos

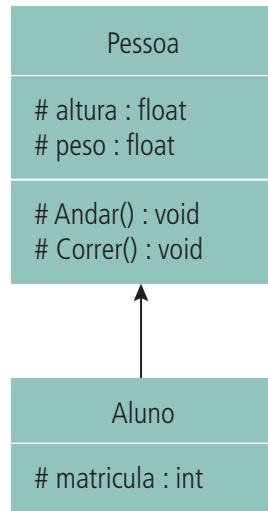
Desenvolver a habilidade de montar uma hierarquia de classes através dos conceitos de herança, sobrecarga e reescrita.

Uma das grandes características da orientação a objetos é a utilização do polimorfismo. A palavra polimorfismo vem do grego *poli morfos* e significa muitas formas. Na orientação a objetos isso representa uma característica em que se admite tratamento idêntico para formas diferentes (baseado em relações de semelhança, isto é, entidades diferentes podem ser tratadas de forma semelhante conferindo grande versatilidade aos programas e classes que se beneficiam dessas características).

Nesta aula estudaremos três formas de polimorfismo: a herança, a sobrecarga de métodos e a reescrita.

### 4.1 Herança

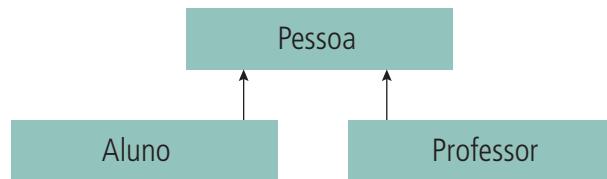
A ideia da herança é permitir que uma classe possa ser derivada de outra classe. Um **Estudante**, por exemplo, é uma **Pessoa**. Dessa forma, um objeto da classe **Estudante** (subclasse) deve “herdar” todos os atributos da classe **Pessoa** (superclasse), como altura, peso etc., bem como seus métodos (andar, correr etc.). O diagrama da Figura 4.1 mostra o relacionamento entre as classes base (pessoa) e derivada (estudante).



**Figura 4.1: Relacionamento das classes**

Fonte: Elaborada pelo autor Hilário Júnior

A grande vantagem da herança é o reaproveitamento de código. Quando utilizada corretamente, a herança permite reduzir o número de linhas de código. Essa redução facilita a programação, manutenção e verificação de erros. O diagrama da Figura 4.2 ilustra esse reaproveitamento. Note que agora, além da classe Estudante (herdeira de Pessoa), temos também a classe Professor (que também herda os atributos e métodos de Pessoa).



**Figura 4.2: Superclasse com duas subclasses**

Fonte: Elaborada pelo autor Hilário Júnior

A Figura 4.3 apresenta outro exemplo: será criada uma classe para os funcionários de um banco e outra específica para os gerentes. Um gerente guarda as mesmas informações que um funcionário comum, mas também possui informações e funcionalidades próprias (por exemplo, uma senha que permite seu acesso ao sistema interno do banco).

```

class Funcionario {
    String nome;
    String cpf;
    double salario;
    //métodos devem vir aqui
}

class Gerente {
    String nome;
    String cpf;
    double salario;
    int senha;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }
    //outros métodos
}

```

**Figura 4.3: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Para utilizar a herança, o correto é fazer com que o **Gerente** possua tudo o que um Funcionario possui sem reescrevermos códigos. Em Java fazemos com que a segunda classe seja uma **extensão** da primeira. Para isso, utiliza-se a palavra-chave **extends**.

```

class Gerente extends Funcionario {
    int senha;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }
}

```

**Figura 4.4: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

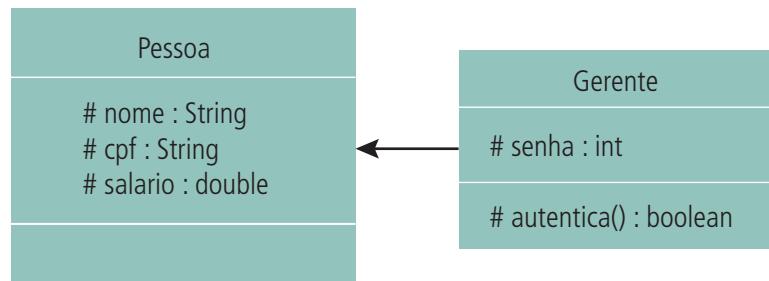
Uma subclasse herda todos os atributos e métodos de uma superclasse. Mas atenção para o seguinte: atributos e métodos privados (*private*) não podem ser acessados diretamente. Para conseguir acessar os atributos e métodos sem torná-los públicos, deve-se defini-los como protegidos (*protected*). Um atributo *protected* só pode ser acessado pela própria classe ou suas subclases. Veja o exemplo da Figura 4.5.

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
    //métodos devem vir aqui  
}
```

**Figura 4.5: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

A Figura 4.6 mostra o diagrama das classes.



**Figura 4.6: Classes Funcionário e Gerente**

Fonte: Elaborada pelo autor Hilário Júnior

## 4.2 Sobrecarga de métodos

Outra técnica de programação orientada a objetos (OO) é a sobrecarga de métodos (*method overload*). Com ela podemos ter vários métodos com o mesmo nome armazenados na mesma classe. Para o compilador conseguir distinguir entre os métodos, a quantidade de parâmetros ou o tipo de parâmetro deve ser diferente entre os métodos com mesmo nome.

O exemplo da Figura 4.7 ilustra este conceito. Note que existem dois métodos de nome “método”. Entretanto, em um deles temos dois parâmetros (um do tipo *long* e outro do tipo *int*) e no outro método um parâmetro do tipo *string*.

```
// Sobrecarga.java
public class Sobrecarga {
    public long metodo (int x) {
        return x+x;
    }

    public long metodo (long x, int y) {
        return x+y;
    }

    public long metodo (String x) {
        return x.length();
    }
}
```

**Figura 4.7: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Outro exemplo de Sobrecarga é justamente quando criamos mais de um construtor para uma classe. Cada construtor possui o número ou tipo de parâmetro diferente dos outros.

### 4.3 Reescrita de métodos

Outra estratégia é a **reescrita de métodos**. Com essa técnica um método em uma subclasse pode possuir comportamento diferente do método da superclasse. Observe o seguinte exemplo: no final do ano, os funcionários comuns do banco recebem uma bonificação de 10% de seus salários, mas os gerentes recebem 15%. As classes ficariam da forma mostrada na Figura 4.8.

```
class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double bonificacao() {
        return this.salario * 0.10;
    }

    // métodos devem vir aqui
}

class Gerente extends Funcionario {
    int senha;

    public double bonificacao () {
        return this.salario * 0.15;
    }

    // outros métodos
}

// outros métodos
```

**Figura 4.8: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Nessa situação, se for preciso acessar o método da superclasse, basta incluir a palavra “super”. Exemplo: “super.bonificacao()”. Nesse caso será chamado o método da superclasse.

## 4.4 Pacotes

Em grandes sistemas, quando normalmente trabalhamos com uma equipe de programadores, podemos nos deparar com situações em que as classes possam ter o mesmo nome. Para permitir essas características e facilitar a administração do código, podemos ter que organizar as classes em diretórios diferentes.

Os diretórios estão diretamente relacionados aos chamados pacotes e costumam agrupar classes de funcionalidades similares ou relacionadas. Por exemplo, no pacote “java.util” temos as classes **Date**, **SimpleDateFormat** e **GregorianCalendar** (todas elas trabalham com datas de formas diferentes).

Para fazer com que a classe **Cliente** esteja em um pacote chamado “banco”, ela deverá estar em um diretório chamado “banco”. A classe Cliente, que se localiza nesse último diretório mencionado, deve ser escrita da seguinte forma:

```
package banco;  
  
Class cliente {  
  
// ....  
  
}
```

A palavra-chave *package* indica em qual pacote/diretório está a classe. Um pacote pode conter nenhum ou mais subpacotes e/ou classes dentro dele. Os pacotes só possuem letras minúsculas, não importa quantas palavras estejam contidas nele. Esse padrão existe para evitar ao máximo o conflito de pacotes de empresas diferentes.

Para utilizarmos uma classe definida em outro arquivo, mas contida em um mesmo pacote, basta fazermos referência simples a ela. Vamos supor a existência das classes **Cliente** e **Banco** no pacote “banco”. A classe **Cliente** deve ser escrita assim:

```
package banco;
```

```
class Cliente {
```

```
    String nome;
```

```
    String endereço;
```

```
}
```

E **Banco** assim:

```
package banco;
```

```
class Banco {
```

```
    String nome;
```

```
    Cliente clientes[];
```

```
}
```

Note que **Banco** se referenciou a **Cliente** da mesma forma como fazíamos quando colocávamos as classes em um mesmo arquivo. Para utilizarmos classes definidas em outro pacote, utilizamos a palavra-chave *import*. Exemplo:

```
package angencia;  
  
import banco.Cliente  
  
class Agencia {  
    String nome;  
    int id;  
    Cliente c[];  
}
```

#### Figura 4.9: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

## Resumo

Nesta aula aprendemos o conceito de herança – fundamental para o desenvolvimento de programas orientados a objetos. Além desse conceito, conhecemos também outras duas técnicas: a sobrecarga e a reescrita.

## Atividades de aprendizagem

1. Implemente uma classe Estudante em um novo projeto, que herda de Pessoa e possui o atributo “matricula”. Em seguida implemente a classe Professor, que também herda de estudante. Qual atributo próprio essa classe poderia ter?
2. Implemente a classe Conta e adicione um método chamado “atualizar”, que atualiza o saldo de uma conta de acordo com uma “taxa” percentual, recebida como parâmetro.
3. Crie duas subclasses que herdam da classe Conta: ContaCorrente e Poupança.
4. O método “aniversário” da classe Pessoa não recebia parâmetros e incrementava a idade de um objeto dessa classe. Crie agora outro método chamado “aniversario”, nessa mesma classe, que receba um inteiro como parâmetro indicando quantos aniversários a pessoa fez.
5. Reescreva o método “atualizar” da classe Conta nas subclasses **ContaCorrente** e **Poupança**. Na primeira o método atualizará o saldo com o dobro da taxa recebida como parâmetro. Na segunda, atualizará com o triplo da taxa (utilize *super* para acessar o valor do saldo na superclasse). Reescreva também o método “depositar” na classe **ContaCorrente**, descontando do saldo o valor de 0,35% do que foi depositado.
6. Crie um objeto da classe **Conta**, um da classe **ContaCorrente** e outro da classe **Poupança**. Deposite R\$1.000,00 em cada uma delas. Chame em seguida o método “atualizar” e depois veja o saldo final de cada uma.

Registre suas respostas num arquivo digital e poste-o no ambiente virtual de ensino-aprendizagem (AVEA) do curso.

# Aula 5 – Controle de erros: exceções

## Objetivo

Criar tratamento para os diversos erros que podem ocorrer durante a execução de um programa.

Um dos aspectos importantes em qualquer linguagem de programação é o tratamento de erros. Por mais correto e testado que esteja o seu programa, ele não estará imune a erros. O usuário pode lançar dados errados, fazer operações inesperadas ou até mesmo podemos nos deparar com problemas técnicos (falta de energia, falha em um determinado dispositivo etc.). Qualquer linguagem de programação que se preze deve possuir estruturas para tratamento de erros. Com a linguagem Java isso não é diferente. Em Java temos estruturas muito parecidas com as existentes na linguagem C++. São os comandos: *try-catch* e *try-finally*. Ambos procuram simplificar um pouco a vida do programador, fazendo com que não sejam necessários muitos testes de verificação e avaliação ao realizar certas operações.

### 5.1 Try-catch

Quando ocorre um ou mais tipos de erro dentro de um trecho de código delimitado, o *TRY-CATCH* desvia automaticamente a execução para uma rotina designada para o tratamento específico desse erro. A sintaxe é a apresentada na Figura 5.1.

```
try {  
    // código normal  
}  
catch (<exceção 1>) {  
    // código de tratamento do primeiro tipo de erro  
}  
catch (<exceção 2>) {  
    // código de tratamento do segundo tipo de erro  
}  
catch (<exceção 3>) {  
    // código de tratamento do terceiro tipo de erro  
}
```

**Figura 5.1: Sintaxe para try-catch**

Fonte: Elaborada pelo autor Hilário Júnior

Por exemplo, podemos criar um programa que precisa receber um número inteiro da linha de comando. Como os argumentos são passados em um vetor de *strings*, precisamos transformar a *string* contendo o número para um inteiro. Se a conversão gerar um erro, significa que o argumento não é um número inteiro válido. A exceção usada, nesse caso, é a *java.lang.NumberFormatException*.

Outro erro que podemos tratar é o caso de não ser fornecido o argumento desse mesmo programa, utilizando a exceção *ArrayIndexOutOfBoundsException*. Nesse caso ocorrerá um erro ao tentarmos acessar o índice 0 do vetor (que está vazio). O código a seguir mostra como fazemos esses dois tratamentos com o *TRY-CATCH*.

```
int j = 10;

try {
    while (j > Integer.parseInt (args [0])) {
        System.out.println ("+"+j);

        j--;
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println ("Não foi fornecido um argumento.");
} catch (java.lang.NumberFormatException e) {
    System.out.println("Argumento não é um inteiro válido.");
}
```

**Figura 5.2: Tratamentos com o comando try-catch**

Fonte: Elaborada pelo autor Hilário Júnior

Podem existir inúmeros blocos *catch* no tratamento de erros (cada um para um tipo de exceção).

## 5.2 Try-finally

Com o *TRY-FINALLY* uma rotina de finalização é sempre executada (independentemente de ocorrer um erro). O trecho de código contido em *FINALLY* será sempre executado. Veja o código na Figura 5.3.

```

try {
    <código normal>;
} finally {
    <código que sempre deve ser executado>;
}

```

**Figura 5.3: Sintaxe de *try-finally***

Fonte: Elaborada pelo autor Hilário Júnior

Um mesmo *try* pode ser usado com as diretivas *catch* e *finally*. A Figura 5.4 mostra um exemplo de código utilizando *try*, *catch* e *finally*.

```

public class TratamentoDeErro {

    public static void main ( String[] args ) {

        int[] array = {0, 1, 2, 3, 4, 5}; // array de 6 posições

        try {
            for (int i=0; i<10; i++) {

                array [i] += i;

                System.out.println(array [i]);

            }

            System.out.println ("bloco executado com sucesso");
        }

        catch (ArrayIndexOutOfBoundsException e) {

            System.out.println ("Acessou im índice inexistente");
        }

        catch ( Exception e ) {

            System.out.println ("Outro tipo de exceção ocorreu");
        }

        finally {

            System.out.println ("Outro tipo de exceção ocorreu");
        }

        finally {

            System.out.println ("Isto SEMPRE executa!");
        }
    }
}

```

**Figura 5.4: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

## 5.3 Tipos de erros

Em Java podemos trabalhar com três tipos de erros:

- a) erro de *Runtime* (*java.lang.Runtime*) – causado por erros desconhecidos durante a execução do programa;
- b) erro de Sistema (*java.lang.Error*) – erros imprevisíveis causados por falha em dispositivos tais como: disco, banco de dados, etc.;
- c) erro Customizado (*java.lang.Exception*) – erros ou condições especiais previstas no programa. Esses erros são subclasse da classe **Exception**.

Algumas exceções muito comuns são:

- a) **ArithmeticException** – divisão de um valor por zero;
- b) **NullPointerException** – quando se faz referência a um objeto ainda não criado (instanciado);
- c) **ArrayIndexOutOfBoundsException** – normalmente esse tipo de erro acontece quando acessamos uma posição em um vetor (*array*) que não existe.

## 5.4 Gerando uma exceção

Uma alternativa é tratarmos determinados erros da nossa aplicação. Vamos retomar o método de controle bancário que controla os débitos em uma conta corrente. Um cenário possível é o cliente tentar fazer um saque no caixa acima do valor do seu saldo. Até agora esse “problema” havia sido tratado da seguinte forma:

```
boolean sacar (double quantidade) {  
    if (quantidade > (this.saldo + this.limite)) {  
        return false;  
    }  
    else {  
        this.saldo = this.saldo - quantidade;  
        return true;  
    }  
}
```

**Figura 5.5: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Uma alternativa é lançar a exceção **IllegalArgumentException**. Veja o código da Figura 5.6.

```
void sacar (double quantidade) throws IllegalArgumentException
{
    if (quantidade > (this.saldo + this.limite)) {
        // aqui lançamos a exceção:
        throw new IllegalArgumentException ();
    }
    else {
        this.saldo = this.saldo - quantidade;
    }
}
```

#### Figura 5.6: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

O comando *throw* lança uma exceção, em seguida um bloco *catch* captura a exceção e faz o tratamento. Agora podemos tratar o erro num nível mais alto, ou seja, onde o método de “*sacar()*” foi chamado. Para isso não usaremos mais um *if/else* e sim um *try-catch* (o que faz mais sentido, já que a falta de saldo é uma exceção):

```
Conta cc = new Conta();
cc.depositar(100);

try {
    cc.sacar(100);
} catch (IllegalArgumentException e) { // capturando a exceção
    System.out.println("Saldo Insuficiente");
}
```

#### Figura 5.7: Exemplo de código

Fonte: Elaborada pelo autor Hilário Júnior

Para melhorar ainda mais nosso tratamento de erro, podemos fazer com que o método “*sacar*” passe uma mensagem junto com a exceção, explicando o motivo do erro:

```
void sacar (double quantidade) throws IllegalArgumentException {  
    if (quantidade > (this.saldo + this.limite)) {
```

**Figura 5.8: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

```
        throw new IllegalArgumentException ("Saldo Insuficiente");  
    }  
    else {  
        this.saldo = this.saldo - quantidade;  
    }  
}
```

**Figura 5.9: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Para termos acesso a essa mensagem, utilizamos o método “getMessage()”:

```
Conta cc = new Conta();  
cc.depositar(100);  
  
try {  
    cc.sacar(100);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage()); /* Aqui estamos im-  
    primindo a mensagem de erro veio junto com a exceção */  
}
```

**Figura 5.10: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Se a exceção lançada for capturada, o restante do código dentro do *try* não será executado. Se tentássemos imprimir uma mensagem confirmando o saque logo após chamarmos o método “sacar”, ela seria realmente impressa somente se o método não retornasse uma exceção. Veja no exemplo da Figura 5.11.

```
try {  
    cc.sacar(100);  
    System.out.println("Saque realizado com sucesso.");  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}
```

**Figura 5.11: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

## 5.5 Criando suas próprias exceções

Uma alternativa quando as exceções disponíveis não são suficientes é que o programador crie sua própria classe de exceção para o tratamento de erros. Veja o código na Figura 5.12.

```
public class SaldoInsuficienteException extends Exception {  
  
    SaldoInsuficienteException (String message) {  
        super(message);  
    }  
}
```

**Figura 5.12: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Neste ponto podemos lançar a exceção criada: **SaldoInsuficienteException**. Uma vez que essa classe é herdeira de **Exception**, qualquer método que a lançar (utilizando a diretiva *throw* deverá, obrigatoriamente, anunciarla com a diretiva *throws*, como já havíamos feito no último código do método sacar. Isso faz com que qualquer chamada a esse método tenha que estar inserida num bloco *try* que capture essa exceção. No caso de usarmos a **IllegalArgumentException**, isso não seria necessário. O código poderia simplesmente ser o mostrado na Figura 5.13.

```
void sacar (double quantidade) {  
    if (quantidade > (this.saldo + this.limite)) {  
        throw new IllegalArgumentException("Saldo Insuficiente");  
    }  
    else {  
        this.saldo = this.saldo - quantidade;  
    }  
}
```

**Figura 5.13: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

A chamada do método, nesse caso, poderia ser:

```
Conta cc = new Conta ();  
cc.depositar (100) ;  
cc.sacar (100) ;
```

**Figura 5.14: Exemplo de código**

Fonte: Elaborada pelo autor Hilário Júnior

Sempre que a exceção for descrita (utilizando *throws*), ela precisa ser tratada no local onde o método foi chamado.

## Resumo

Nesta aula aprendemos a manipular os comandos *try-catch* e *try-finally* para tratar e criar exceções. Além disso, descobrimos como o tratamento correto de erros pode manter o programa em execução, mesmo na ocorrência de eventos indesejados.

## Atividades de aprendizagem

1. Insira exceções no método “depositar” da classe **Conta**, verificando se o valor passado como parâmetro for inválido (negativo, por exemplo), utilizando **IllegalArgumentException**. Teste o método criado.
2. Crie sua própria exceção chamada **ValorInvalido-Exception** e utilize-a no método acima.
3. Em quais outros métodos vistos neste material (nos exemplos e nos exercícios) podemos inserir o tratamento de exceções?

Registre suas respostas num arquivo digital e poste-o no ambiente virtual de ensino-aprendizagem (AVEA) do curso.

## Referências

COSTA, Daniel Gouveia. **Java em rede:** programação distribuída na internet. Rio de Janeiro: Brasport, 2008.

DEITEL, Harvey M.; DEITEL, Paul J. **Java:** como programar. São Paulo: Prentice-Hall, 2005.

SEIBEL JÚNIOR, Hilário. **Linguagens de programação.** Colatina: IFES, 2009.

SANTOS, Rafael. **Introdução à programação OO usando Java.** Rio de Janeiro: Editora: Campus, 2003.

## **Curriculum dos professores-autores**

**Gustavo Campos Menezes** graduado em Tecnologia em Informática, pelo Unicentro Newton Paiva, mestre em Ciência da Computação, pela UFMG. Professor efetivo do CEFET-MG desde setembro de 2006, trabalha principalmente com disciplinas relacionadas a Linguagens de Programação e Algoritmos para cursos de Informática (Ensino técnico) e Engenharias (Graduação).

**Ygor Colen Morato** técnico em Eletrônica formado pelo Colégio COTEOM em 1998, graduado em Sistemas de Informação pela Fadom, em Divinópolis, em 2005. Trabalhou na Gerdau Divinópolis como técnico em Eletrônica durante 11 anos. Atualmente é coordenador técnico do Colégio Cecon e professor Substituto do CEFET –MG Campus V Divinópolis, onde leciona Aplicativos para Web 2, Linguagem de Programação 2, Redes de Computadores e Sistemas Operacionais.

**Hilário Seibel Júnior** mestre em Informática e graduado em Ciência da Computação pela Universidade Federal do Espírito Santo (Ufes). É professor efetivo do Ifes desde dezembro de 2007, tendo lecionado para o Bacharelado em Sistemas de Informação, Tecnólogo de Redes e Tecnólogo de Análise e Desenvolvimento de Sistemas. Atualmente é coordenador dos Cursos Superiores de Informática do IFES (Campus Serra). Foi professor substituto na Ufes de 2005 a 2007, tendo lecionado para Ciência da Computação, Engenharia da Computação, Engenharia Elétrica e Física.



e-Tec<sup>rede</sup>  
Brasil

CÓDIGO DE BARRAS  
ISBN