GOVERNO DO ESTADO DO CEARÁ

Secretaria da Educação

ESCOLA ESTADUAL DE EDUCAÇÃO PROFISSIONAL - EEEP

ENSINO MÉDIO INTEGRADO À EDUCAÇÃO PROFISSIONAL

CURSO TÉCNICO EM INFORMÁTICA

PROGRAMAÇÃO ORIENTADA A OBJETO I

GOVERNO DO ESTADO DO CEARÁ

Secretaria da Educação

Governador
Cid Ferreira Gomes

Vice Governador
Francisco José Pinheiro

Secretária da Educação
Maria Izolda Cela de Arruda Coelho

Secretário Adjunto
Maurício Holanda Maia

Secretário Executivo
Antônio Idilvan de Lima Alencar

Assessora Institucional do Gabinete da Seduc
Cristiane Carvalho Holanda

Coordenadora de Desenvolvimento da Escola
Maria da Conceição Ávila de Misquita Vinãs

Coordenadora da Educação Profissional – SEDUC
Thereza Maria de Castro Paes Barreto

Índice

Apresentação

Autor

Florence Tiu Balagtas

Equipe

Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4

- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional Processador Memória HD Livre

Microsoft Windows 500 MHz Intel Pentium III
workstation ou equivalente
512 MB 850 MB
Linux 500 MHz Intel Pentium III
workstation ou equivalente
512 MB 450 MB
Solaris OS (SPARC) UltraSPARC II 450 MHz 512 MB 450 MB
Solaris OS (x86/x64
Platform Edition)
AMD Opteron 100 Série 1.8 GHz 512 MB 450 MB
Mac OS X PowerPC G4 512 MB 450 MB

Configuração Recomendada de Hardware

Sistema Operacional Processador Memória HD Livre

Microsoft Windows 1.4 GHz Intel Pentium III
workstation ou equivalente
1 GB 1 GB
Linux 1.4 GHz Intel Pentium III
workstation ou equivalente
1 GB 850 MB
Solaris OS (SPARC) UltraSPARC IIIi 1 GHz 1 GB 850 MB
Solaris OS (x86/x64
Platform Edition)
AMD Opteron 100 Series 1.8 GHz 1 GB 850 MB
Mac OS X PowerPC G5 1 GB 850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço:
http://developer.apple.com/java (é necessário registrar o download da JDK).
Para mais informações:
http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori	Hugo Leonardo Malheiros Ferreira	Mauro Regis de Sousa Lima
Alexis da Rocha Silva	Ivan Nascimento Fonseca	Namor de Sá e Silva
Aline Sabbatini da Silva Alves	Jacqueline Susann Barbosa	Néres Chaves Rebouças
Allan Wojcik da Silva	Jader de Carvalho Belarmino	Pedro Antonio Pereira Miranda
André Luiz Moreira	João Aurélio Telles da Rocha	Pedro Henrique Pereira de Andrade
Andro Márcio Correa Louredo	João Paulo Cirino Silva de Novais	Renato Alves Félix
Antoniele de Assis Lima	João Vianney Barrozo Costa	Renato Barbosa da Silva

Antonio Jose R. Alves Ramos	José Augusto Martins Nieviadonski	Reyderson Magela dos Reis
Aurélio Soares Neto	José Leonardo Borges de Melo	Ricardo Ferreira Rodrigues
Bruno da Silva Bonfim	José Ricardo Carneiro	Ricardo Ulrich Bomfim
Bruno dos Santos Miranda	Kleberth Bezerra G. dos Santos	Robson de Oliveira Cunha
Bruno Ferreira Rodrigues	Lafaiete de Sá Guimarães	Rodrigo Pereira Machado
Carlos Alberto Vitorino de Almeida	Leandro Silva de Morais	Rodrigo Rosa Miranda Corrêa
Carlos Alexandre de Sene	Leonardo Leopoldo do Nascimento	Rodrigo Vaez
Carlos André Noronha de Sousa	Leonardo Pereira dos Santos	Ronie Dotzlaw
Carlos Eduardo Veras Neves	Leonardo Rangel de Melo Filardi	Rosely Moreira de Jesus
Cleber Ferreira de Sousa	Lucas Mauricio Castro e Martins	Seire Pareja
Cleyton Artur Soares Urani	Luciana Rocha de Oliveira	Sergio Pomerancblum
Cristiano Borges Ferreira	Luís Carlos André	Silvio Sznifer
Cristiano de Siqueira Pires	Luís Octávio Jorge V. Lima	Suzana da Costa Oliveira
Derlon Vandri Aliendres	Luiz Fernandes de Oliveira Junior	Tásio Vasconcelos da Silveira
Fabiano Eduardo de Oliveira	Luiz Victor de Andrade Lima	Thiago Magela Rodrigues Dias
Fábio Bombonato	Manoel Cotts de Queiroz	Tiago Gimenez Ribeiro
Fernando Antonio Mota Trinta	Marcello Sandi Pinheiro	Vanderlei Carvalho Rodrigues Pinto
Flávio Alves Gomes	Marcelo Ortolan Pazzetto	Vanessa dos Santos Almeida
Francisco das Chagas	Marco Aurélio Martins Bessa	Vastí Mendes da Silva Rocha
Francisco Marcio da Silva	Marcos Vinicius de Toledo	Wagner Eliezer Roncoletta
Gilson Moreno Costa	Maria Carolina Ferreira da Silva	Nolyanne Peixoto Brasil Vieira
Givailson de Souza Neves	Massimiliano Giroldi	Paulo Afonso Corrêa
Gustavo Henrique Castellano	Mauricio Azevedo Gamarra	Paulo José Lemos Costa
Hebert Julio Gonçalves de Paula	Mauricio da Silva Marinho	Paulo Oliveira Sampaio Reis
Heraldo Conceição Domingues	Mauro Cardoso Mortoni	

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel de Oliveira** – JU GLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão,

disponibilização dos materiais e inserção de novos módulos

- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa

Rommel Feria – Criador da Iniciativa

Lição 1

1. Objetivos

Nesta seção, vamos discutir os componentes básicos de um computador, tanto em relação a hardware como a software. Também veremos uma pequena introdução sobre linguagens de programação e sobre o ciclo de vida do desenvolvimento. Por fim, mostraremos os diferentes sistemas numéricos e as conversões entre eles.

Ao final desta lição, o estudante será capaz de:

- Identificar os diferentes componentes de um computador.
- Conhecer as linguagens de programação e suas categorias.
- Entender o ciclo de vida de desenvolvimento de programas e aplicá-lo na solução de problemas.
- Conhecer os diferentes sistemas numéricos e as conversões entre eles.

2. Explorando o Java

2.1. Um pouco da história

Java foi criado em 1991 por James Gosling da Sun Microsystems. Inicialmente chamada OAK(Carvalho), em homenagem à uma árvore de janela do Gosling, seu nome foi mudado para Javadevido a existência de uma linguagem com o nome OAK.

Figura 1: James Gosling criador do Java

A motivação original do Java era a necessidade de uma linguagem independente de plataformaque podia ser utilizada em vários produtos eletrônicos, tais como torradeiras e refrigeradores. Um dos primeiros projetos desenvolvidos utilizando Java era um controle remoto pessoal chamado *7 (Star Seven).

Ao mesmo tempo, a World Wide Web e a Internet foram ganhando popularidade. Gosling achava que a linguagem Java poderia ser usada para programação da Internet.

2.2. O que é uma tecnologia JAVA?

2.2.1. Uma linguagem de programação

Como linguagem de programação, Java pode ser utilizado para criar todos os tipos de aplicações existentes, de programas de Inteligência Artificial para Robôs até programas para aparelhos celulares.

2.2.2. Um ambiente de desenvolvimento

Como ambiente de desenvolvimento, a tecnologia Java fornece um grande conjunto deferramentas: um compilador, um interpretador, um gerador de documentação, ferramenta de empacotamento de classes de arquivos e outros.

2.2.3. Um ambiente de aplicação

Aplicações de tecnologia Java são tipicamente programas de propósito geral que executam sobreuma máquina onde o Java Runtime Environment é instalado.

2.2.4. Um ambiente de distribuição

Há dois ambientes de distribuição principais: Primeiro, o **JRE**, fornecido através do Java 2 Software Development Kit (SDK), contém um conjunto completo de arquivos de classes para todos pacotes de tecnologia Java. Outro ambiente de distribuição é o **navegador web**, ou seja, o **browser**. Os navegadores web atuais fornecem interpretação à tecnologia e ambiente Java em tempo de execução.

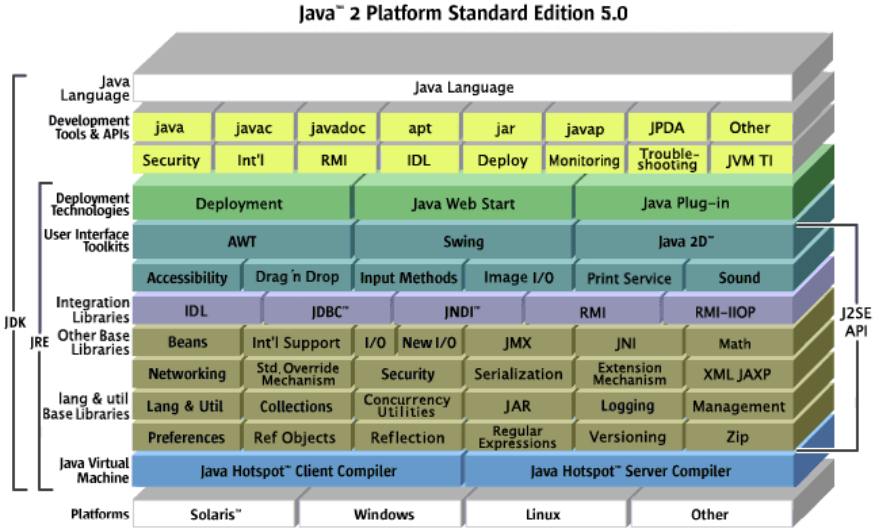


Figura 3: JDK e JRE

2.3. Algumas Características do JAVA

2.3.1. Máquina Virtual Java

A **Máquina Virtual Java** é uma máquina imaginária que é implementada através de um softwareemulador em uma máquina real. A JVM provê especificações de plataforma de hardware na qual compila-se todo código de tecnologia Java. Essas especificações permitem que o software Javaseja uma plataforma independente pois a compilação é feita por uma máquina genéricaconhecida como JVM.

O **bytecode** é uma linguagem de máquina especial que pode ser entendida pela **Máquina Virtual Java (JVM)**. O bytecode é independente de qualquer hardware de computador particular. Assim, qualquer computador com o interpretador Java pode executar um programa Java compilado, não importando em que tipo de computador o programa foi compilado.

2.3.2. Garbage Collection

Muitas linguagens de programação permitem ao programador alocar memória durante o tempo de execução. Entretanto, após utilizar a memória alocada, deve existir uma maneira para desalocar o bloco de memória de forma que os demais programas a utilizem novamente. Em C, C++ e outras linguagens o programador é o responsável por isso. Isso, às vezes, pode ser difícil já que instâncias podem ser esquecidas de serem desalocadas da memória pelos programadores e resultar no que chamamos de escapes da memória.

Em Java, o programador não possui a obrigação da retirar uma variável criada das áreas de memória, isto é feito por uma parte da JVM específica que chamamos de **Garbage Collection**. O **Garbage Collection** é o grande responsável pela liberação automática do espaço em memória.

Isso acontece automaticamente durante o tempo de vida do programa Java.

2.3.3. Segurança de código

Segurança do Código é alcançada em Java através da implementação da **Java Runtime Environment** (**JRE**). A JRE roda códigos compilados para a JVM e executa o carregamento de classes (através do **Class Loader**), verificação de código (através do verificador de **bytecode**) e finalmente o código executável.

O **Class Loader** é responsável por carregar todas as classes necessárias ao programa Java. Isso adiciona segurança através da separação do **namespace** entre as classes do sistema de arquivos local e aquelas que são importadas pela rede. Isso limita qualquer ação de programas que podem causar danos, pois as classes locais são carregadas primeiro. Depois de carregar todas as classes, a quantidade de memória que o executável irá ocupar é determinada. Isto acrescenta,

novamente, uma proteção ao acesso não autorizado de áreas restritas ao código pois a quantidade de memória ocupada é determinada em tempo de execução.

Após carregar as classes e definir a quantidade de memória, o **verificador de bytecode** verifica o formato dos fragmentos de código e pesquisa nestes fragmentos por códigos ilegais que possam violar o direito de acesso aos objetos.

Depois que tudo isso tiver sido feito, o código é finalmente executado.

2.4. Fases do Java programa

A figura seguinte descreve o processo de compilação e execução de um programa Java. O primeiro passo para a criação de um programa Java é escrever os programas em um editor de texto. Exem-

plos de editores de texto que podem ser utilizados: bloco de notas, vi, emacs, etc.Esses arquivos são armazenados no disco rígido com a extensão .java.

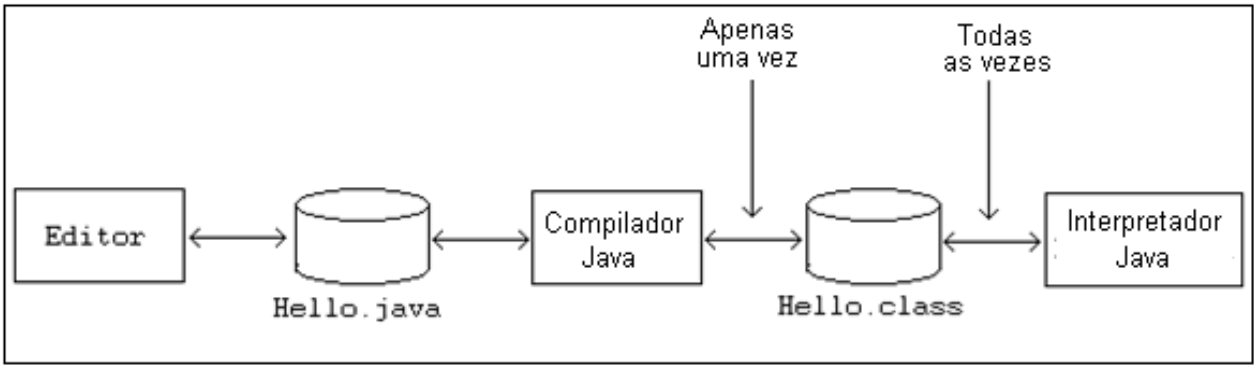


Figura 1: Fases de um Programa Java.

Após o programa Java ter sido criado e salvo, compile o programa utilizando o Compilador Java. A saída desse processo é um arquivo de **bytecode** com extensão .class. O arquivo .class é então lido pelo Interpretador Java que converte os bytecodes em linguagem de máquina do computador que se está usando.

Tarefa Ferramenta utilizada Saída

Escrever o programa Qualquer editor de texto Arquivo com extensão .java Compilar o programa Compilador Java Arquivo com extensão .class (Java bytecode). Executar o programa Interpretador Java Saída do programa

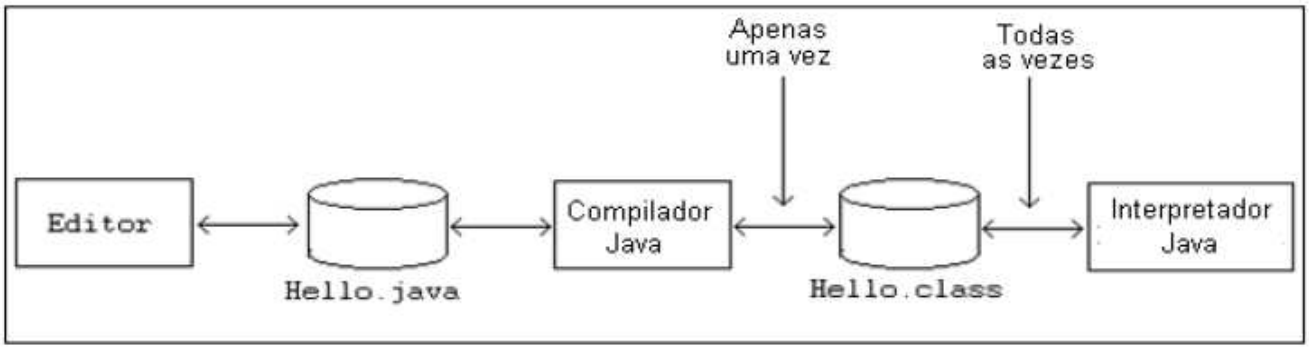


Tabela 1: Resumo das fases de um programa Java.

3. Primeiro Programa Java

Antes de explicar o que o programa significa, vamos escrevê-lo e executá-lo.

3.1. Utilizando a Console e um editor de texto

Neste exemplo utilizaremos um simples editor de texto, que pode ser o **gedit** do Linux ou o **notepad** do Windows, para editar o código fonte. Em seguida será necessário abrir uma janela terminal para compilar e executar os programas.

Passo 1: executar um editor de texto

Para iniciar um editor de texto no **Linux** selecione Applications ⟹ Accessories ⟹ Text Editor. Para iniciar um editor de texto no **Windows** selecione Start ⟹ Programs ⟹ Accessories ⟹ Notepad.

Passo 2: Abrir a janela de console

Para abrir o terminal no **Linux**, selecione Applications ⟹ Accessories ⟹ Terminal. Para abrir o terminal no **Windows**, selecione Start ⟹ Run... e na janela que se apresenta, digite **cmd** e pressione o botão OK.

Passo 3: Escrever as instruções utilizando o Editor de Texto

Digite as seguintes instruções no editor de textos:

```
public class Hello
{
      /**
      * Meu primeiro programa Java
      */
      public static void main(String[] args) {
      // Mostra na tela o texto "Hello world"
      System.out.println("Hello world!");
      }
}
```

Passo 4: Salvar o programa Java

Chamaremos o programa de "Hello.java" e o colocaremos em uma pasta denominada "myJavaPrograms". Caso esta pasta não tenha sido criada, retorne à janela de terminal aberta e insira as seguintes instruções:

Para o **Linux**:

```
$ md myJavaPrograms
```

Para o **Windows**:

```
C:\> md myJavaPrograms
```

Retorne ao Editor de textos e salve o programa. Para abrir a caixa de diálogo salvar selecione aopção "File" localizada na barra de menus e depois clique na opção "Save". Selecione a nova pasta cria-

da como myJavaPrograms para entrar nela. A pasta deve estar vazia porque ainda não salvamos nada dentro dela.

Na caixa de texto "Name", digite o nome do programa (Hello.java), e depois clique no botão salvar.

ATENÇÃO: Para o **Notepad** no Windows, mude o Tipo para "All Files" (em Save as Type).

Após salvar o arquivo observe que o título da janela mudou de "Untitled" para "Hello.java", caso deseje alterar novamente o arquivo basta editá-lo e depois salvá-lo novamente clicando em File $\Rightarrow$ Save.

Passo 5: Entrar na pasta que contém o programa

O próximo passo deve ser o de compilar o programa. Inicialmente, precisamos entrar na pasta que o contém. Retorne à janela do terminal.

Em **Linux**:

Normalmente, quando abrimos uma janela terminal, ela vai diretamente para sua pasta home (identificada por $). Para ver o que tem dentro do diretório digite **ls** (LS em minúscula, significando "List Sources") e pressione ENTER. Isso fará com que sejam listados os arquivos e pastas da pasta home. Verifique a existência de uma pasta chamada "myJavaPrograms", criada a pouco, sendo esta olocal em que foi salvo o programa "Hello.java". Mudaremos o contexto para esta pasta.

Para entrar nesta pasta devemos utilizar o comando: cd [nome da pasta]. O comando "cd" significa "Change Directory". Digitaremos:

```
$ cd myJavaPrograms
```

Agora que estamos dentro da pasta onde o arquivo do programa está, poderemos então compilá-lo. Certifique-se de que o arquivo está realmente dentro desta, executando o comando **ls** (LS em minúscula) novamente.

Em **Windows**:

Normalmente, quando abrimos uma janela terminal ela vai diretamente para sua pasta raiz (identificada por C:\). Para conhecer o conteúdo do diretório digite **dir** (significando "directory") e pressione ENTER. Isso fará com que sejam listados os arquivos e pastas da pasta principal. Verifique a existência de uma pasta chamada "myJavaPrograms", criada a pouco, sendo esta o local em que foi salvo o programa "Hello.java". Mudaremos o contexto para esta pasta.

Para entrar nesta pasta devemos utilizar o comando: cd [nome da pasta]. O comando "cd" significa "Change Directory". Digitaremos:

```
C:\>cd myJavaPrograms
```

Agora que estamos dentro da pasta onde o arquivo do programa está, poderemos então compilá-lo. Certifique-se de que o arquivo está realmente dentro desta, executando o comando **dir** novamente.

Passo 6: Compilar o programa

Para compilar o programa, utilizamos o comando: **javac [Nome do Arquivo]**. Ou seja:

```
javac Hello.java
```

Durante a compilação, é criado o arquivo: **[Nome do Arquivo].class,** neste caso, **Hello.class**, que contém o código em linguagem de máquina (chamado de **bytecode**).

Passo 7: Executar o programa

Assumindo que não ocorreu problemas na compilação (caso tenha ocorrido qualquer problema refaça os passos realizados), estamos prontos para executar o programa.

Para executar o programa, utilizamos o comando: **java [nome do arquivo sem a extensão]**.

No caso do exemplo, digite:

```
java Hello
```

Veremos na mesma tela, em que foi executado o comando, a seguinte mensagem:

```
Hello world!
```

3.2. Erros

Vimos um pequeno programa Java, geralmente não encontraremos qualquer problema para compilar e executar esses programas, entretanto nem sempre este é o caso, como mencionamos na primeira parte deste curso, ocasionalmente encontramos erros durante esse processo.

Como mencionamos antes, há dois tipos de erros: o primeiro pode ocorrer durante a compilação, chamado de erro de sintaxe, o segundo pode ocorrer durante a execução, chamado *runtime error*.

3.2.1 Erros de Sintaxe

Os erros de sintaxe normalmente são erros de digitação, ocasionados pelo programador que podeter se equivocado e digitar uma instrução errada, ou por esquecimento de alguma parte da instrução, por exemplo, um ponto e vírgula. O Compilador tenta isolar o erro exibindo a linha de instrução e mostrando o primeiro caractere incorreto naquela linha, entretanto, um erro pode não estar exatamente neste ponto. Outros erros comuns são a troca de letras, troca de letras maiúscula por minúscula (a linguagem Java é completamente **case- sensitive**, ou seja, o caractere "a" é completamente diferente do caractere "A", e o uso incorreto da pontuação.

Vamos retornar ao exemplo, o programa **Hello.java**. Intencionalmente, escreveremos a palavrachave "static" de forma errada e omitiremos o ponto-e-vírgula em uma instrução e a deixaremos errada.

```
public class Hello
{
	/**
	* Meu primeiro programa Java
	*/
	public statict void main(String[] args) {
		// A linha abaixo foi retirado o ;
		System.out.println("Hello world!")
	}
}
```

Salve o programa e execute os passos necessários para compilá-lo. Observe a mensagem de erro gerada ao se tentar compilar novamente o programa:

```
Hello.java:6: <identifier> expected
public statict void main(String[] args) {
	^
	Hello.java:10: ';' expected
}
^
1 error
```

A primeira mensagem de erro sugere que existe um erro na linha 6 do programa apontado para apalavra **void**, entretanto esta palavra está correta. O erro é na palavra anterior **statict** que deveser digitada como **static**.

A segunda mensagem de erro sugere que faltou um ponto-e-vírgula na linha 10, entretanto, esta contém simplesmente o comando de fechar o bloco do método main. O erro está exatamente na linha anterior. Como regra, ao encontrar muitas mensagens de erros devemos corrigir o primeiro erro da lista e tente novamente compilar o programa. Deste modo reduziremos o número total de mensagens de erro dramaticamente, pois podem existir o que chamamos de erros derivados, ou seja, um erro que tem por causa a instrução anterior.

3.2.2 Erros em tempo de execução (Erros de run- time)

Os erros em tempo de execução são erros que não aparecerão até que tentemos executar o programa. Os programas são compilados com sucesso, mas apresentarão respostas erradas, quepodem ter como causa se o programador não obedeceu uma lógica coerente ou no caso em erro de estruturas do programa.

4. Usando NetBeans

Construímos o programa sem utilizar nenhum recurso sofisticado, iremos aprender como fazertodo o processo da seção anterior utilizando uma IDE. Nesta parte da lição utilizaremos o **NetBeans** que é um Ambiente de Desenvolvimento Integrado (IDE - Integrated Development Environment).

Um ambiente de desenvolvimento integrado é um software aplicativo que possui uma interface construtora, um editor de texto, um editor de código, um compilador e/ou interpretador e umdepurador.

Passo 1 : executar o NetBeans

Existem duas formas de executar o NetBeans: a primeira é utilizando a linha de comandos de uma janela terminal e segunda é selecionar o ícone de atalho encontrado na janela da área de trabalho.

Para executar o NetBeans por intermédio da linha de comando, abra uma janela terminal (Os passos para abrir a janela terminal foram discutidos anteriormente) e digite:

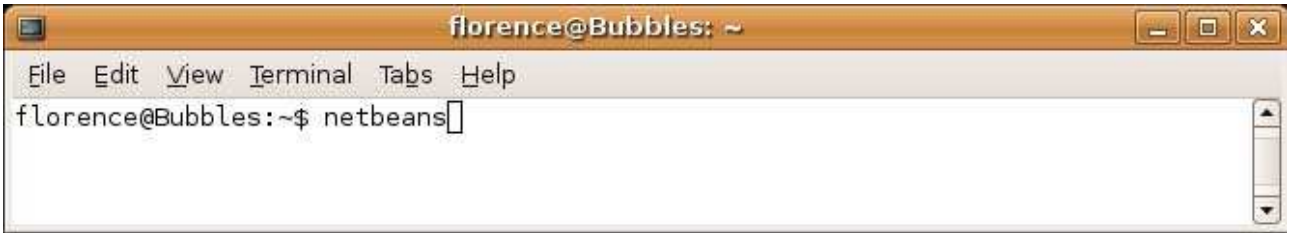


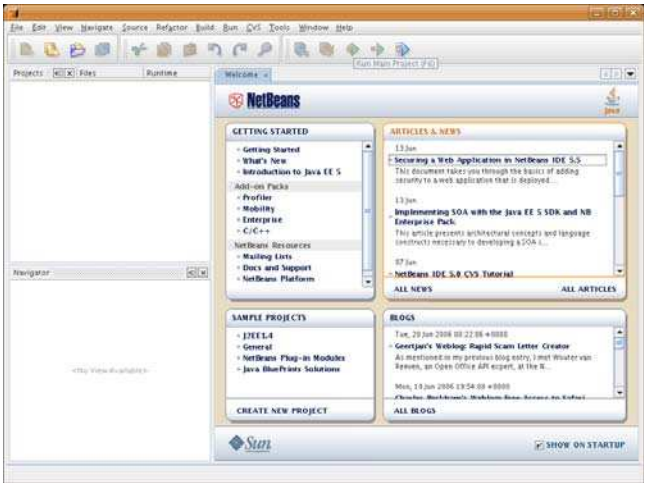
Figura 1: Executando o NetBeans pela linha de comandos

Para o Windows, este comando deve ser executado na pasta em que o NetBeans foi instalado, por exemplo:

```
C:\Program Files\netbeans-5.5\bin>netbeans
```

A segunda maneira de executar o NetBeans é clicando no ícone de atalho encontrado na área de trabalho do computador.

Depois de abrir a IDE NetBeans será mostrada a interface gráfica GUI, conforme à Figura 3:



Passo 2: construir o projeto

Clique em File ⇒ New Project, depois de fazer isso, uma janela de diálogo aparecerá. Neste momento deve-se clicar em "Java Application" e em seguida clicar no botão "Next >".

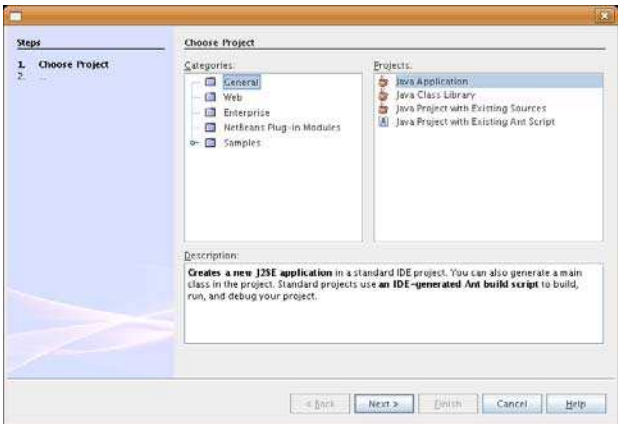
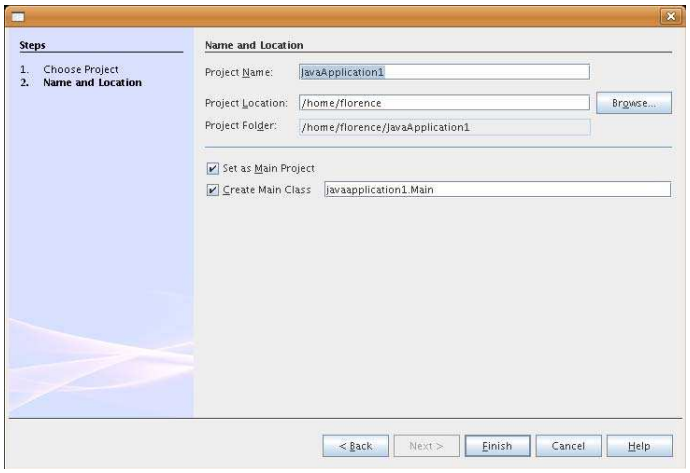


Figura 3: Janela de Welcome do NetBeans

Será mostrada uma nova janela de diálogo, conforme a figura 5.



Troque o local da aplicação clicando no botão "Browse...". Aparecerá uma janela de diálogo para localização do diretório. Dê um clique duplo no seu diretório home.

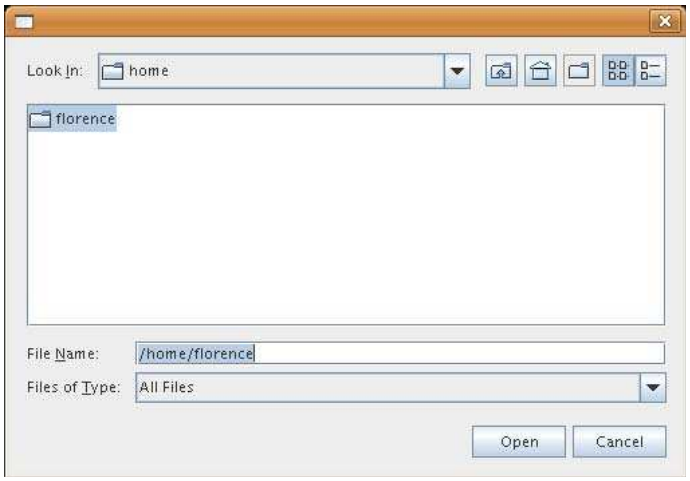


Figura 5: Inserindo as informações do projeto

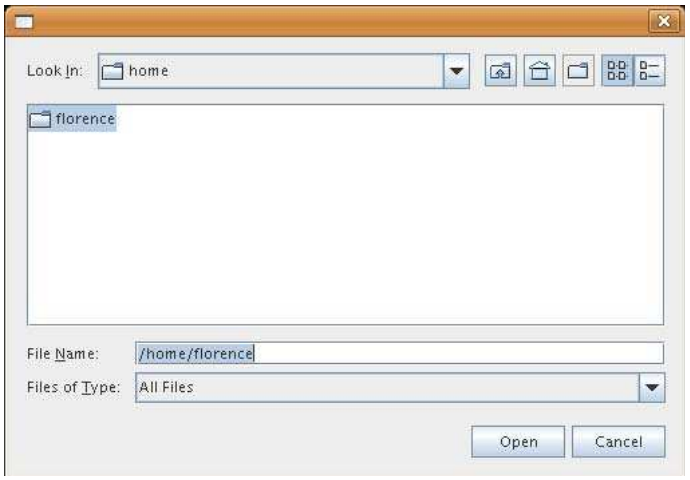
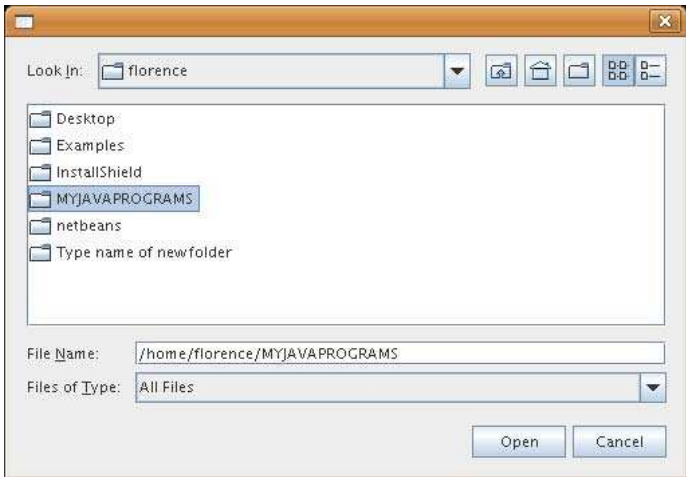


Figura 6: Acertando a Localização do Projeto

O conteúdo da raiz do diretório será apresentado. Dê um clique duplo no diretório MYJAVAPROGRAMS e depois dê um clique no botão "Open".



Veja que a localização do projeto mudou para /home/florence/MYJAVAPROGRAMS. Finalmente, no campo "Create Main Class", digite "Hello", que será o nome da classe principal, e em seguida clique no botão "Finish".

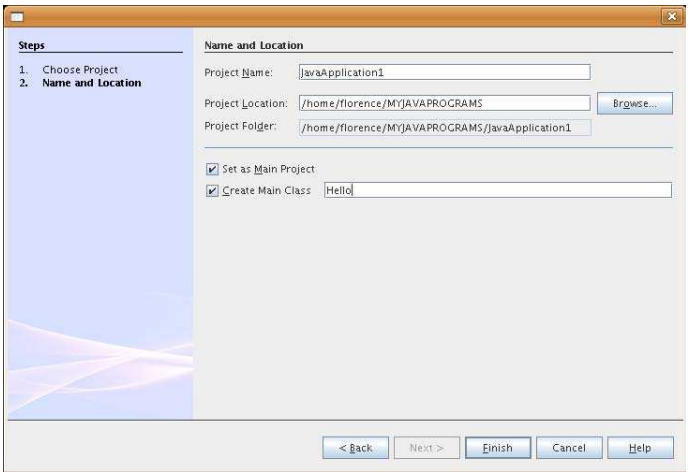
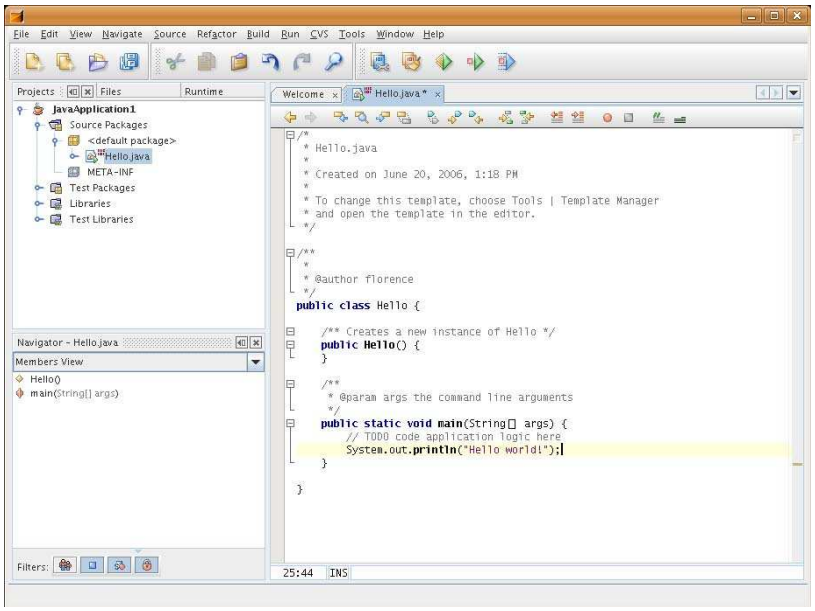


Figura 7: Definindo o Nome da Classe Principal

Passo 3: escrever os detalhes do programa

Antes de escrever o programa descreveremos a janela principal. Como mostrado na **figura 8**, automaticamente, o NetBeans cria um código básico para o programa Java. Poderemos adicionar as declarações neste código gerado. No lado esquerdo da janela visualizamos uma lista de pastas e arquivos que o NetBeans gerou antes de criar o projeto. Tudo se encontra dentro da sua pasta MYJAVAPROGRAMS, onde foi configurado o local do projeto. No lado direito, visualizamos o código gerado.



Modifique o código gerado pelo NetBeans, por hora ignoraremos as outras partes das instruções discutindo os detalhes destas posteriormente. Insira a seguinte instrução:

```
System.out.println("Hello world!");
```

Isto significa que você deseja que seja mostrada a mensagem "Hello world!" na saída padrão do computador, em seguida seja feito um salto de linha. Poderíamos substituir esta instrução por duas equivalentes:

```
System.out.print("Hello");
System.out.println(" world!");
```

O método print() faz com que não seja provocado o salto de linha, utilizaremos para este exemplo a primeira instrução. Insira esta instrução após a linha de comentário (que será desprezada pelo compilador):

```
//TODO code application logic here.
```

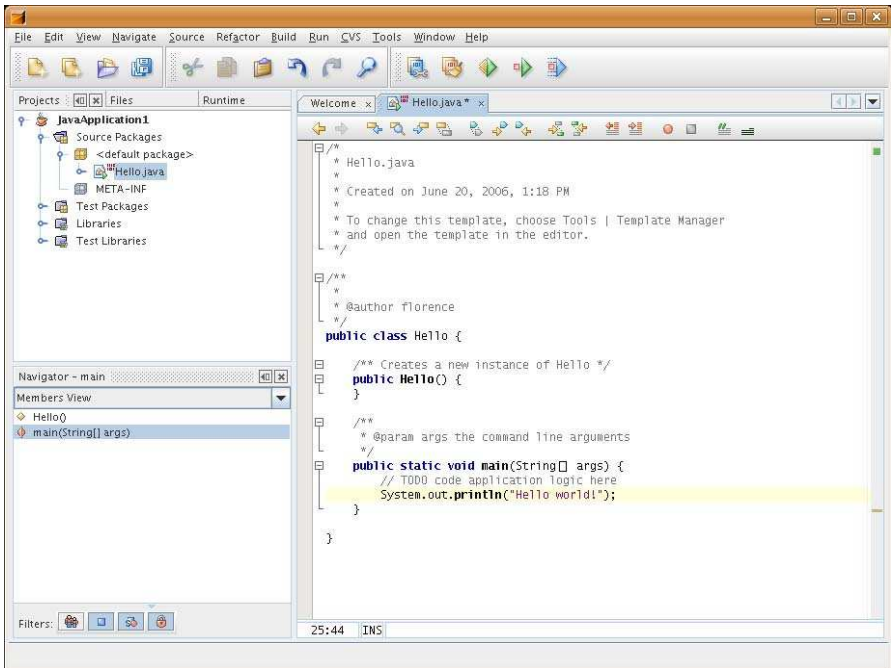


Figura 8: Visão do projeto criado

Passo 4 : compilar o projeto

Para compilar o programa, a partir do Menu Principal selecione Build ⇒ Build Main Project, ou utilize a tecla de atalho F11, ou utilize o botão de atalho para compilar o código.

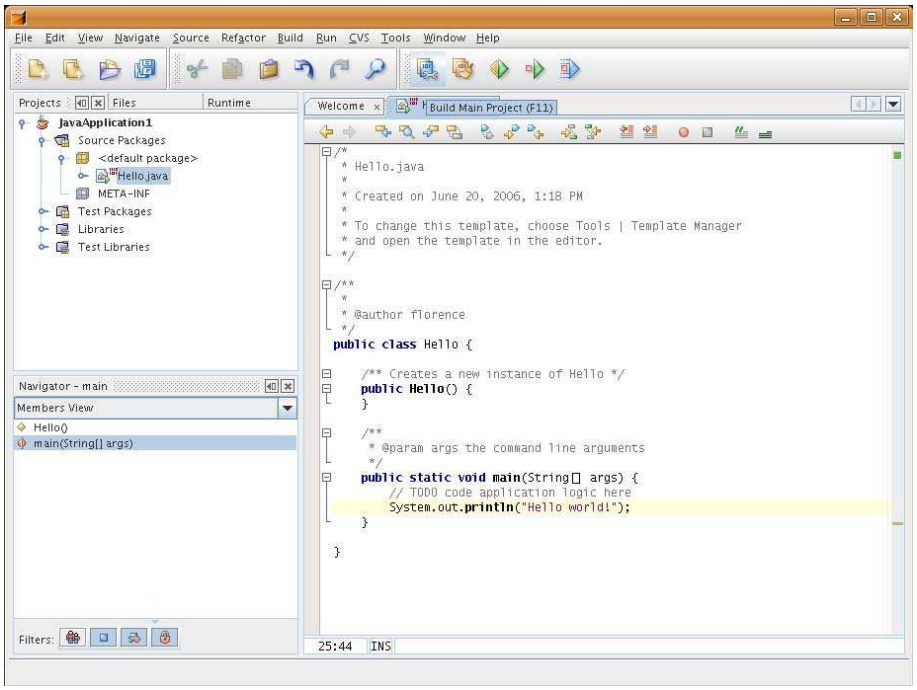


Figura 10: Botão de Atalho para executar o Projeto

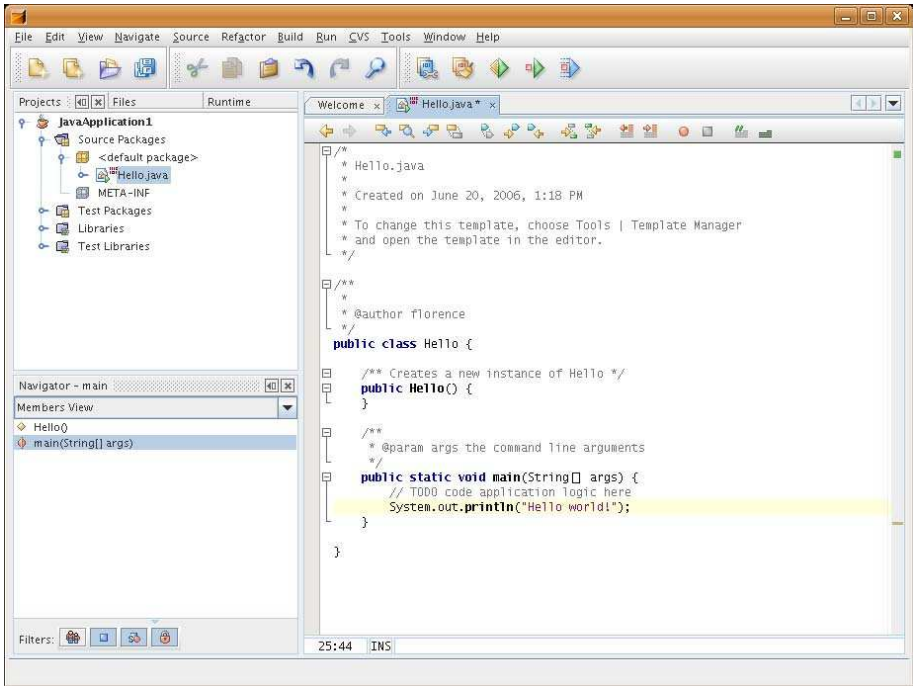


Figura 9: Inserindo sua instrução

Se não existir erros no programa, veremos a mensagem de sucesso na janela de saída.

Passo 5: Executar o projeto

Para executar o programa, clique em Run ⟹ Run Main Project, ou utilize a tecla de atalho F6, ouutilize o botão de atalho para executar o programa.

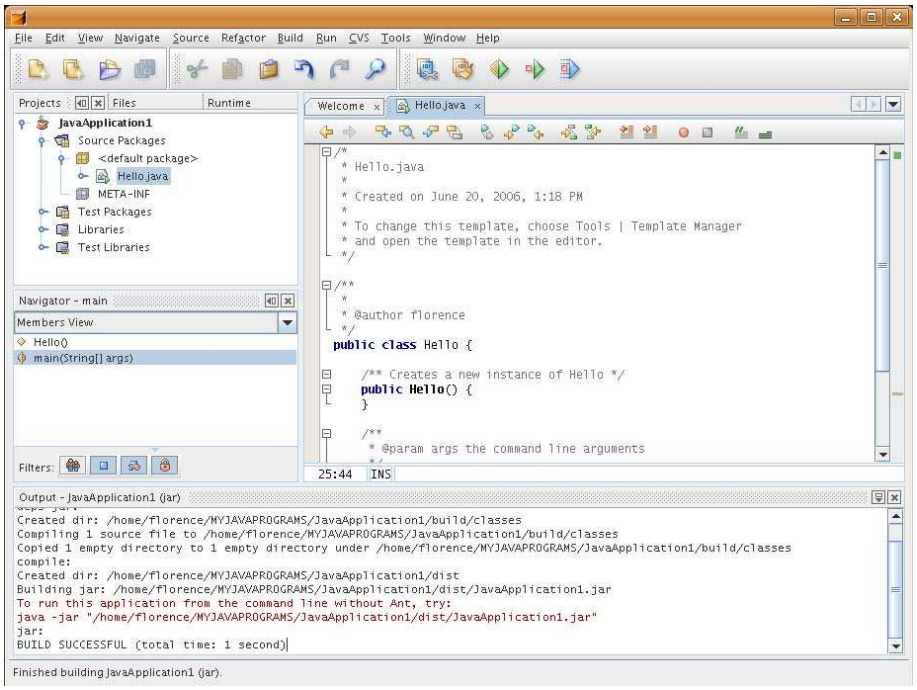


Figura 11: Verificando o Sucesso da Compilação

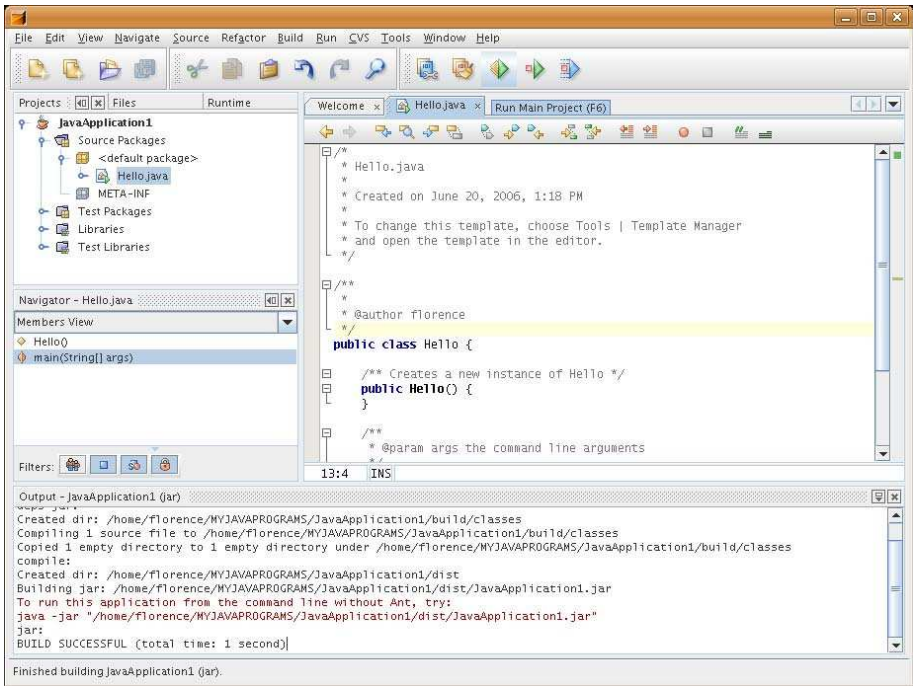


Figura 12: Executando o projeto

O resultado final do programa, será mostrado na janela de saída.

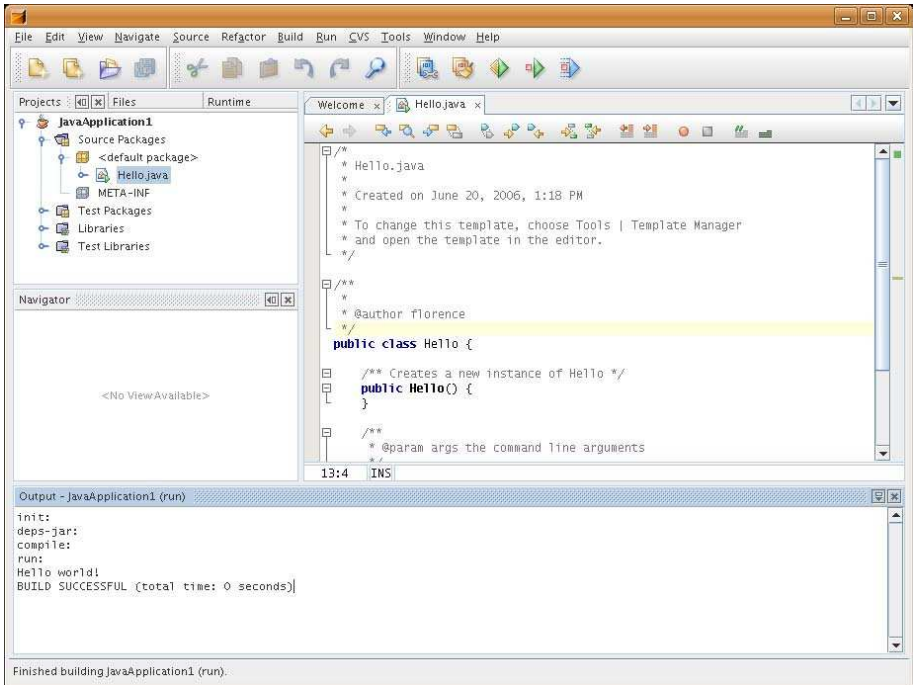


Figura 13: Resultado final da execução do projeto

5. Exercícios

5.1 Melhorando o Hello World!

Utilizando o NetBeans crie uma classe chamada [SeuNome], o programa deverá mostrar comoresultado a mensagem:

```
Welcome to Java Programming [SeuNome]!!!
```

5.2. A Árvore

Utilizando o NetBeans, crie uma classe chamada **TheTree**. O programa deverá mostrar asseguintes linhas na saída:

```
I think that I shall never see,
[Eu acho que nunca verei,]
a poem as lovely as a tree.
[um poema tão adorável quanto uma árvore.]
A tree whose hungry mouth is pressed
[Uma árvore cuja boca faminta é pressionada]
Against the Earth’s sweet flowing breast.
[Contra a Terra fluindo em seu seio docemente.]
```

Lição 2

1. Objetivos

Nesta lição iremos discutir um pouco da história de Java e o que é a tecnologia Java. Também iremos discutir as fases de um programa Java.

Ao final desta lição, o estudante será capaz de:
- Descrever as características da tecnologia Java como a JVM - Máquina Virtual Java, Garbage Collection e segurança do código;
- Descrever as diferentes fases de um programa Java.

2. Entendendo meu primeiro programa em Java

Tentaremos compreender este primeiro programa.

```
public class Hello
{
/**
* Meu primeiro programa em Java
*/
      public static void main(String[] args) {
      // Exibir a mensagem "Hello world" na tela
            System.out.println("Hello world!");
      }
}
```

Esta primeira linha do código:

```
public class Hello
```

Indica o nome da classe, que neste caso é **Hello**. Em Java, todo e qualquer código deverá serescrito dentro da declaração de uma classe. Fazemos isso usando a palavra-chave **class**. Além disso, a classe usa um identificador de acesso **public**, indicando que a classe é acessível para outras classes de diferentes pacotes (pacotes são coleções de classes). Trataremos de pacotes e identificadores de acesso mais tarde, ainda neste módulo.

A próxima linha contém uma chave { . Indica o início de um bloco de instruções. Neste programa posicionamos a chave na linha após a declaração da classe, entretanto, poderíamos também colocá-la na mesma linha em que a declaração foi feita. Então, o código seria escrito da seguinte forma:

```
public class Hello
{
```

ou

```
public class Hello {
```

As próximas 3 linhas indicam um comentário em Java. Um comentário é uma explicação do programador usada na documentação de uma parte do código. Este comentário não é propriamente uma parte do código, é usado apenas para fins de documentação do programa.

É uma boa prática de programação adicionar comentários relevantes ao código.

```
/**
* Meu primeiro programa em Java
*/
```

Um comentário pode ser indicado pelos delimitadores "/*" e "*/". Qualquer coisa entre estes

delimitadores é ignorado pelo compilador Java e é tratado como comentário. A próxima linha,

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

que também pode ser escrita da seguinte forma:

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

indica o nome de um método no programa que é o método principal **main.** O método **main** éo ponto de partida para qualquer programa feito em Java. Todo e qualquer programa escrito em Java, com exceção de Applets, inicia com o método **main**. Certifique-se de que a assinatura do método (conforme descrita acima) está correta.

A linha seguinte é também um comentário em Java,

```
// exibe a mensagem "Hello world" na tela
```

Até agora, já aprendemos duas maneiras de fazer comentários em Java. Uma é posicionar o comentário entre "/*" e "*/", e a outra é colocar "//" antes do comentário. A linha de instrução abaixo,

```
System.out.println("Hello world!");
```

escreve o texto "Hello World!" na tela. O comando System.out.println( ), escreve na saídapadrão do computador o texto situado entre aspas duplas. As últimas duas linhas, que contêm somente uma chave em cada, simbolizam, respectivamente, o fechamento do método **main** e da **classe**.

Dicas de programação :

1. Os programas em Java devem sempre conter a terminação .java no nome do arquivo.
2. O nome do arquivo deve sempre ser idêntico ao nome da classe pública. Então, por exemplo, se o nome da classe pública é **Hello** o arquivo deverá ser salvo com o nome: **Hello.java**.
3. Inserir comentários sobre o que a classe ou método realiza, isso facilitará o entendimento de quem posteriormente ler o programa, incluindo o próprio autor.

3. Comentários em Java

Comentários são notas escritas pelo programador para fins de documentação. Estas notas não fazem parte do programa e não afetam o fluxo de controle. Java suporta três tipos de comentários: comen-

tário de linha estilo C++, comentário de bloco estilo C e um comentário estilo Javadoc (utilizado compor a documentação do programa).

3.1. Comentário de linha

Comentários com estilo em C++ se iniciam por "//". Todo e qualquer texto colocado após as **//** é ignorado pelo compilador e tratado como comentário. Por exemplo:

```
// Este é um comentário estilo C++ ou comentário de linha
```

3.2. Comentário de bloco

Comentários com estilo em C, também chamados de comentários multi-linhas, se iniciam com **/*** e terminam com ***/**. Todo o texto posto entre os dois delimitadores é tratado como comentário. Diferente do comentário estilo C++, este pode se expandir para várias linhas. Por exemplo:

```
/*
* Este é um exemplo de comentário
* estilo C ou um comentário de bloco
*
*/
```

3.3. Comentário estilo Javadoc

Este comentário é utilizado na geração da documentação em **HTML** dos programas escritos em Java. Para se criar um comentário em estilo **Javadoc** deve se iniciar o comentário com **/**** e terminá-lo com ***/**. Assim como os comentários estilo C, este também pode conter várias linhas. Este comentário também pode conter certas tags que dão mais informações à documentação. Por exemplo:

```
/**
Este é um exemplo de um comentário especial usado para \n
gerar uma documentação em HTML. Este utiliza tags como:
@author Florence Balagtas
@version 1.2
*/
```

Este tipo de comentário deve ser utilizado antes da assinatura da classe:

```
public class Hello {
```

Para documentar o objetivo do programa ou antes da assinatura de métodos:

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

Para documentar a utilidade de um determinado método.

4. Instruções e Bloco em Java

Uma instrução é composta de uma ou mais linhas terminadas por ponto-e-vírgula. Um exemplo de uma simples instrução pode ser:

```
System.out.println("Hello world");
```

Um bloco é formado por uma ou mais instruções agrupadas entre chaves indicando que formam uma só unidade. Blocos podem ser organizados em estruturas aninhadas indefinidamente. Qualquer quantidade de espaços em branco é permitida. Um exemplo de bloco pode ser:

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

Dicas de programação:

1. Na criação de blocos, a chave que indica o início pode ser colocada ao final da linha anterior ao bloco, como no exemplo:

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

ou na próxima linha, como em:

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

2. É uma boa prática de programação organizar as instruções que serão colocadas apóso início de um bloco, como por exemplo:

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

5. Identificadores em Java

Identificadores são representações de nomes de variáveis, métodos, classes, etc. Exemplos de identificadores podem ser: Hello, main, System, out. O compilador Java difere as letras maiúsculas de minúsculas (case-sensitive). Isto significa que o identificador **Hello** não é o mesmo que **hello**. Os identificadores em Java devem começar com uma letra, um underscore "_", ou um sinal de cifrão "$". As letras podem estar tanto em maiúsculo quanto em minúsculo. Os caracteres subseqüentes podem usar números de 0 a 9. Os identificadores não podem ter nomes iguais às palavras-chave ou palavras reservadas do Java, como: class, public, void, int, etc. Discutiremos mais sobre estas palavras mais tarde.

Dicas de programação:
1. Para nomes de classes, a primeira letra deve ser maiúscula. Nomes de métodos ou variáveis devem começar com letra minúscula. Por exemplo:
ExemploDeNomeDeUmaClasse
exemploDeNomeDeUmMetodo
2. No caso de identificadores com mais de uma palavra, a primeira letra de cada palavra, com exceção da primeira, deve vir em maiúsculo. Por exemplo:
`charArray – fileNumber - className`
3. Evite o uso de undescores no início de um identificador. Por exemplo:
`_NomeDeClasse.`

6. Palavras- chave em Java

Palavras-chave são identificadores que, em Java, foram pré-definidas para propósitos específicos. Não se pode usar esses identificadores como nomes de variáveis, métodos, classes, etc. A seguir, temos a lista com as palavras-chave em Java.

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Figura 1: Palavras-Chave em Java

Ao longo dos tópicos seguintes, iremos abordar os significados destas palavras-chave e comosão usadas nos programas em Java.

Nota: **true**, **false** e **null** não são palavras-chave, porém, são palavras-reservadas, e, da mesma maneira, não é permitido seu uso na atribuição a nomes de variáveis, métodos ou classes.

7. Tipos de Dados em Java

Java possui 4 tipos de dados. Estes tipos de dados são divididos em: **boolean**, **character**, **integer** e **float- point**.

7.1. Boolean

Um dado **boolean** poderá assumir somente dois valores: **true** ou **false**.

7.2. Character

Os **characters** são representações da tabela **Unicode**. Um símbolo da tabela **Unicode** é um valor de 16 bits que pode substituir os símbolos da tabela **ASCII** (que possuem 8 bits). A tabela **Unicode** permite a inserção de símbolos especiais ou de outros idiomas. Para representar um caractere usam-se aspas simples. Por exemplo, a letra "a" será representada como 'a'. Para representar caracteres especiais, usa-se a "\" seguido pelo código do caractere especial. Por exemplo, '\n' para o caractere de nova linha, '\'' representar o **character '** (aspas simples) e '\u0061' representação **Unicode** do símbolo 'a'.

7.3. Integer

Os dados do tipo **integer** vêm em diferentes formatos: **decimal** (base 10), **hexadecimal** (base 16), e **octal** (base 8). Ao usar estes diferentes tipos de dados **Integer** nos programas é necessário seguir algumas notações preestabelecidas. Para dados decimais não existe notação, basta escrever o número. Os números hexadecimais deverão ser precedidos por "0x" ou "0X". E os octais deverão ser precedidos por "0".

Por exemplo, considere o número **12**. Sua representação em decimal é apenas o número **12**, sem nenhuma notação adicional, enquanto que sua representação em hexadecimal é **0xC** (pois o número 12 em hexadecimal é representado pela letra C), e em octal ele é representado por **014**. O valor padrão para tipos de dados **Integer** é o tipo **int**. Um **int** é um valor, com sinal, de 32 bits.

Em alguns casos pode ser necessário forçar o dado **Integer** a ser do tipo **long**, para fazer isso basta colocar a letra "l" (L minúscula) ou "L" após o número. Um dado do tipo **long** é um valor com sinal de 64 bits. Falaremos mais sobre os tipos de dados mais tarde.

7.4. Float- Point

Os tipos de **float- point** representam dados **Integer** com parte fracionária. Um exemplo é o número 3.1415 (lembrando que o padrão inglês utiliza o "." como divisor da parte fracionária).

Esses tipos podem ser expressos em notação científica ou padrão. Um exemplo de notação padrão é o número 583.45 enquanto que em notação científica ele seria representado por 5.8345e2. O valor padrão para um dado ponto-flutuante é o double, que é um valor de 64 bits.

Se for necessário usar um número com uma precisão menor (32 bits) usa-se o **float**, que é finalizado pela letra "f" ou "F" acrescida ao número em questão, por exemplo, 583.45f.

8. Tipos de Dados Primitivos

A linguagem Java possui 8 tipos de dados primitivos. Eles são divididos nas seguintes representações:

Representação	Tipo de Dado	Dado Primitivo
lógico	Boolean	boolean
inteiro	Integer e Character	char, byte, short e int
inteiro longo	Integer	long
número fracionário	Float-point	float e double

Tabela 1: Representações dos dados primitivos

8.1. Lógico

O tipo **boolean** pode representar dois estados: ***true*** (verdadeiro) ou ***false*** (falso). Um exemplo é:

```
boolean resultado = true;
```

No exemplo demonstrado acima, é declarado um atributo chamado **resultado** do tipo **boolean** e atribuído a este o valor verdadeiro.

8.2. Inteiro

Os inteiros em Java podem ser representados em 5 formas, como já foi visto, e estas são: decimal, octal, hexadecimal, ASCII e **Unicode**. Como por exemplo:

```
2 // valor 2 em decimal
077 // 0 indica que ele está representado em octal.
0xBACC // 0x indica que ele está representado em hexadecimal.
'a' // representação ASCII
'\u0061' // representação Unicode
```

O dado do tipo **char** é um inteiro especial, sendo exclusivamente positivo e representa um único **Unicode**. Ele deve ser, obrigatoriamente, colocado entre aspas simples (''). Sua representação como inteiro pode ser confusa para o iniciante, entretanto, o tempo e a prática farão com que se acostume com este tipo. Por exemplo:

```
char c = 97; // representa o símbolo 'a'
byte b = 'a'; // em inteiro representa o número 97
```

É possível definir para qualquer inteiro nas formas mostradas. O que difere o tipo **char** dosdemais inteiros é que a sua saída sempre será mostrada como um valor ASCII. Enquanto que os inteiros serão sempre mostrados por números decimais. Por exemplo:

```
char c = 97;
byte b = 'a';
System.out.println("char = " + c + " - byte = " + b);
Resultará:
char = a - byte = 97
```

Um cuidado deve ser tomado quanto aos inteiros: qualquer operação efetuada entre eles terá sempre como resultado um tipo **int**. Por exemplo:

```
byte b1 = 1;
byte b2 = 2;
byte resultado = b1 + b2;
```

Esta instrução final causará um erro de compilação, devendo ser modificada para:

```
int resultado = b1 + b2;
```

8.3. Inteiro Longo

Os inteiros têm por padrão o valor representado pelo tipo primitivo **int.** Pode-se representá-loscomo **long** adicionando, ao final do número, um "l" ou "L". Os tipos de dados inteiros assumem valores nas seguintes faixas:

Tamanho em memória	*Dado primitivo*	*Faixa*
8 bits	byte	-2^7 até 2^7-1
16 bits	char	0 até $2^{16}-1$
16 bits	short	-2^{15} até $2^{15}-1$
32 bits	int	-2^{31} até $2^{31}-1$
64 bits	long	-2^{63} até $2^{63}-1$

Tabela 2: Tipos e faixa de valores dos Inteiros e Inteiro Longo

Dicas de programação:

1. Para declarar um número como sendo um long é preferível usar “L” maiúsculo, pois, se este estiver em minúsculo, pode ser difícil distinguí-lo do dígito 1.

8.4. Número Fracionário

Os dados do tipo ponto-flutuante possuem o valor double como padrão. Os números flutuantes possuem um ponto decimal ou um dos seguintes caracteres:

```
E ou e // expoente
F ou f // float
D ou d // double
```

São exemplos,

```
3.14 // tipo double
6.02E23 // double com expoente
2.718F // float
123.4E+306D // double
```

No exemplo acima, o número 23 após o E é implicitamente positivo. É equivalente a 6.02E+23. Os dados de tipo ponto-flutuante podem assumir valores dentro das seguintes faixas:

Tamanho em memória	*Dado primitivo*	*Faixa*
32 bits	float	-10^{38} até $10^{38}-1$
64 bits	double	-10^{308} até $10^{308}-1$

Tabela 3: Tipos e faixa de valores dos Número Fracionários

9. Variáveis

Uma **variável** é um espaço na memória usado para armazenar o estado de um objeto.Uma variável deve ter um **nome** e um **tipo**. O **tipo da variável** indica o tipo de dado que ela pode conter. O **nome das variáveis** deve seguir as mesmas regras de nomenclatura que os identificadores.

9.1. Declarando e inicializando Variáveis

A seguir, vemos como é feita a declaração de uma variável:

```
<tipo do dado> <nome> [= valor inicial];
```

nota: os valores colocados entre < > são obrigatórios, enquanto que os valores contidos entre [ ] são opcionais.

Aqui vemos um exemplo de programa que declara e inicializa algumas variáveis:

```
public class VariableSamples {
      public static void main( String[] args ){
            // declara uma variável com nome result e tipo boolean
            boolean result;

            // declara uma variável com nome option e tipo char
            char option;
            // atribui o símbolo C para a variável
            option = 'C';

            // declara uma variável com nome grade e tipo double
            // e a inicializa com o valor 0.0
            double grade = 0.0;
      }
}
```

Dicas de programação:
1. É sempre preferível que se inicialize uma variável assim que ela for declarada.
2. Use nomes com significado para suas variáveis. Se usar uma variável para armazenar a nota de um aluno, declare-a com o nome 'nota' e não simplesmente com uma letra aleatória 'x'.
3. É preferível declarar uma variável por linha, do que várias na mesma linha. Por exemplo:
int variavel1;
int variavel2;
E não:
int variavel1, variavel2;

9.2. Exibindo o valor de uma Variável

Para exibirmos em qualquer dispositivo de saída o valor de uma variável, fazemos uso dos seguintes comandos:

```
System.out.println()
System.out.print()
```

Aqui está um simples programa como exemplo:

```
public class OutputVariable {
      public static void main( String[] args ){
           int value = 10;
           char x;
           x = 'A';
           System.out.println(value);
           System.out.println("The value of x = " + x );
      }
}
```

A saída deste programa será a seguinte:

```
10
The value of x = A
```

9.3. System.out.println( ) e System.out.print( )

Qual é a diferença entre os comandos **System.out.println( )** e o **System.out.print( )**? O primeiro faz iniciar uma nova linha após ser exibido seu conteúdo, enquanto que o segundo não.

Considere as seguintes instruções:

```
System.out.print("Hello ");
System.out.print("world!");
```

Essas instruções apresentarão a seguinte saída:

```
Hello world!
```

Considere as seguintes:

```
System.out.println("Hello ");
System.out.println("world!");
```

Estas apresentarão a seguinte saída:

```
Hello
world!
```

9.4. Referência de Variáveis e Valor das Variáveis

Iremos diferenciar os dois tipos de variáveis suportados pelo Java. Estes podem ser de **referência** ou de **valor**. **As variáveis de "valor", ou primitivas,** são aquelas que armazenam dados no exato espaçode memória onde a variável está. **As variáveis de referência** são aquelas que armazenam o endereço de memória onde o dado está armazenado. Ao declarar uma variável de certa classe (variável de classe), se declara uma variável de referência a um objeto daquela classe. Por exemplo, vamos supor que se tenha estas duas variáveis do tipo **int** e da classe **String.**

```
int num = 10;
String nome = "Hello";
```

Suponha que o quadro abaixo represente a memória do computador, com seus endereços de memória, o nome das variáveis e os tipos de dados que ele pode suportar.

Endereço de memória Nome da variável Dado

```
1001 num 10
: :
1563 nome Endereço (2000)
:
::
2000 "Hello"
```

A variável (do tipo int) **num** o dado é o atual valor contido por ela e, a referência da variável (do tipo string) **nome** somente é armazenado o endereço de memória que contém o valor da variável.

10. Operadores

Em Java temos diferentes tipos de operadores. Existem **operadores aritméticos**, **operadores relacionais**, **operadores lógicos** e **operadores condicionais**. Estes operadores obedecem a uma ordem de precedência para que o compilador saiba qual operação executar primeiro, no caso de uma sentença possuir grande variedade destes.

10.1. Operadores Aritméticos

Aqui temos a lista dos operadores aritméticos que podem ser utilizados na criação de expressões matemáticas:

Operador	*Uso*	*Descrição*
*	op1 * op2	Multiplica op1 por op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Resto da divisão de op1 por op2
-	op1 - op2	Subtrai op2 de op1
+	op1 + op2	Soma op1 e op2

Tabela 4: Operadores aritméticos e suas funções

Aqui temos um programa que exemplifica o uso destes operadores:

```
public class ArithmeticDemo {
    public static void main(String[] args) {
        // alguns números
        int i = 37;
        int j = 42;
        double x = 27.475;
        double y = 7.22;
        System.out.println("Variables values...");
        System.out.println(" i = " + i);
        System.out.println(" j = " + j);
        System.out.println(" x = " + x);
```

```
            System.out.println(" y = " + y);

            // adição dos números
            System.out.println("Adding...");
            System.out.println(" i + j = " + (i + j));
            System.out.println(" x + y = " + (x + y));

            // subtração dos números
            System.out.println("Subtracting...");
            System.out.println(" i - j = " + (i - j));
            System.out.println(" x - y = " + (x - y));

            // multiplicação dos números
            System.out.println("Multiplying...");
            System.out.println(" i * j = " + (i * j));
            System.out.println(" x * y = " + (x * y));

            // divisão dos números
            System.out.println("Dividing...");
            System.out.println(" i / j = " + (i / j));
            System.out.println(" x / y = " + (x / y));

            // resto da divisão
            System.out.println("Comuting the remainder...");
            System.out.println(" i % j = " + (i % j));
            System.out.println(" x % y = " + (x % y));

            // misturando operações
            System.out.println("Mixing types...");
            System.out.println(" j + y = " + (j + y));
            System.out.println(" i * x = " + (i * x));
        }
    }
```

e como saída, temos:

```
Variables values...
    i = 37
    j = 42
    x = 27.475
    y = 7.22
    Adding...
    i + j = 79
    x + y = 34.695
```

```
Subtracting...
i - j = -5
x - y = 20.255
Multiplying...
i * j = 1554
x * y = 198.37
Dividing...
i / j = 0
x / y = 3.8054
Comuting the remainder...
i % j = 37
x % y = 5.815
Mixing types...
j + y = 49.22
i * x = 1016.58
```

Nota: Quando um número de tipo inteiro e um outro de número fracionário são usados numa única operação, o resultado será dado pela variável de maior tipo, no caso, valor de número fracionário. O número inteiro é implicitamente convertido para o número fracionário antes da operação ter início.

11. Operadores de Incremento e Decremento

Além dos operadores aritméticos básicos, Java dá suporte ao operador unário de incremento (++) e ao operador unário de decremento (--). Operadores de incremento ou decremento aumentam ou diminuem em 1 o valor da variável. Por exemplo, a expressão,

```
count = count + 1; // incrementa o valor de count em 1
```

é equivalente a,

```
count++;
```

Operador	*Uso*	*Descrição*
++	op++	Incrementa op em 1; Avalia a expressão antes do valor ser acrescido
++	++op	Incrementa op em 1; Incrementa o valor antes da expressão ser avaliada
--	op--	Decrementa op em 1; Avalia a expressão antes do valor ser decrescido
--	--op	Decrementa op em 1; Decrementa op em 1 antes da expressão ser avaliada

Tabela 5: Operadores de incremento e decremento

Como visto na tabela acima, os operadores de incremento e decremento podem ser usadostanto antes como após o operando. E sua utilização dependerá disso. Quando usado antes do operando, provoca acréscimo ou decréscimo de seu valor antes da avaliação da expressão em que ele aparece. Por exemplo:

```
int i = 10,
int j = 3;
int k = 0;
k = ++j + i; //resultará em k = 4+10 = 14
```

Quando utilizado depois do operando, provoca, na variável, acréscimo ou decréscimo do seuvalor após a avaliação da expressão na qual ele aparece. Por exemplo:

```
int i = 10,
int j = 3;
int k = 0;
k = j++ + i; //resultará em k = 3+10 = 13
```

Dicas de programação:

1. Mantenha sempre as operações incluindo operadores de incremento ou decremento de forma simples e de fácil compreensão.

12. Operadores Relacionais

Os operadores relacionais são usados para comparar dois valores e determinar orelacionamento entre eles. A saída desta avaliação será fornecida com um valor **lógico**: true ou false.

Operador	*Uso*	*Descrição*
>	op1 > op2	op1 é maior do que op2
>=	op1 >= op2	op1 é maior ou igual a op2
<	op1 < op2	op1 é menor do que op2
<=	op1 <= op2	op1 é menor ou igual a op2
==	op1 == op2	op1 é igual a op2
!=	op1 != op2	op1 não igual a op2

Tabela 6: Operadores relacionais

O programa a seguir, mostra a utilização destes operadores:

```
public class RelationalDemo {
    public static void main(String[] args) {
        // alguns números
        int i = 37;
        int j = 42;
        int k = 42;
        System.out.println("Variables values...");
        System.out.println(" i = " + i);
        System.out.println(" j = " + j);
        System.out.println(" k = " + k);

        // maior que
        System.out.println("Greater than...");
        System.out.println(" i > j = " + (i > j)); //false
        System.out.println(" j > i = " + (j > i)); //true
        System.out.println(" k > j = " + (k > j)); //false

        // maior ou igual a
        System.out.println("Greater than or equal to...");
        System.out.println(" i >= j = " + (i >= j)); //false
        System.out.println(" j >= i = " + (j >= i)); //true
        System.out.println(" k >= j = " + (k >= j)); //true

        // menor que
        System.out.println("Less than...");
        System.out.println(" i < j = " + (i < j)); //true
        System.out.println(" j < i = " + (j < i)); //false
        System.out.println(" k < j = " + (k < j)); //false

        // menor ou igual a
        System.out.println("Less than or equal to...");
        System.out.println(" i <= j = " + (i <= j)); //true
        System.out.println(" j <= i = " + (j <= i)); //false
        System.out.println(" k <= j = " + (k <= j)); //true

        // igual a
        System.out.println("Equal to...");
        System.out.println(" i == j = " + (i == j)); //false
        System.out.println(" k == j = " + (k == j)); //true

        // diferente
        System.out.println("Not equal to...");
        System.out.println(" i != j = " + (i != j)); //true
        System.out.println(" k != j = " + (k != j)); //false
```

```
        }
    }
```

A seguir temos a saída deste programa:

```
Variables values...
```

i = 37
j = 42
k = 42

```
Greater than...
```

i > j = false
j > i = true
k > j = false

```
Greater than or equal to...
```

i >= j = false
j >= i = true
k >= j = true

```
Less than...
```

i < j = true
j < i = false
k < j = false

```
Less than or equal to...
```

i <= j = true
j <= i = false
k <= j = true

```
Equal to...
```

i == j = false
k == j = true

```
Not equal to...
```

i != j = true
k != j = false

13. Operadores Lógicos

Operadores lógicos avaliam um ou mais operandos **lógicos** que geram um único valor final **true** ou **false** como resultado da expressão. São seis os operadores lógicos: && (e lógico), & (e binário), || (ou lógico), | (ou binário), ^ (ou exclusivo binário) e ! (negação).

A operação básica para um operador lógico é:

```
x1 op x2
```

Onde x1 e x2 podem ser expressões, variáveis ou constantes lógicas, e op pode tanto ser &&, &, ||, | ou ^ .

13.1. && (e lógico) e & (e binário)

x1	x2	Resultado
VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO	FALSO	FALSO
FALSO	VERDADEIRO	FALSO
FALSO	FALSO	FALSO

Tabela 7: Tabela para && e &

A diferença básica do operador && para & é que o && suporta uma avaliação de curto-circuito (ou avaliação parcial), enquanto que o & não. O que isso significa?

Dado o exemplo:

```
exp1 && exp2
```

o operador **e lógico** irá avaliar a expressão exp1, e, imediatamente, retornará um valor **false** se a operação exp1 for falsa. Se a expressão exp1 resultar em um valor **false** o operador nunca avaliará a expressão exp2, pois o valor de toda a expressão será falsa mesmo que o resultado isolado de exp2 seja verdadeiro. Já o operador & sempre avalia as duas partes da expressão, mesmo que a primeira tenha o valor **false**.

O programa a seguir, mostra a utilização destes operadores:

```
public class TestAND {
        public static void main( String[] args ) {
                int i = 0;
                int j = 10;
                boolean test = false;

                // demonstração do operador &&
                test = (i > 10) && (j++ > 9);
                System.out.println(i);
                System.out.println(j);
                System.out.println(test);

                // demonstração do operador &
                test = (i > 10) & (j++ > 9);
                System.out.println(i);
                System.out.println(j);
```

```
            System.out.println(test);
        }
    }
```

Como resultado, o programa produzirá a seguinte saída:

```
0
10
false
0
11
false
```

Note que o comando j++, na linha contendo &&, nunca será executado, pois o operador não o avalia, visto que a primeira parte da expressão (i>10) retorna um valor booleano **false.**

13.2. || (ou lógico) e | (ou binário)

x1	*x2*	*Resultado*
VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO	FALSO	VERDADEIRO
FALSO	VERDADEIRO	VERDADEIRO
FALSO	FALSO	FALSO

Tabela 8: Tabela para || e |

A diferença básica entre os operadores || e |, é que, semelhante ao operador &&, o || também suporta a avaliação parcial. O que isso significa?

Dada a expressão,

```
exp1 || exp2
```

o operador **ou lógico** irá avaliar a expressão exp1, e, imediatamente, retornará um valorlógico **true** para toda a expressão se a primeira parte for avaliada como verdadeira. Se a expressão exp1 resultar em **verdadeira** a segunda parte exp2 nunca será avaliada, pois o valor final da expressão será **true** independentemente do resultado da segunda expressão.

O programa a seguir, mostra a utilização destes operadores:

```
public class TestOR {
    public static void main( String[] args ){
        int i = 0;
        int j = 10;
        boolean test = false;

        // demonstração do operador ||
```

```
            test = (i < 10) || (j++ > 9);
            System.out.println(i);
            System.out.println(j);
            System.out.println(test);

            // demonstração do operador |
            test = (i < 10) | (j++ > 9);
            System.out.println(i);
            System.out.println(j);
            System.out.println(test);
        }
    }
```

Como resultado, o programa produzirá a seguinte saída:

```
0
10
true
0
11
true
```

Note que a expressão j++ nunca será avaliada na instrução que usa o operador ||, pois a primeira parte da expressão (i<10) já retorna **true** como valor final da expressão.

13.3. ^ (ou exclusivo binário)

x1	*x2*	*Resultado*
VERDADEIRO	VERDADEIRO	FALSO
VERDADEIRO	FALSO	VERDADEIRO
FALSO	VERDADEIRO	VERDADEIRO
FALSO	FALSO	FALSO

Tabela 9: Tabela para o operador ^

O resultado de uma expressão usando o operador **ou exclusivo binário** terá um valor **true** somente se uma das expressões for verdadeira e a outra falsa. Note que ambos os operandos são necessariamente avaliados pelo operador ^ .

O programa a seguir, mostra a utilização deste operador:

```
public class TestXOR {
    public static void main( String[] args ){
        boolean val1 = true;
        boolean val2 = true;
```

```
            System.out.println(val1 ^ val2);
            val1 = false;
            val2 = true;
            System.out.println(val1 ^ val2);
            val1 = false;
            val2 = false;
            System.out.println(val1 ^ val2);
            val1 = true;
            val2 = false;
            System.out.println(val1 ^ val2);
        }
}
```

Como resultado, o programa produzirá a seguinte saída:

```
false
true
false
true
```

13.4. ! (negação)

x1	Resultado
VERDADEIRO	FALSO
FALSO	VERDADEIRO

Tabela 10: Tabela para o operador !

O operador de **negação** inverte o resultado lógico de uma expressão, variável ou constante, ou seja, o que era verdadeiro será falso e vice-versa. O programa a seguir, mostra a utilização deste operador:

```
public class TestNOT {
        public static void main( String[] args ){
            boolean val1 = true;
            boolean val2 = false;
            System.out.println(!val1);
            System.out.println(!val2);
        }
}
```

Como resultado, o programa produzirá a seguinte saída:

```
false
true
```

14. Operador Condicional (?:)

O operador **condicional** é também chamado de operador **ternário**. Isto significa que ele tem 3 argumentos que juntos formam uma única expressão condicional. A estrutura de uma expressão utilizando um operador condicional é a seguinte:

```
exp1?exp2:exp3
```

Onde exp1 é uma expressão lógica que deve retornar **true** ou **false**. Se o valor de exp1 for verdadeiro, então, o resultado será a expressão exp2, caso contrário, o resultado será exp3. O programa a seguir, mostra a utilização deste operador:

```
public class ConditionalOperator {
        public static void main( String[] args ){
                String status = "";
                int grade = 80;

                //status do aluno
                status = (grade >= 60)?"Passed":"Fail";

                //print status
                System.out.println( status );
        }
}
```

Como resultado, o programa produzirá a seguinte saída:

```
Passed
```

Veremos na Figura 2 um fluxograma que demonstra como o operador **condicional** funciona.

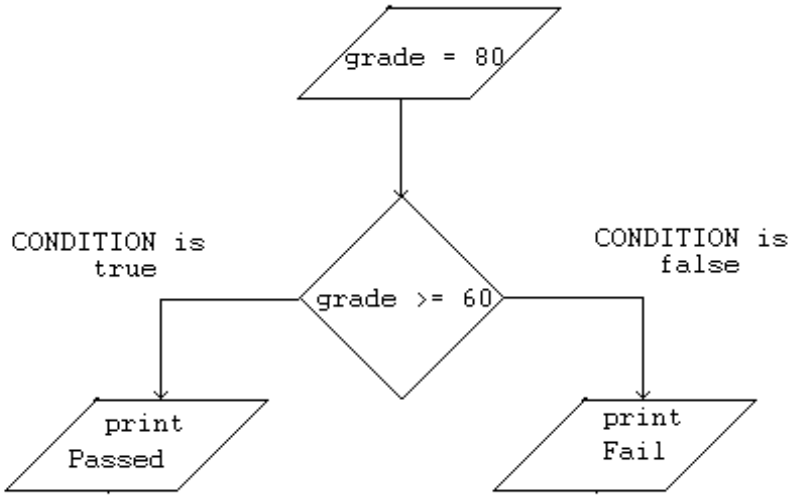


Figura 2: Fluxograma utilizando o operador **condicional**

Veremos outro programa que também utiliza o operador condicional:

```
public class ConditionalOperator {
        public static void main( String[] args ){
                int score = 0;
                char answer = 'a';
                score = (answer == 'a') ? 10 : 0;
                System.out.println("Score = " + score );
        }
}
```

Como resultado, o programa produzirá a seguinte saída:

```
Score = 10
```

15. Precedência de Operadores

A precedência serve para indicar a ordem na qual o compilador interpretará os diferentes tiposde operadores, para que ele sempre tenha como saída um resultado coerente e não ambíguo.

Ordem	Operador
1	( ) parênteses
2	++ pós-incremento e -- pós-decremento
3	++ pré-incremento e -- pré-decremento
4	! negação lógica
5	* multiplicação e / divisão
6	% resto da divisão
7	+ soma e – subtração
8	< menor que, <= menor ou igual, > maior que e >= maior ou igual
9	== igual e != não igual
10	& e binário
11	| ou binário
12	^ ou exclusivo binário
13	&& e lógico
14	|| ou lógico
15	?: condicional
16	= atribuição

TabelTabela 11: Precedência de operadores

No caso de dois operadores com mesmo nível de precedência, terá prioridade o que estivermais à esquerda da expressão. Dada uma expressão complexa como:

```
6%2*5+4/2+88-10
```

O ideal seria fazer uso de parênteses para reescrevê-la de maneira mais clara:

```
((6%2)*5)+(4/2)+88-10
```

Dicas de programação:
1. Para evitar confusão na avaliação de suas expressões matemáticas, deixe-as o mais simples possível e use parênteses.

16. Exercícios

16.1. Declarar e mostrar variáveis

Dada a tabela abaixo, declare as variáveis que se seguem de acordo com seus tipos correspondentes e valores iniciais. Exiba o nomes e valor das variáveis.

Nome das Variáveis	*Tipo do dado*	*Valor inicial*
number	integer	10
letter	character	a
result	boolean	true
str	String	hello

O resultado esperado do exercício é:

number = 10
letter = a
result = true
str = hello

16.2. Obter a média entre três números

Crie um programa que obtenha a média de 3 números. Considere o valor para os três númeroscomo sendo 10, 20 e 45. O resultado esperado do exercício é:

número 1 com o valor 10
número 2 com o valor 20
número 3 com o valor 45
A média é 25

16.3. Exibir o maior valor

Dados três números, crie um programa que exiba na tela o maior dentre os números informados. Use o operador **?:** que já foi estudado nesta sessão (**dica:** será necessário utilizar dois operadores **?:** para se chegar ao resultado). Por exemplo, dados os números 10, 23 e 5, o resultado esperado do exercício deve ser:

número 1 com o valor 10
número 2 com o valor 23
número 3 com o valor 5
O maior número é 23

16.4. Precedência de operadores

Dadas as expressões abaixo, reescreva-as utilizando parênteses de acordo com a forma comoelas são interpretadas pelo compilador.

1. a / b ^ c ^ d – e + f – g * h + i
2. 3 * 10 *2 / 15 – 2 + 4 ^ 2 ^ 2

3. r ^ s * t / u – v + w ^ x – y++

Lição 3

1. Objetivos

Agora que já estudamos alguns conceitos básicos e escrevemos alguns códigos simples, vamos fazer as aplicações ficarem mais interativas começando com a captura de dados digitados pelo usuário. Nesta lição, discutiremos três modos de obter dados de entrada (input). O primeiro é através do uso da classe BufferedReader do pacote java.util; o segundo, através do uso da nova classe Scanner no mesmo pacote; e, por fim, envolveremos a utilização da interface gráfica utilizando JOptionPane.

Ao final desta lição, o estudante será capaz de:

- Criar códigos para a captura de dados pelo teclado.
- Usar a classe BufferedReader para captura, através de uma janela de console, dos dados digitados no teclado.
- Utilizar a classe Scanner para captura, através de uma janela de console, dos dados digitados no teclado.
- Utilizar a classe JOptionPane para captura, através da uma interface gráfica, dos dados digitados no teclado.

2. BufferedReader para capturar dados

Primeiramente, utilizaremos a classe BufferedReader do pacote java.io para capturar dados de entrada através do teclado. Passos para capturar os dados digitados, tomemos por base o programa visto na lição anterior:

1. Digite a seguinte instrução no início do programa:

```
import java.io.*;
```

2. Adicione as seguintes instruções no corpo do método **main**:

```
BufferedReader dataIn = new BufferedReader(
new InputStreamReader(System.in));
```

3. Declare uma variável temporária do tipo String para gravar os dados digitados pelo usuário e chame o método readLine() que vai capturar linha por linha do que o usuário digitar.
Isso deverá ser escrito dentro de um bloco **try- catch** para tratar possíveis exceções.

```
try {
      String temp = dataIn.readLine();
} catch (IOException e) {
      System.out.println("Error in getting input");
}
```

Abaixo, segue o programa completo:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class GetInputFromKeyboard {
      public static void main(String[] args) {
            BufferedReader dataIn = new BufferedReader(new InputStreamReader(System.in));
            String name = "";
            System.out.print("Please Enter Your Name:");
```

```
                try {
                        name = dataIn.readLine();
                } catch (IOException e) {
                        System.out.println("Error!");
                }
                System.out.println("Hello " + name +"!");
        }
}
```

Faremos uma análise deste programa linha por linha:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

Estas linhas acima mostram que estamos utilizando as classes BufferedReader, InputStreamReader e IOException cada qual dentro do pacote java.io. Essas APIs ou Interfaces de Programação de Aplicações (Application Programming Interface) contêm centenas de classes pré-definidas que se pode usar nos programas. Essas classes são organizadas dentro do que chamamos de pacotes.

Pacotes contêm classes que se relacionam com um determinado propósito. No exemplo, o pacote java.io contém as classes que permitem capturar dados de entrada e saída. Estas linhas poderiam ser reescritas da seguinte forma:

```
import java.io.*;
```

que importará todas as classes encontradas no pacote java.io, deste modo é possível utilizar todas classes desse pacote no programa.

As próximas linhas:

```
public class GetInputFromKeyboard {
public static void main( String[] args ) {
```

já foram discutidas na lição anterior. Isso significa que declaramos uma classe nomeada **GetInputFromKeyboard** e, em seguida, iniciamos o método principal (main).

Na instrução:

```
BufferedReader dataIn = new BufferedReader(new InputStreamReader(System.in));
```

declaramos a variável dataIn do tipo BufferedReader. Não se preocupe com o significado da sintaxe, pois será abordado mais à frente.

A seguir, declaramos a variável **name** do tipo String:

```
String name = "";
```

na qual armazenaremos a entrada de dados digitada pelo usuário. Note que foi inicializada como uma String vazia "". É uma boa prática de programação inicializar as variáveis quando declaradas.

Na próxima instrução, solicitamos que o usuário escreva um nome:

```
System.out.print("Please Enter Your Name:");
```

As seguinte linhas definem um bloco **try- catch:**

```
try {
      name = dataIn.readLine();
} catch (IOException e) {
      System.out.println("Error!");
}
```

que asseguram, caso ocorram exceções serão tratadas. Falaremos sobre o tratamento de exceções na última parte deste curso. Por hora, é necessário adicionar essas linhas para utilizar o método **readLine()** e receber a entrada de dados do usuário.

Em seguida:

```
name = dataIn.readLine();
```

capturamos a entrada dos dados digitados pelo usuário e as enviamos para a variável String criada anteriormente. A informação é guardada na variável **name**.

Como última instrução:

```
System.out.println("Hello " + name + "!");
```

montamos a mensagem final para cumprimentar o usuário.

3. Classe Scanner para capturar dados

Vimos uma maneira para obter dados de entrada através do teclado. O JDK 5.0 lançou uma nova classe chamada **Scanner** que engloba diversos métodos para facilitar este serviço.

Abaixo, segue o programa completo utilizando esta classe:

```
import java.util.Scanner;
public class GetInputFromScanner{
      public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            System.out.println("Please Enter Your Name:");
            String name = sc.next();
            System.out.println("Hello " + name +"!");
      }
}
```

Compare-o com o programa visto anteriormente. Percebe-se que fica mais simples conseguir ames-ma funcionalidade. Inicialmente, definimos a chamada ao pacote que contém a classe **Scanner**:

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

Em seguida, as instruções que define a classe e o método main:

```
public class GetInputFromScanner{
      public static void main(String[] args) {
```

Definimos uma variável, denominada **sc**, que será criada a partir da classe **Scanner** e direcionada para a entrada padrão:

```
Scanner sc = new Scanner(System.in);
```

De forma semelhante, mostramos uma mensagem solicitando informação do usuário:

```
System.out.println("Please Enter Your Name:");
```

Utilizamos a variável **sc** para chamarmos o método que fará o recebimento dos dados digitados:

```
String name = sc.nextLine();
```

A classe **Scanner** possui diversos métodos que podem ser utilizados para realizar este serviço. Os principais métodos que podemos utilizar, neste caso, são:

Método	**Finalidade**
next()	Aguarda uma entrada em formato String
nextInt()	Aguarda uma entrada em formato Inteiro
nextByte()	Aguarda uma entrada em formato Inteiro
nextLong()	Aguarda uma entrada em formato Inteiro Longo
nextFloat()	Aguarda uma entrada em formato Número Fracionário
nextDouble()	Aguarda uma entrada em formato Número Fracionário

*Tabela 1: Métodos da Classe **Scanner** para obter dados*

Por fim, mostramos o resultado e encerramos o método main e a classe:

```
        System.out.println("Hello " + name +"!");
        }
}
```

4. Utilizando a JOptionPane para receber dados

Um outro modo de receber os dados de entrada é utilizar a classe **JOptionPane**, que pertenceao pacote **javax.swing**. A **JOptionPane** possui métodos que conseguem criar caixas de diálogo na qual o usuário pode informar ou visualizar algum dado.

Dado o seguinte código:

```
import javax.swing.JOptionPane;
public class GetInputFromKeyboard {
      public static void main( String[] args ){
            String name = "";
            name = JOptionPane.showInputDialog("Please enter your name");
            String msg = "Hello " + name + "!";
            JOptionPane.showMessageDialog(null, msg);
      }
}
```

esta classe apresentará o seguinte resultado:

Figura 1: Aguardando dados no JOptionPane

Figura 2: Digitando florence no JOptionPane

‘

A primeira instrução:

```
import javax.swing.JOptionPane;
```

mostra que estamos importando a classe **JOptionPane** do pacote **javax.swing**. Poderíamos, de forma semelhante, escrever estas instruções do seguinte modo:

```
import javax.swing.*;
```

A instrução seguinte:

```
name = JOptionPane.showInputDialog("Please enter your name");
```

cria uma caixa de entrada que exibirá um diálogo com uma mensagem, um campo de texto para receber os dados do usuário e um botão OK, conforme mostrado na **figura 1**. O resultado será armazenado na variável do tipo String **name**.

Na próxima instrução, criamos uma mensagem de cumprimento, que ficará armazenada na variável **msg**:

```
String msg = "Hello " + name + "!";
```

Finalizando a classe, exibiremos uma janela de diálogo que conterá a mensagem e o botão de OK, conforme mostrado na **figura 3**.

```
JOptionPane.showMessageDialog(null, msg);
```

5. Exercícios

5.1. As 3 palavras (versão Console)

Utilizando a classe **BufferedReader** ou **Scanner**, capture três palavras digitadas pelo usuário e mostre-as como uma única frase na mesma linha. Por exemplo:

```
Palavra 1: Goodbye
Palavra 2: and
Palavra 3: Hello
Goodbye and Hello
```

5.2. As 3 palavras (versão Interface Gráfica)

Utilizando a classe **JOptionPane**, capture palavras em três caixas de diálogos distintas e mostre-as como uma única frase. Por exemplo:

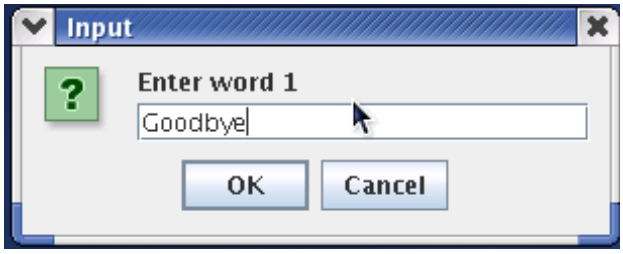


Figura 4: Primeira Palavra

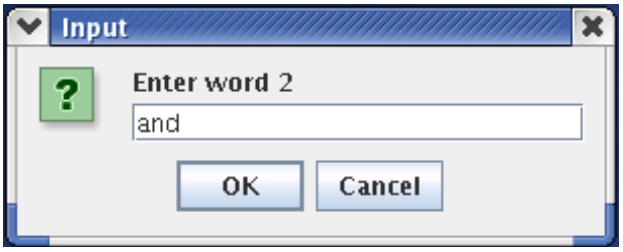


Figura 5: Segunda Palavra

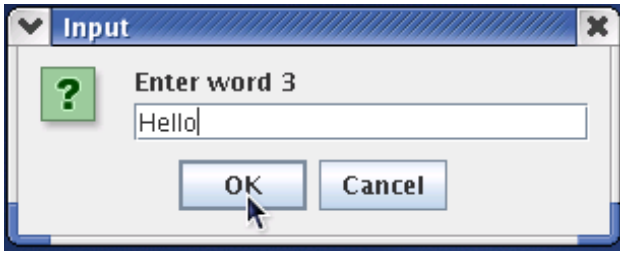


Figura 6: Terceira Palavra

Figura 7: Mostrando a Mensagem

Lição 4

1. Objetivos

Nas lições anteriores, foram mostrados programas seqüenciais, onde as instruções foram executadas uma após a outra de forma fixa. Nesta lição, discutiremos estruturas de controle que permitem mudar a ordem na qual as instruções são executadas.

Ao final desta lição, o estudante será capaz de:

- Usar estruturas de controle de decisão (**if** e **switch**) que permitem a seleção de partes específicas do código para execução
- Usar estruturas de controle de repetição (**while**, **do- while** e **for**) que permitem a repetição da execução de partes específicas do código
- Usar declarações de interrupção (**break**, **continue** e **return**) que permitem o redirecionamento do fluxo do programa

2. Estruturas de controle de decisão

Estruturas de controle de decisão são instruções em linguagem Java que permitem que blocos específicos de código sejam escolhidos para serem executados, redirecionando determinadas partes do fluxo do programa.

2.1. Declaração if

A declaração **if** especifica que uma instrução ou bloco de instruções seja executado se, esomente se, uma expressão lógica for verdadeira. A declaração **if** possui a seguinte forma:

```
if (expressão_lógica)
      instrução;
ou:
if (expressão_lógica) {
      instrução1;
      instrução2
      ...
}
```

onde, **expressão_lógica** representa uma expressão ou variável lógica.

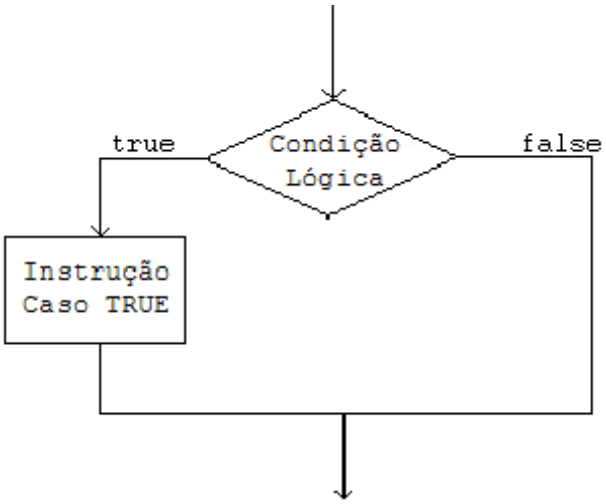


Figura 1: Fluxograma da declaração

If

Por exemplo, dado o trecho de código:

```
int grade = 68;
if (grade > 60) System.out.println("Congratulations!");
```

ou:

```
int grade = 68;
if (grade > 60) {
      System.out.println("Congratulations!");
      System.out.println("You passed!");
}
```

Dicas de programação:

1. Expressão lógica é uma declaração que possui um valor lógico. Isso significa que a execução desta expressão deve resultar em um valor true ou false.
2. Coloque as instruções de forma que elas façam parte do bloco **if**. Por exemplo:

```
if (expressão_lógica) {
      // instrução1;
      // instrução2;
}
```

2.2. Declaração if- else

A declaração **if- else** é usada quando queremos executar determinado conjunto de instruçõesse a condição for verdadeira e outro conjunto se a condição for falsa.

Possui a seguinte forma:

```
if (expressão_lógica)
      instrução_caso_verdadeiro;
else
      instrução_caso_falso;
```

Também podemos escrevê-la na forma abaixo:

```
if (expressão_lógica) {
      instrução_caso_verdadeiro1;
      instrução_caso_verdadeiro2;
      ...
} else {
      instrução_caso_falso1;
      instrução_caso_falso2;
      ...
}
```

Por exemplo, dado o trecho de código:

```
int grade = 68;
if (grade > 60)
      System.out.println("Congratulations! You passed!");
else
      System.out.println("Sorry you failed");
```

ou:

```
int grade = 68;
if (grade > 60) {
      System.out.print("Congratulations! ");
      System.out.println("You passed!");
} else {
      System.out.print("Sorry ");
      System.out.println("you failed");
}
```

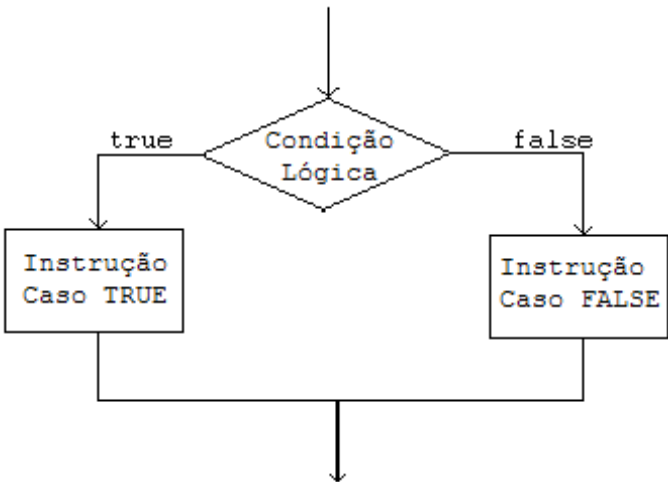


*Figura 2: Fluxograma da declaração **if- else***

Dicas de programação:

1. Para evitar confusão, sempre coloque a instrução ou instruções contidas no bloco **if** ou **if- else** entre chaves { }.
2. Pode-se ter declarações **if- else** dentro de declarações **if- else**, por exemplo:

```
if (expressão_lógica) {
      if (expressão_lógica) {
      ...
      } else {
      ...
      }
} else {
      ...
}
```

2.3. Declaração if- else- if

A declaração **else** pode conter outra estrutura **if- else**. Este cascateamento de estruturaspermite ter decisões lógicas muito mais complexas.

A declaração **if- else- if** possui a seguinte forma:

```
if (expressão_lógica1)
      instrução1;
else if(expressão_lógica2)
      instrução2;
else
      instrução3;
```

Podemos ter várias estruturas **else- if** depois de uma declaração **if**. A estrutura **else** é opcional e pode ser omitida. No exemplo mostrado acima, se a **expressão_lógica1** é verdadeira, o programa executa a **instrução1** e salta as outras instruções. Caso contrário, se a **expressão_lógica1** é falsa, o fluxo de controle segue para a análise da **expressão_lógica2**.

Se esta for verdadeira, o programa executa a **instrução2** e salta a **instrução3**. Caso contrário, se a **expressão_lógica2** é falsa, então a **instrução3** é executada.

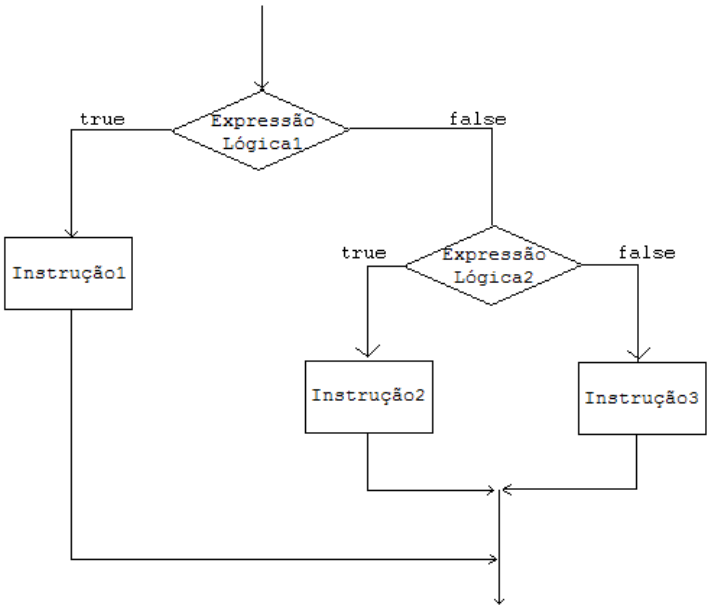


*Figura 3: Fluxograma da declaração **if- else- if***

Observe um exemplo da declaração **if- else- if** no seguinte trecho de código:

```
	public class Grade {
		public static void main( String[] args ) {
			double grade = 92.0;
			if (grade >= 90) {
			System.out.println("Excellent!");
			} else if((grade < 90) && (grade >= 80)) {
			System.out.println("Good job!");
			} else if((grade < 80) && (grade >= 60)) {
			System.out.println("Study harder!");
			} else {
			System.out.println("Sorry, you failed.");
		}
	}
}
```

2.4. Erros comuns na utilização da declaração if

1. A condição na declaração **if** não avalia um valor lógico. Por exemplo:

```
	// ERRADO
	int number = 0;
	if (number) {
		// algumas instruções aqui
	}
```

a variável **number** não tem valor lógico.

2. Usar = (sinal de atribuição) em vez de == (sinal de igualdade) para comparação. Por exemplo:

```
// ERRADO
int number = 0;
if (number = 0) {
// algumas instruções aqui
}
```

3. Escrever **elseif** em vez de **else if**.

```
// ERRADO
int number = 0;
if (number == 0) {
      // algumas instruções aqui
} elseif (number == 1) {
      // algumas instruções aqui
}
```

2.5. Declaração switch

Outra maneira de indicar uma condição é através de uma declaração **switch**. A construção **switch** permite que uma única variável inteira tenha múltiplas possibilidades de finalização.

A declaração **switch** possui a seguinte forma:

```
switch (variável_inteira) {
case valor1:
      instrução1; //
      instrução2; // bloco 1
      ... //
      break;
case valor2:
      instrução1; //
      instrução2; // bloco 2
      ... //
      break;
default:
      instrução1; //
      instrução2; // bloco n
      ... //
      break;
}
```

onde, **variável_inteira** é uma variável de tipo byte, short, char ou int. **valor1**, **valor2**, e assim por diante, são valores constantes que esta variável pode assumir.

Quando a declaração **switch** é encontrada, o fluxo de controle avalia inicialmente a **variável_inteira** e segue para o **case** que possui o valor igual ao da variável. O programa executa todas instruções a partir deste ponto, mesmo as do próximo **case**, até encontrar uma instrução **break**, que interromperá a execução do **switch**.

Se nenhum dos valores **case** for satisfeito, o bloco **default** será executado. Este é um bloco opcional. O bloco **default** não é obrigatório na declaração **switch**.

Notas:

1. Ao contrário da declaração **if**, múltiplas instruções são executadas sem a necessidade das chaves que determinam o início e término de bloco { }.

2. Quando um **case** for selecionado, todas as instruções vinculadas ao case serão executadas. Além disso, as instruções dos **case** seguintes também serão executadas.

3. Para prevenir que o programa execute instruções dos outros **case** subseqüentes, utilizamos a declaração **break** após a última instrução de cada **case**.

Dicas de Programação:

1. A decisão entre usar uma declaração **if** ou **switch** é subjetiva. O programador pode decidir com base na facilidade de entendimento do código, entre outros fatores.

2. Uma declaração **if** pode ser usada para decisões relacionadas a conjuntos, escalas de variáveis ou condições, enquanto que a declaração **switch** pode ser utilizada para situações que envolvam variável do tipo inteiro. Também é necessário que o valor de cada cláusula **case** seja único.

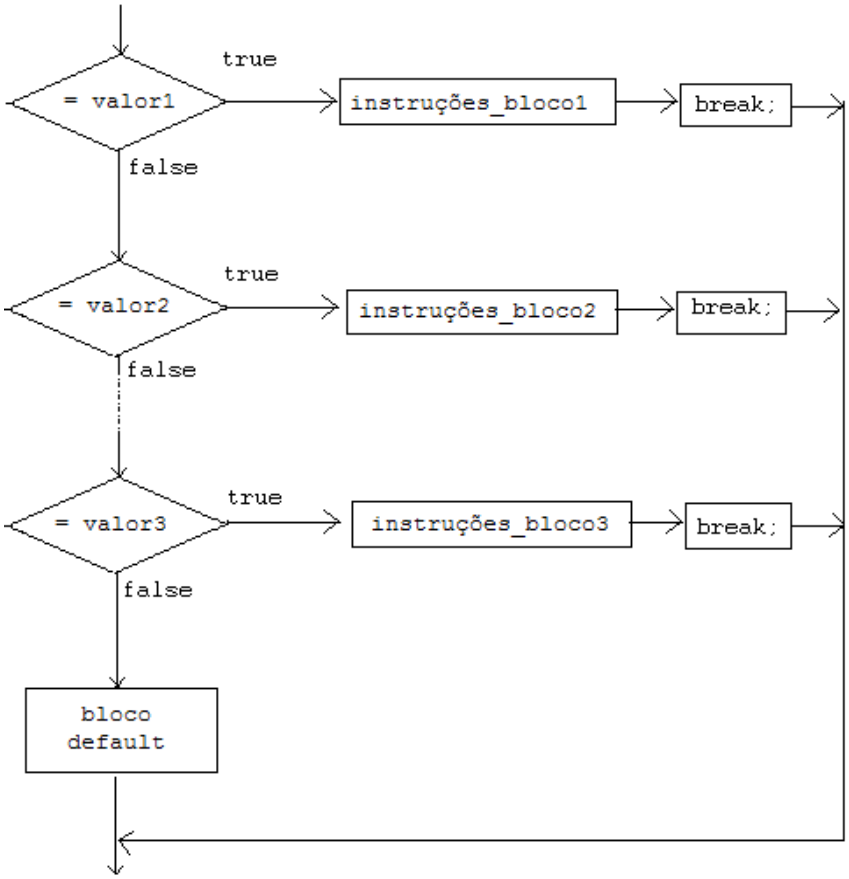


*Figura 4: Fluxograma da declaração **switch***

2.6. Exemplo para switch

```
public class Grade {
      public static void main(String[] args) {
      int grade = 92;
            switch(grade) {
            case 100:
                  System.out.println("Excellent!");
                  break;
            case 90:
                  System.out.println("Good job!");
                  break;
            case 80:
                  System.out.println("Study harder!");
                  break;
            default:
            System.out.println("Sorry, you failed.");
            }
      }
}
```

Compile e execute o programa acima e veremos que o resultado será:

```
Sorry, you failed.
```

pois a variável **grade** possui o valor 92 e nenhuma das opções **case** atende a essa condição. Note que para o caso de intervalos a declaração **if- else- if** é mais indicada.

3. Estruturas de controle de repetição

Estruturas de controle de repetição são comandos em linguagem Java que permitem executar partes específicas do código determinada quantidade de vezes. Existem 3 tipos de estruturas de controle de repetição: **while**, **do- while** e **for**.

3.1. Declaração while

A declaração **while** executa repetidas vezes um bloco de instruções enquanto uma determinada condição lógica for verdadeira.

A declaração **while** possui a seguinte forma:

```
while (expressão_lógica) {
instrução1;
instrução2;
...
}
```

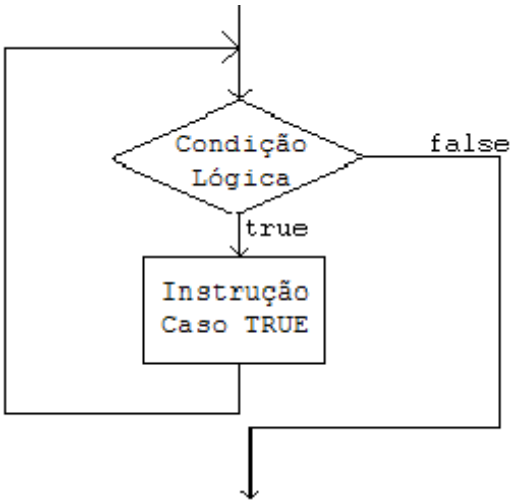


*Figura 5: Fluxograma da declaração **while***

As instruções contidas dentro do bloco **while** são executadas repetidas vezes enquanto o valorde **expressão_lógica** for verdadeira.

Por exemplo, dado o trecho de código:

```
int i = 4;
while (i > 0){
System.out.print(i);
i--;
}
```

O código acima irá imprimir 4321 na tela. Se a linha contendo a instrução **i- -** for removida,teremos uma repetição infinita, ou seja, um código que não termina. Portanto, ao usar laços **while**, ou qualquer outra estrutura de controle de repetição, tenha a certeza de utilizar uma estrutura de repetição que encerre em algum momento.

Abaixo, temos outros exemplos de declarações **while**:

Exemplo 1:

```
int x = 0;
while (x<10) {
System.out.println(x);
x++;
}
```

Exemplo 2:

```
// laço infinito
while (true)
System.out.println("hello");
```

Exemplo 3:

```
// a instrução do laço não será executada
while (false)
System.out.println("hello");
```

3.2. Declaração do- while

A declaração **do- while** é similar ao **while**. As instruções dentro do laço **do- while** serão executadas pelo menos uma vez.

A declaração **do- while** possui a seguinte forma:

```
do {
      instrução1;
      instrução2;
      ...
} while (expressão_lógica);
```

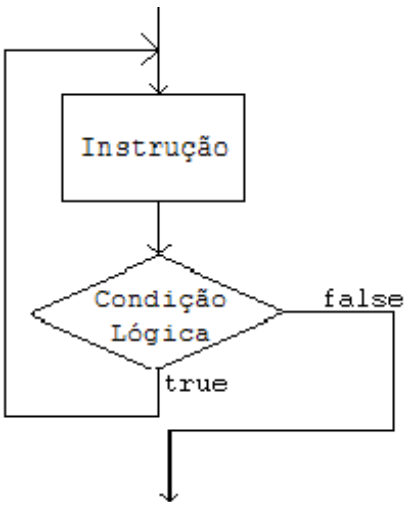


*Figura 6: Fluxograma da declaração **do- while***

Inicialmente, as instruções dentro do laço **do- while** são executadas. Então, a condição na**expressão_lógica** é avaliada. Se for verdadeira, as instruções dentro do laço **do- while** serão executadas novamente. A diferença entre uma declaração **while** e **do- while** é que, no laço **while**, a avaliação da expressão lógica é feita antes de se executarem as instruções nele contidas enquanto que, no laço **do- while**, primeiro se executam as instruções e depois realiza-se a avaliação da expressão lógica, ou seja, as instruções dentro em um laço **do- while** são executadas pelo menos uma vez.

Abaixo, temos alguns exemplos que usam a declaração **do- while**:

Exemplo 1:

```
int x = 0;
do {
System.out.println(x);
x++;
} while (x<10);
Este exemplo terá 0123456789 escrito na tela.
```

Exemplo 2:

```
// laço infinito
do {
System.out.println("hello");
} while(true);
```

Este exemplo mostrará a palavra **hello** escrita na tela infinitas vezes.

Exemplo 3:

```
// Um laço executado uma vez
do
System.out.println(“hello”);
while (false);
```

Este exemplo mostrará a palavra **hello** escrita na tela uma única vez.

Dicas de programação:

1. Erro comum de programação ao utilizar o laço **do- while** é esquecer o ponto-evírgula (;) após a declaração **while**.

```
do {
...
} while (boolean_expression) // ERRADO -> faltou ;
```

2. Como visto para a declaração **while**, tenha certeza que a declaração **do- while** poderá terminar em algum momento.

3.3. Declaração for

A declaração **for**, como nas declarações anteriores, permite a execução do mesmo código uma quantidade determinada de vezes.

A declaração **for** possui a seguinte forma:

```
for (declaração_inicial; expressão_lógica; salto) {
instrução1;
instrução2;
...
}
```

onde:

declaração_inicial – inicializa uma variável para o laço

expressão_lógica – compara a variável do laço com um valor limite

salto – atualiza a variável do laço

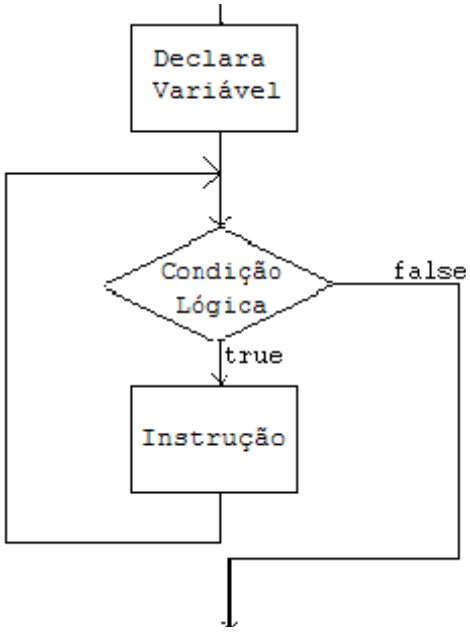


*Figura 7: Fluxograma da declaração **for***

Um exemplo para a declaração **for** é:

```
for (int i = 0; i < 10; i++) {
      System.out.print(i);
}
```

Neste exemplo, uma variável **i**, do tipo **int**, é inicializada com o valor zero. A expressão lógica "**i** é menor que 10" é avaliada. Se for verdadeira, então a instrução dentro do laço é executada. Após

isso, a expressão **i** terá seu valor adicionado em 1 e, novamente, a condição lógica será avaliada. Este processo continuará até que a condição lógica tenha o valor falso.

Este mesmo exemplo, utilizando a declaração **while**, é mostrado abaixo:

```
int i = 0;
while (i < 10) {
System.out.print(i);
i++;
}
```

4. Declarações de Interrupção

Declarações de interrupção permitem que redirecionemos o fluxo de controle do programa. A linguagem Java possui três declarações de interrupção. São elas: **break**, **continue** e **return**.

4.1. Declaração break

A declaração **break** possui duas formas: **unlabeled** (não identificada - vimos esta forma com a declaração **switch**) e **labeled** (identificada).

4.1.1. Declaração unlabeled break

A forma **unlabeled** de uma declaração **break** encerra a execução de um **switch** e o fluxo de controle é transferido imediatamente para o final deste. Podemos também utilizar a forma para terminar declarações **for**, **while** ou **do- while**.

Por exemplo:

```
String names[] = {"Beah", "Bianca", "Lance", "Belle",
"Nico", "Yza", "Gem", "Ethan"};
String searchName = "Yza";
boolean foundName = false;
for (int i=0; i < names.length; i++) {
      if (names[i].equals(searchName)) {
      foundName = true;
      break;
      }
}
if (foundName) {
      System.out.println(searchName + " found!");
} else {
      System.out.println(searchName + " not found.");
}
```

Neste exemplo, se a String “Yza” for encontrada, a declaração **for** será interrompida e o controle do programa será transferido para a próxima instrução abaixo da declaração **for**.

4.1.2. Declaração labeled break

A forma **labeled** de uma declaração **break** encerra o processamento de um laço que é identificado por um **label** especificado na declaração **break**.

Um **label**, em linguagem Java, é definido colocando-se um nome seguido de dois-pontos, como por exemplo:

```
teste:
```

esta linha indica que temos um **label** com o nome **teste**.

O programa a seguir realiza uma pesquisa de um determinado valor em um array bidimensional. Dois laços são criados para percorrer este array. Quando o valor é encontrado, um **labeled break** termina a execução do laço interno e retorna o controle para o laço mais externo.

```
int[][] numbers = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int searchNum = 5;
boolean foundNum = false;
searchLabel: for (int i=0; i<numbers.length; i++) {
      for (int j=0; j<numbers[i].length; j++) {
            if (searchNum == numbers[i][j]) {
                  foundNum = true;
                  break searchLabel;
            }
      } // final do laço j
} // final do laço i

if (foundNum) {
      System.out.println(searchNum + " found!");
} else {
      System.out.println(searchNum + " not found!");
}
```

A declaração **break**, ao terminar a declaração **for**, não transfere o controle do programa aofinal de seu laço, controlado pela variável **j**. O controle do programa segue imediatamente para a declaração **for** marcada com o **label**, neste caso, interrompendo o laço controlado pela variável **i**.

4.2. Declaração continue

A declaração **continue** tem duas formas: **unlabeled** e **labeled**. Utilizamos uma declaração **continue** para saltar a repetição atual de declarações **for**, **while** ou **do- while**.

4.2.1. Declaração unlabeled continue

A forma **unlabeled** salta as instruções restantes de um laço e avalia novamente a expressão lógica que o controla.

O exemplo seguinte conta a quantidade de vezes que a expressão "Beah" aparece no array.

```
String names[] = {"Beah", "Bianca", "Lance", "Beah"};
int count = 0;
for (int i=0; i < names.length; i++) {
      if (!names[i].equals("Beah")) {
      continue; // retorna para a próxima condição
}
      count++;
}
System.out.println(count + " Beahs in the list");
```

4.2.2. Declaração labeled continue

A forma **labeled** da declaração **continue** interrompe a repetição atual de um laço e salta para a repetição exterior marcada com o **label** indicado.

```
outerLoop: for (int i=0; i<5; i++) {
for (int j=0; j<5; j++) {
      System.out.println("Inside for(j) loop"); // mensagem1
      if (j == 2)
            continue outerLoop;
      }
      System.out.println("Inside for(i) loop"); // mensagem2
}
```

Neste exemplo, a mensagem 2 nunca será mostrada, pois a declaração **continue outerloop** interromperá este laço cada vez que **j** atingir o valor 2 do laço interno.

4.3. Declaração return

A declaração **return** é utilizada para sair de um método. O fluxo de controle retorna para a declaração que segue a chamada do método original. A declaração de retorno possui dois modos: o que retorna um valor e o que não retorna nada.

Para retornar um valor, escreva o valor (ou uma expressão que calcula este valor) depois da palavra chave **return**. Por exemplo:

```
return ++count;
```

ou

```
return "Hello";
```

Os dados são processados e o valor é devolvido de acordo com o tipo de dado do método. Quando um método não tem valor de retorno, deve ser declarado como **void**. Use a forma de **return** que não devolve um valor. Por exemplo:

```
return;
```

Abordaremos as declarações **return** nas próximas lições, quando falarmos sobre métodos.

5. Exercícios

5.1. Notas

Obtenha do usuário três notas de exame e calcule a média dessas notas. Reproduza a média dos três exames. Junto com a média, mostre também um :-) no resultado se a média for maior ou igual a 60; caso contrário mostre :-(

Faça duas versões deste programa:

1. Use a classe BufferedReader (ou a classe Scanner) para obter as notas do usuário, e System.out para mostrar o resultado.

2. Use JOptionPane para obter as notas do usuário e para mostrar o resultado.

5.2. Número por Extenso

Solicite ao usuário para digitar um número, e mostre-o por extenso. Este número deverá variar entre 1 e 10. Se o usuário introduzir um número que não está neste intervalo, mostre:

"número inválido".

Faça duas versões deste programa**:**

1. Use uma declaração **if- else- if** para resolver este problema

2. Use uma declaração **switch** para resolver este problema

5.3. Cem vezes

Crie um programa que mostre seu nome cem vezes. Faça três versões deste programa:

1. Use uma declaração **while** para resolver este problema

2. Use uma declaração **do- while** para resolver este problema

3. Use uma declaração **for** para resolver este problema

5.4. Potências

Receba como entrada um número e um expoente. Calcule este número elevado ao expoente.

Faça três versões deste programa:

1. Use uma declaração **while** para resolver este problema

2. Use uma declaração **do- while** para resolver este problema

3. Use uma declaração **for** para resolver este problema

Lição 5

1. Objetivos

Nesta lição, abordaremos Array em Java. Primeiro, definiremos o que é array e, então, discutiremos como declará-los e usá-los.

Ao final desta lição, o estudante será capaz de:

- Declarar e criar array
- Acessar elementos de um array
- Determinar o número de elementos de um array
- Declarar e criar array multidimensional

2. Introdução a Array

Em lições anteriores, discutimos como declarar diferentes variáveis usando os tipos de dados primitivos. Na declaração de variáveis, freqüentemente utilizamos um identificador ou um nome e um tipo de dados. Para se utilizar uma variável, deve-se chamá-la pelo nome que a identifica.

Por exemplo, temos três variáveis do tipo **int** com diferentes identificadores para cada variável:

```
int number1;
int number2;
int number3;
number1 = 1;
number2 = 2;
number3 = 3;
```

Como se vê, inicializar e utilizar variáveis pode torna-se uma tarefa tediosa, especialmente se elas forem utilizadas para o mesmo objetivo. Em Java, e em outras linguagens de programação, pode-se utilizar uma variável para armazenar e manipular uma lista de dados com maior eficiência. Este tipo de variável é chamado de **array**.

Um **array** armazena múltiplos itens de um mesmo tipo de dado em um bloco contínuo de memória, dividindo-o em certa quantidade de **posições**. Imagine um array como uma variável esticada – que tem um nome que a identifica e que pode conter mais de um valor para esta mesma variável.

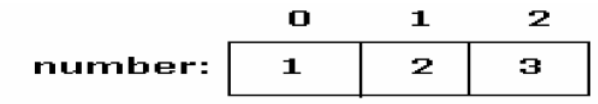
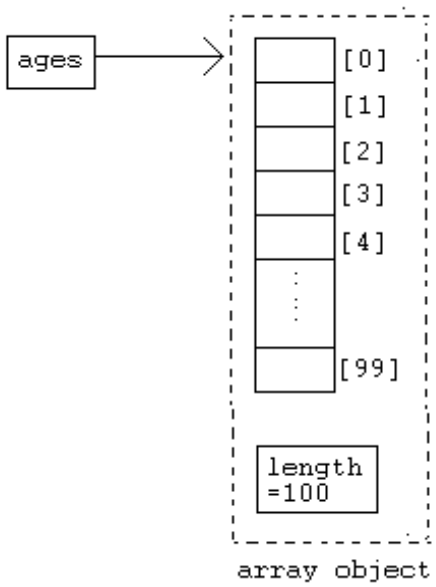


Figura 1: Exemplo de um array de inteiros

3. Declarando Array



Array precisa ser declarados como qualquer variável. Ao declarar um array, defina o tipo de dados deste seguido por colchetes [] e pelo nome que o identifica. Por exemplo:

```
int [] ages;
```

ou colocando os colchetes depois do identificador. Por exemplo:

```
int ages[];
```

Depois da declaração, precisamos criar o array e especificar seu tamanho. Este processo é chamado de **construção** (a palavra, em orientação a objetos, para a criação de objetos). Para se construir um objeto, precisamos utilizar um **construtor**. Por exemplo:

```
// declaração
int ages[];
// construindo
ages = new int[100];
```

ou, pode ser escrito como:

```
// declarar e construir
int ages[] = new int[100];
```

No exemplo, a declaração diz ao compilador Java que o identificador ages será usado comoum nome de um array contendo inteiros, usadopara criar, ou construir, um novo array contendo 100 elementos.

Em vez de utilizar uma nova linha de instrução para construir um array, também é possível automaticamente declarar, construir e adicionar um valor uma única vez.

Exemplos:

```
// criando um array de valores lógicos em uma variável
// results. Este array contém 4 elementos que são
// inicializados com os valores {true, false, true, false}
boolean results[] ={ true, false, true, false };

// criando um array de 4 variáveis double inicializados
// com os valores {100, 90, 80, 75};
double []grades = {100, 90, 80, 75};

// criando um array de Strings com identificador days e
// também já inicializado. Este array contém 7 elementos
String days[] = {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};
```

Uma vez que tenha sido inicializado, o tamanho de um array não pode ser modificado, pois é armazenado em um bloco contínuo de memória.

4. Acessando um elemento do Array

Para acessar um elemento do array, ou parte de um array, utiliza-se um número inteirochamado de **índice**.Um **índice** é atribuído para cada membro de um array, permitindo ao programa e ao programador acessar os valores individualmente quando necessário. Os números dos índices são sempre **inteiros**. Eles começam com zero e progridem seqüencialmente por todas as posições até o fim do array. Lembre-se que os elementos dentro do array possuem índice de **0** a **tamanhoDoArray- 1**.

Por exemplo, dado o array **ages** que declaramos anteriormente, temos:

```
// atribuir 10 ao primeiro elemento do array
ages[0] = 10;

// imprimir o último elemento do array
System.out.print(ages[99]);
```

Lembre-se que o array, uma vez declarado e construído, terá o valor de cada membro

inicializado automaticamente. Conforme a seguinte tabela:

Tipo primitivo	**Iniciado com**
boolean	false
byte, short e int	0
char	'\u0000'
long	0L
float	0.0F
double	0.0

Tabela 1: Valor de inicialização automatica para os tipos primitivos

Entretanto, tipos de dados por referência, como as Strings, não serão inicializados caracteres em branco ou com uma string vazia "", serão inicializados com o valor **null**. Deste modo, o ideal é preencher os elementos do arrays de forma explícita antes de utilizá-los. A manipulação de objetos nulos pode causar a desagradável surpresa de uma exceção do tipo **NullPointerException**, por exemplo, ao tentar executar algum método da classe String, conforme o exemplo a seguir:

```
public class ArraySample {
      public static void main(String[] args){
            String [] nulls = new String[2];
            System.out.print(nulls[0]); // Linha correta, mostra null
            System.out.print(nulls[1].trim()); // Causa erro
      }
}
```

O código abaixo utiliza uma declaração **for** para mostrar todos os elementos de um array.

```
public class ArraySample {
      public static void main(String[] args){
            int[] ages = new int[100];
            for (int i = 0; i < 100; i++) {
                  System.out.print(ages[i]);
            }
      }
}
```

Dicas de programação:

1. Normalmente, é melhor inicializar, ou instanciar, um array logo após declará-lo.
Por exemplo, a instrução:

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

é preferível, ao invés de:

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

2. Os elementos de um array de n elementos tem índices de 0 a n-1. Note que não existe o elemento arr[n]. A tentativa de acesso a este elemento causará uma exceção do tipo **ArrayIndexOutOfBoundsException**, pois o índice deve ser até n-1.
3. Não é possível modificar o tamanho de um array.

5. Tamanho do Array

Para se obter o número de elementos de um array, pode-se utilizar o atributo **length**. O atributo **length** de um array retorna seu tamanho, ou seja, a quantidade de elementos. É utilizado como no código abaixo:

```
nomeArray.length
```

Por exemplo, dado o código anterior, podemos reescrevê-lo como:

```
public class ArraySample {
      public static void main (String[] args) {
            int[] ages = new int[100];
            for (int i = 0; i < ages.length; i++) {
            System.out.print(ages[i]);
            }
      }
}
```

Dicas de programação:

1. Quando criar laços com **for** para o processamento de um array, utilize o campo **length** como argumento da expressão lógica. Isto irá permitir ao laço ajustarse, automaticamente para tamanhos de diferentes arrays.
2. Declare o tamanho dos arrays utilizando variáveis do tipo constante para facilitar alterações posteriores. Por exemplo:

```
final int ARRAY_SIZE = 1000; // declarando uma constante
...
int[] ages = new int[ARRAY_SIZE];
```

6. Arrays Multidimensionais

Arrays multidimensionais são implementados como arrays dentro de arrays. São declarados ao atribuir um novo conjunto de colchetes depois do nome do array. Por exemplo:

```
// array inteiro de 512 x 128 elementos
int [][] twoD = new int[512][128];
// array de caracteres de 8 x 16 x 24
char [][][] threeD = new char[8][16][24];
// array de String de 4 linhas x 2 colunas
String [][] dogs = {{"terry", "brown"},
{"Kristin", "white"},
{"toby", "gray"},
{"fido", "black"}};
```

Acessar um elemento em um array multidimensional é semelhante a acessar elementos em

um array de uma dimensão. Por exemplo, para acessar o primeiro elemento da primeira linha

do array **dogs**, escreve-se:

```
System.out.print(dogs[0][0]);
```

Isso mostrará a String "terry" na saída padrão. Caso queira mostrar todos os elementos deste array, escreve-se:

```
for (int i = 0; i < dogs.length; i++) {
      for (int j = 0; j < dogs[i].length; j++) {
            System.out.print(dogs[i][j] + " ");
      }
}
```

7. Exercícios

7.1. Dias da semana

Criar um array de Strings inicializado com os nomes dos sete dias da semana. Por exemplo: `String days[] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};`

Usando uma declaração **while**, imprima todo o conteúdo do array. Faça o mesmo para as declarações **do- while** e **for**.

7.2. Maior número

Usando as classes **BufferedReader**, **Scanner** ou **JOptionPane**, solicite 10 números ao usuário. Utilize um array para armazenar o valor destes números. Mostre o número de maior valor.

7.3. Entradas de agenda telefônica

Dado o seguinte array multidimensional, que contém as entradas da agenda telefônica:

```
String entry = {{"Florence", "735-1234", "Manila"},
{"Joyce", "983-3333", "Quezon City"},
{"Becca", "456-3322", "Manila"}};
```

mostre-as conforme o formato abaixo:

```
Name : Florence
Tel. # : 735-1234
Address: Manila
Name : Joyce
Tel. # : 983-3333
Address: Quezon City
Name : Becca
Tel. # : 456-3322
Address: Manila
```

Lição 6

1. Objetivos

Nesta lição, aprenderemos sobre como processar a entrada que vem da linha de comando usando argumentos passados para um programa feito em Java.

Ao final desta lição, o estudante será capaz de:

- Utilizar o argumento de linha de comando
- Receber dados enviados pelo usuário utilizando os argumentos de linha de comando
- Aprender como passar argumentos para os programas no NetBeans

2. Argumentos de linha de comando

Uma aplicação em Java aceita qualquer quantidade de argumentos passados pela linha decomando. Argumentos de linha de comando permitem ao usuário modificar a operação de uma aplicação a partir de sua execução. O usuário insere os argumentos na linha de comando no momento da execução da aplicação. Deve-se lembrar que os argumentos de linha de comando são especificados depois do nome da classe a ser executada.

Por exemplo, suponha a existência de uma aplicação Java, chamada **Sort**, que ordena cinco números que serão recebidos. Essa aplicação seria executada da seguinte maneira:

```
java Sort 5 4 3 2 1
```

Lembre-se que os argumentos são separados por espaços. Em linguagem Java, quando uma aplicação é executada, o sistema repassa os argumentos da linha de comando para a o método **main** da aplicação através de um array de String. Cada elemento deste array conterá um dos argumentos de linha de comando passados. Lembre-se da declaração do método main:

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

Os argumento que são passados para o programa são salvos em um array de String com o identificador args. No exemplo anterior, os argumentos de linha de comando passados para a aplicação Sort estarão em um array que conterá cinco strings: "5", "4", "3", "2" e "1". É possível conhecer o número de argumentos passados pela linha de comando utilizando-se o atributo **length** do array.

Por exemplo:

```
int numberOfArgs = args.length;
```

Se o programa precisa manipular argumento de linha de comando numérico, então, deve-se converter o argumento do tipo String, que representa um número, assim como "34", para um número. Aqui está a parte do código que converte um argumento de linha de comando para

inteiro:

```
int firstArg = 0;
```

```
if (args.length > 0) {
      firstArg = Integer.parseInt(args[0]);
}
```

parseInt dispara uma exceção do tipo **NumberFormatException** se o conteúdo do elemento arg[0] não for um número.

Dicas de programação:

1. Antes de usar os argumentos de linha de comando, observe a quantidade de argumentos passados para a aplicação. Deste modo, nenhuma exceção será disparada.

3. Argumentos de linha de comando no NetBeans

Para ilustrar a passagem de alguns argumentos para um projeto no NetBeans, vamos criar um projeto em Java que mostrará na tela o número de argumentos e o primeiro argumento passado.

```
public class CommandLineExample {
      public static void main( String[] args ) {
            System.out.println("Number of arguments=" +
            args.length);
            System.out.println("First Argument="+ args[0]);
      }
}
```

Abra o NetBeans, crie um novo projeto e dê o nome de **CommandLineExample**. Copie o código mostrado anteriormente e o compile. Em seguida, siga estas etapas para passar argumentos para o programa, utilizando o NetBeans.

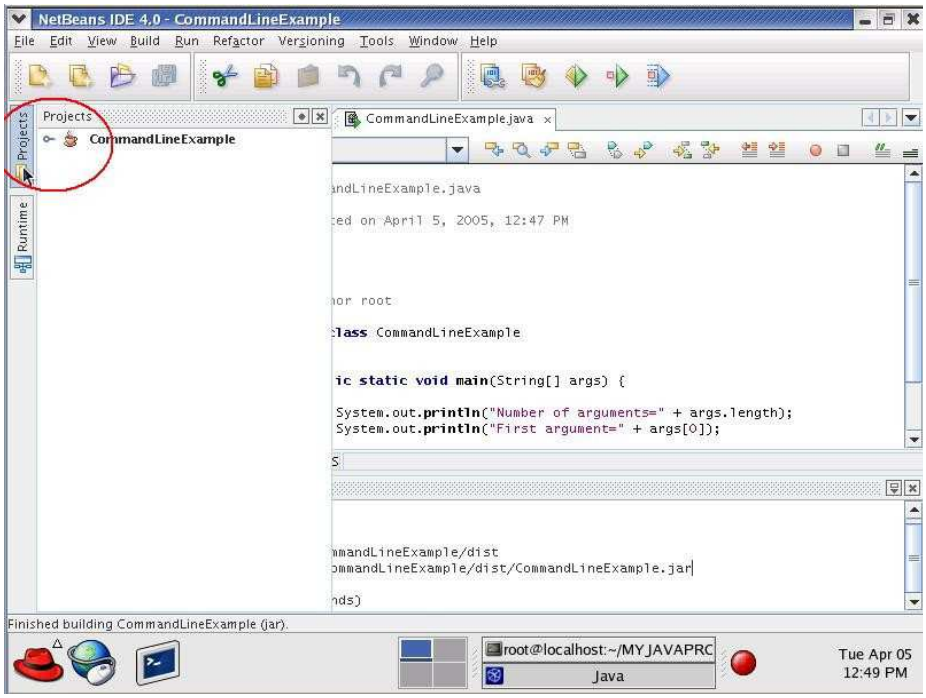


Figura 1: Abrindo o projeto

Dê um clique com o botão direito do mouse no ícone **CommandLineExample**, conformedestacado na **Figura 1**. Um menu aparecerá, conforme a **Figura 2**. Selecione a opção "Properties".

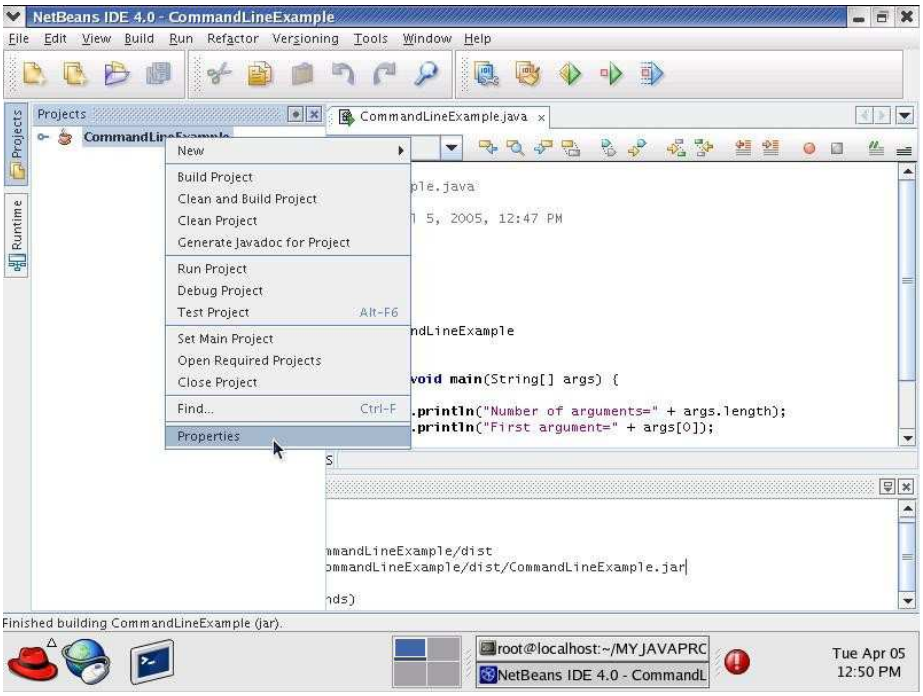


Figura 2: Abrindo a janela de propriedades

A janela "Project Properties" irá aparecer, conforme a **Figura 3**.

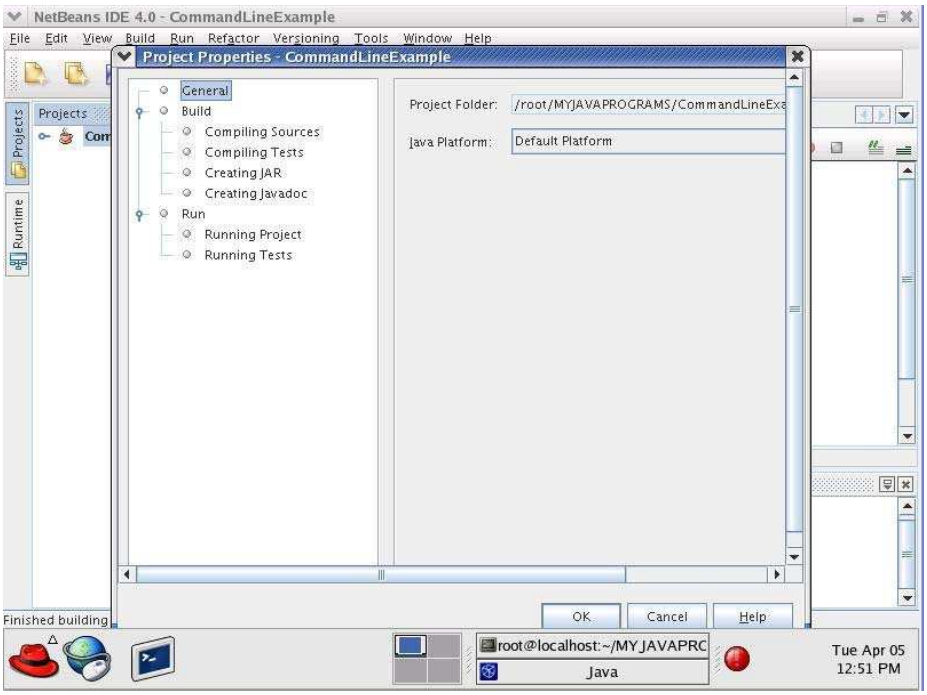


Figura 3: Janela de propriedades

Acesse a opção Run ⇒ Running Project.

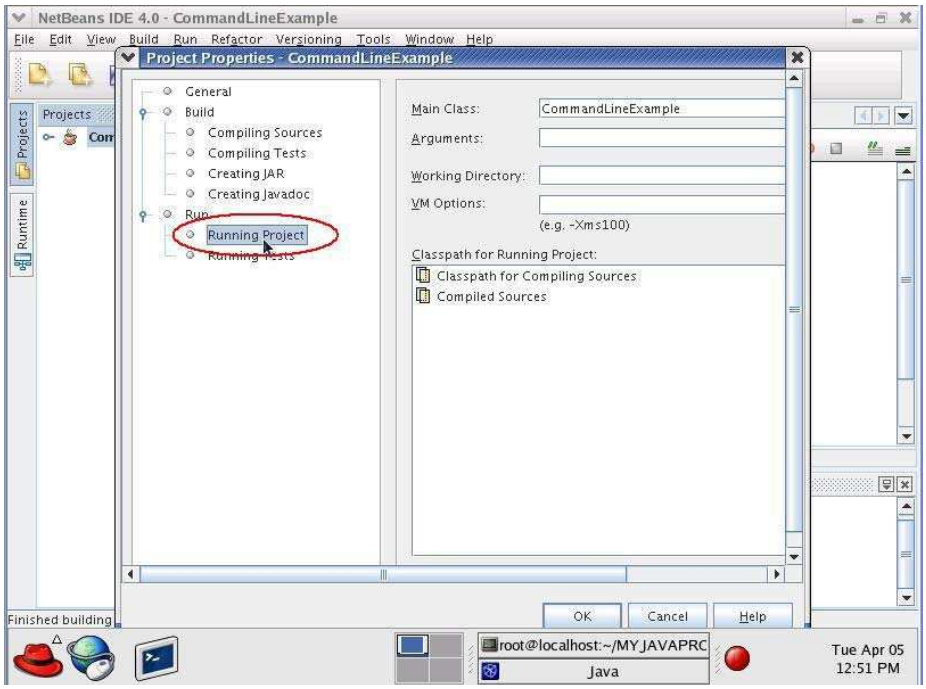


*Figura 4: Acessando através de **Running Project***

Na caixa de texto dos argumentos, digite os argumentos que se quer passar para o programa.

Neste caso, digitamos os argumentos 5 4 3 2 1. Pressione o botão **OK**.

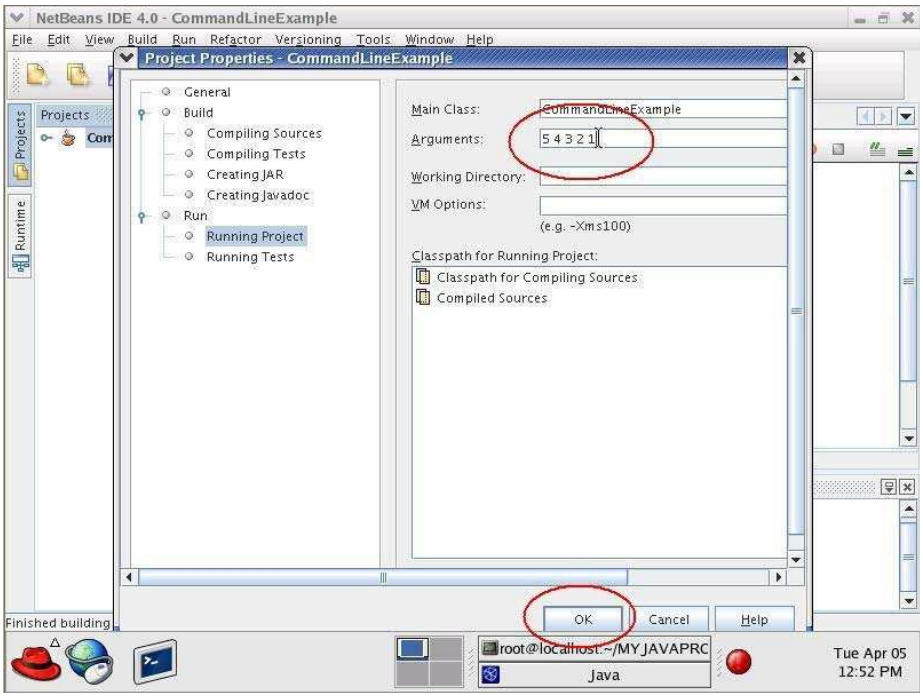


Figura 5: Salvando os Argumentos de Linha de Comando

Execute o projeto.

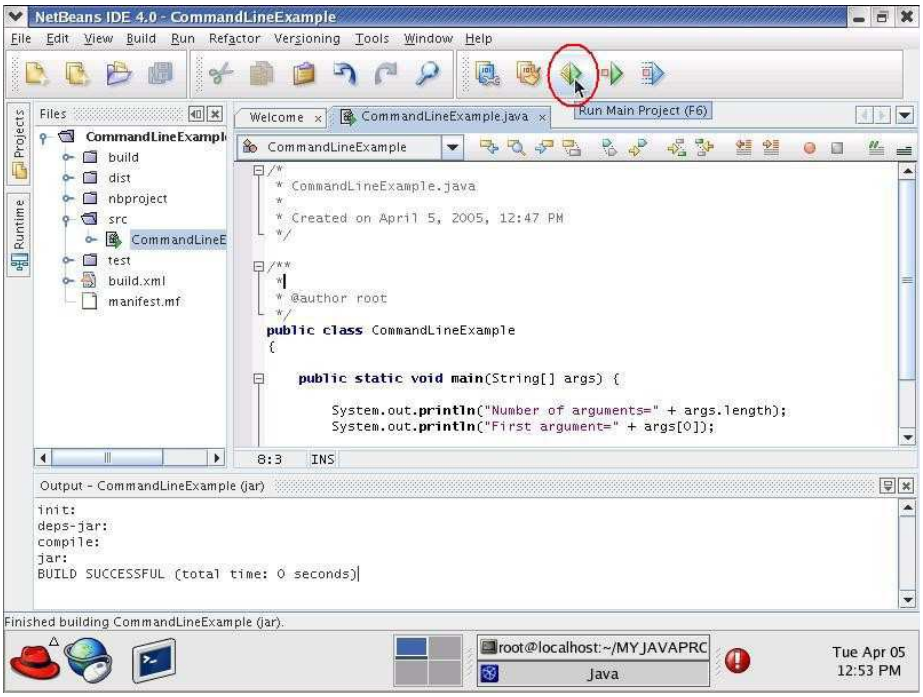


Figura 6: Executando o programa com botão de atalho

Como pode-se ver, a saída do projeto é a quantidade de argumentos, que é 5, e o primeiro argumento passado, que também é 5.

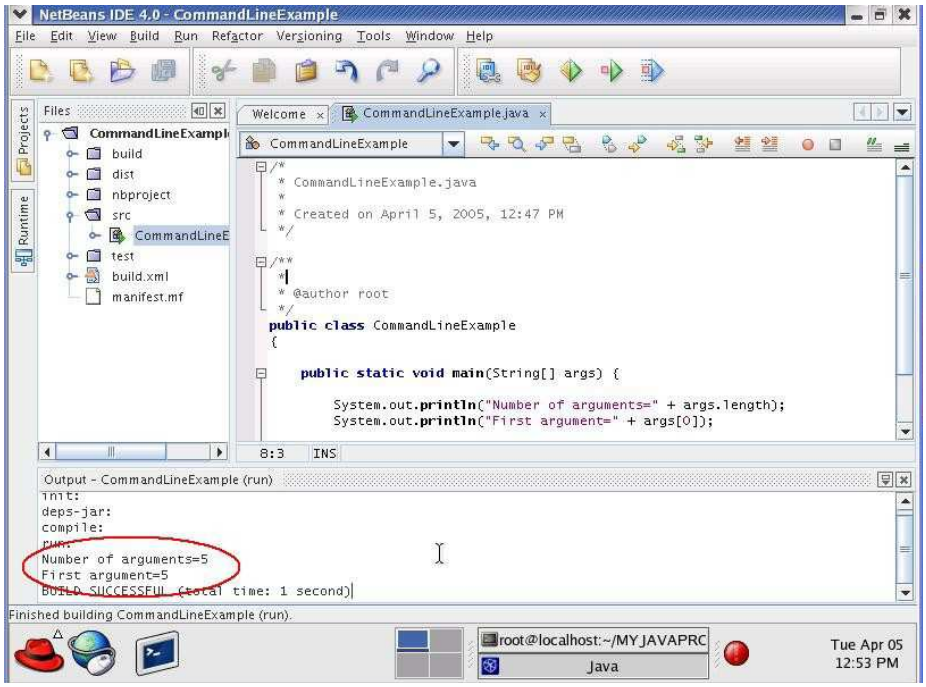


Figura 7: Saída do Programa

4. Exercícios

4.1. Argumentos de Exibição

Utilizando os dados passados pelo usuário através dos argumentos de linha de comando, exiba os argumentos recebidos. Por exemplo, se o usuário digitar:

```
java Hello world that is all
```

o programa deverá mostrar na tela:

```
world
that
is
all
```

4.2. Operações aritméticas

Obtenha dois números, passados pelo usuário usando argumentos de linha de comando, e mostre o resultado da soma, subtração, multiplicação e divisão destes números. Por exemplo,

se o usuário digitar:

```
java ArithmeticOperation 20 4
```

o programa deverá mostrar na tela:

```
sum = 24
subtraction = 16
multiplication = 80
division = 5
```

Lição 7

1. Objetivos

Nesta lição, abordaremos alguns conceitos básicos da Programação Orientada a Objetos ou POO. Mais adiante, discutiremos o conceito de classes e objetos e como usar as classes e seus membros. Comparação, conversão e casting de objetos, também serão vistos. Por enquanto, o foco será o uso das classes já definidas nas bibliotecas Java e, posteriormente, discutiremos como criar nossas próprias classes.

Ao final desta lição, o estudante será capaz de:

- Explicar o que é Programação Orientada a Objetos e alguns dos seus conceitos
- Diferenciar entre classes e objetos
- Diferenciar atributos e métodos de objeto de atributos e métodos de classe
- Explicar o que são métodos, como invocá-los e como enviar argumentos
- Identificar o escopo de um atributo
- Realizar conversões entre tipos de dados primitivos e entre objetos
- Comparar objetos e determinar suas classes

2. Introdução à Programação Orientada a Objeto

Programação Orientada a Objetos (POO) refere-se ao conceito de **objetos** como elemento básico das classes. O mundo físico é constituído por objetos tais como carro, leão, pessoa, dentre outros. Estes objetos são caracterizados pelas suas **propriedades** (ou **atributos**) e seus **comportamentos**.

Por exemplo, um objeto "carro" tem as propriedades, tipo de câmbio, fabricante e cor. O seu comportamento pode ser 'virar', 'frear' e 'acelerar'. Igualmente, podemos definir diferentes propriedades e comportamentos para um leão. Veja exemplos na **Tabela 1**.

Objetos	Propriwedades	Comportamentos
carro	tipo de câmbio fabricante cor	virar frear acelerar
Leão	peso cor apetite (faminto ou saciado) temperamento (dócil ou selvagem)	rugir dormir caçar

Tabela 1: Exemplos de objetos do mundo real

Com tais descrições, os objetos do mundo físico podem ser facilmente modelados como objetos de software usando as **propriedades como atributos** e os **comportamentos como métodos**. Estes atributos e métodos podem ser usados em softwares de jogos ou interativos para simular objetos do mundo real! Por exemplo, poderia ser um objeto de 'carro' numa competição de corrida ou um objeto de 'leão' num aplicativo educacional de zoologia para crianças.

3. Classes e Objetos

3.1. Diferenças entre Classes e Objetos

No mundo do computador, um **objeto** é um componente de software cuja estrutura é similar a um objeto no mundo real. Cada objeto é composto por um conjunto de **atributos** (propriedades) que são as variáveis que descrevem as características essenciais do objeto e, consiste também, num conjunto de **métodos** (comportamentos) que descrevem como o objeto se comporta. Assim, um objeto é uma coleção de atributos e métodos relacionados. Os atributos e métodos de um objeto Java são formalmente conhecidos como **atributos e métodos de objeto**, para distinguir dos atributos e métodos de classes, que serão discutidos mais adiante.

A **classe** é a estrutura fundamental na Programação Orientada a Objetos. Ela pode ser pensada como um gabarito, um protótipo ou, ainda, uma **planta** para a construção de um objeto. Ela consiste em dois tipos de elementos que são chamados **atributos** (ou propriedades) e **métodos**. Atributos especificam os tipos de dados definidos pela classe, enquanto que os métodos especificam as operações. Um objeto é uma **instância** de uma classe.

Para diferenciar entre classes e objetos, vamos examinar um exemplo. O que temos aqui é uma classe Carro que pode ser usada pra definir diversos objetos do tipo carro. Na tabela mostrada abaixo, Carro A e Carro B são objetos da classe Carro. A classe tem os *campos* número da placa, cor, fabricante e velocidade que são preenchidos com os valores correspondentes do carro A e B. O carro também tem alguns métodos: acelerar, virar e frear.

Classe Carro		**Objeto Carro A**	**Objeto Carro B**
Atributos de Objeto	Número da placa	ABC 111	XYZ 123
	Cor	Azul	Vermelha
	Fabricante	Mitsubishi	Toyota
	Velocidade	50 km/h	100 km/h
Métodos de Objeto	Método Acelerar		
	Método Girar		
	Método Frear		

Tabela 2: Exemplos da classe Carro e seus objetos

Quando construídos, cada objeto adquire um conjunto novo de estado. Entretanto, as implementações dos métodos são compartilhadas entre todos os objetos da mesma classe. As classes fornecem o benefício do **Reutilização de Classes** (ou seja, utilizar a mesma classe em vários projetos). Os programadores de software podem reutilizar as classes várias vezes para criar os objetos.

3.2. Encapsulamento

Encapsulamento é um princípio que propõe ocultar determinados elementos de uma classe das demais classes. Ao colocar uma proteção ao redor dos atributos e criar métodos para prover o acesso a estes, desta forma estaremos prevenindo contra os efeitos colaterais indesejados que podem afetá-los ao ter essas propriedades modificadas de forma inesperada.

Podemos prevenir o acesso aos dados dos nossos objetos declarando que temos controle desse acesso. Aprenderemos mais sobre como Java implementa o encapsulamento quando discutirmos mais detalhadamente sobre as classes.

3.3. Atributos e Métodos de Classe

Além dos atributos de objeto, também é possível definir **atributos de classe,** que são atributos que pertencem à classe como um todo. Isso significa que possuem o mesmo valor para todos os objetos daquela classe. Também são chamados de **atributos estáticos.**

Para melhor descrever os atributos de classe, vamos voltar ao exemplo da classe **Carro**. Suponha que a classe **Carro** tenha um atributo de classe chamado **Contador**. Ao mudarmos o valor de **Contador** para 2, todos os objetos da classe **Carro** terão o valor 2 para seus atributos Contador.

Classe Carro		**Objeto Carro A**	**Objeto Carro B**
Atributos de Objeto	Número da placa	ABC 111	XYZ 123
	Cor	Azul	Vermelho
	Fabricante	Mitsubishi	Toyota
	Velocidade	50 km/h	100 km/h
Atributos de Classe	Contador = 2		
Métodos de Objeto	Método Acelerar		
	Método virar		
	Método Frear		

Tabela 3: Atributos e métodos da classe Carro

3.4 Instância de Classe

Para criar um objeto ou uma instância da classe, utilizamos o operador **new**. Por exemplo, para criar uma instância da classe String, escrevemos o seguinte código:

```
String str2 = new String("Hello world!");
```

ou, o equivalente:

```
String str2 = "Hello world!";
```

O operador **new** aloca a memória para o objeto e retorna uma **referência** para essa alocação. Ao criar um objeto, invoca-se, na realidade, o **construtor** da classe. O **construtor** é um método onde todas as inicializações do objeto são declaradas e possui o mesmo nome da classe.

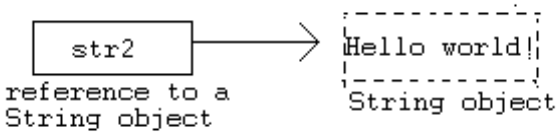


Figura 1: Instanciação de uma classe

4. Métodos

4.1. O que são métodos e porque usar métodos?

Nos exemplos apresentados anteriormente, temos apenas um método, o método main(). Em Java, nós podemos definir vários métodos e podemos chamá-los a partir de outros métodos. Um **método** é um trecho de código distinto que pode ser chamado por qualquer outro método para realizar alguma função específica.

Métodos possuem as seguintes características:

- Podem ou não retornar um valor;
- Podem aceitar ou não argumentos;
- Após o método encerrar sua execução, o fluxo de controle é retornado a quem o chamou.

O que é necessário para se criar métodos? Porque não colocamos todas as instruções dentro de um grande método? O foco destas questões é chamado de **decomposição**. Conhecido o problema, nós o separamos em partes menores, que torna menos crítico o trabalho de escrever grandes classes.

4.2. Chamando Métodos de Objeto e Enviando Argumentos

Para ilustrar como chamar os métodos, utilizaremos como exemplo a classe **String**. Pode-se usar a documentação da API Java para conhecer todos os atributos e métodos disponíveis na classe **String**. Posteriormente, iremos criar nossos próprios métodos.

Para chamar um **método** a partir de um objeto, escrevemos o seguinte:

```
nomeDoObjeto.nomeDoMétodo([argumentos]);
```

Vamos pegar dois métodos encontrados na classe String como exemplo:

Declaração do método Definição

public char charAt(int index)

Retorna o caractere especificado no índice.

Um índice vai de 0 até length() - 1. O primeiro caractere da seqüência está no índice 0, o seguinte, no índice 1, e assim sucessivamente por todo o array.

public boolean equalsIgnoreCase (String anotherString)

Compara o conteúdo de duas Strings, ignorando maiúsculas e minúsculas. Duas strings são consideradas iguais quando elas têm o mesmo tamanho e os caracteres das duas strings são iguais, sem considerar caixa alta ou baixa.

Declaração do método	Definição
`public char charAt(int index)`	Retorna o caractere especificado no índice. Um índice vai de 0 até length() - 1. O primeiro caractere da seqüência está no índice 0, o seguinte, no índice 1, e assim sucessivamente por todo o array.
`public boolean equalsIgnoreCase (String anotherString)`	Compara o conteúdo de duas Strings, ignorando maiúsculas e minúsculas. Duas strings são consideradas iguais quando elas têm o mesmo tamanho e os caracteres das duas strings são iguais, sem considerar caixa alta ou baixa.

Tabela 4: Exemplos de Métodos da classe String

Usando os métodos:

```
String str1 = "Hello";
char x = str1.charAt(0); // retornará o caracter H

// e o armazenará no atributo x
String str2 = "hello";

// aqui será retornado o valor booleano true
boolean result = str1.equalsIgnoreCase(str2);
```

4.3. Envio de Argumentos para Métodos

Em exemplos anteriores, enviamos atributos para os métodos. Entretanto, não fizemos nenhuma distinção entre os diferentes tipos de atributos que podem ser enviados como argumento para os métodos. Há duas formas para se enviar argumentos para um método, o primeiro é envio por valor e o segundo é envio por referência.

4.3.1. Envio por valor

Quando ocorre um **envio por valor**, a chamada do método faz uma cópia do valor do atributo e o reenvia como argumento. O método chamado não modifica o valor original do argumento mesmo que estes valores sejam modificados durante operações de cálculo implementadas pelo método. Por exemplo:

```
public class TestPassByValue {
      public static void main( String[] args ){
            int i = 10;
            //exibe o valor de i
            System.out.println( i );
            //chama o método test
            //envia i para o método test
            test( i );
            //exibe o valor de i não modificado
            System.out.println( i );
      }
      public static void test( int j ){
            //muda o valor do argumento j
            j = 33;
      }
}
```

No exemplo dado, o método **test** foi chamado e o valor de **i** foi enviado como argumento. O valor de **i** é copiado para o atributo do método **j**. Já que **j** é o atributo modificado no método **test**, não afetará o valor do atributo **i**, o que significa uma cópia diferente do atributo.

Como padrão, todo tipo primitivo, quando enviado para um método, utiliza a forma de **envio por valor**.

4.3.2. Envio por referência

Quando ocorre um **envio por referência**, a referência de um objeto é enviada para o método chamado. Isto significa que o método faz uma cópia da referência do objeto enviado.

Entretanto, diferentemente do que ocorre no envio por valor, o método pode modificar o objeto para o qual a referência está apontando. Mesmo que diferentes referências sejam usadas nos métodos, a localização do dado para o qual ela aponta é a mesma. Por exemplo:

```
class TestPassByReference {
      public static void main(String[] args) {
            // criar um array de inteiros
            int []ages = {10, 11, 12};
            // exibir os valores do array
            for (int i=0; i < ages.length; i++) {
                  System.out.println( ages[i] );
            }
            // chamar o método test e enviar a
            // referência para o array
```

```
            test( ages );
            // exibir os valores do array
            for (int i=0; i < ages.length; i++) {
                  System.out.println(ages[i]);
            }
      }
      public static void test( int[] arr ){
            // mudar os valores do array
            for (int i=0; i < arr.length; i++) {
                  arr[i] = i + 50;
            }
      }
}
```

Dicas de programação:

1. Um erro comum sobre **envio por referência** acontece quando criamos ummétodo para fazer trocas (**swap**) usando referência. Note que Java manipula objetos 'por referência', entretanto enviase a referência para um método 'por valor'. Como conseqüência, não se escreve um método padrão para fazer troca de valores (**swap**) entre objetos.

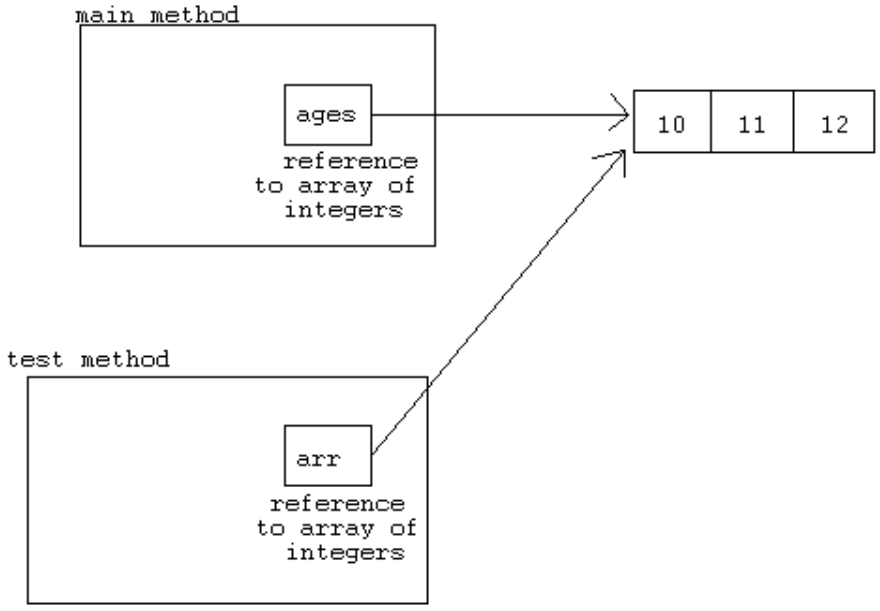


Figura 2: Exemplo de Envio por referência

4.4. Chamando métodos estáticos

Métodos estáticos são métodos que podem ser invocados sem que um objeto tenha sido instanciado pela classe (sem utilizar a palavra-chave *new*). Métodos estáticos pertencem a classe como um todo e não ao objeto especifico da classe. Métodos estáticos são diferenciados dos métodos de objeto pela declaração da palavra-chave *static* na definição do método.

Para chamar um método estático, digite:

```
NomeClasse.nomeMétodoEstático(argumentos);
```

Alguns métodos estáticos, que já foram usados em nossos exemplos são:

```
// Converter a String 10 em um atributo do tipo inteiro
int i = Integer.parseInt("10");

// Retornar uma String representando um inteiro sem o sinal da
// base 16
String hexEquivalent = Integer.toHexString(10);
```

4.5. Escopo de um atributo

Além do atributo ter um nome e um tipo, ele também possui um escopo. O **escopo** determina onde o atributo é acessível dentro da classe. O escopo também determina o tempo de vida do atributo ou quanto tempo o atributo irá existir na memória. O escopo é determinado pelo local onde o atributo é declarado na classe.

Para simplificar, vamos pensar no escopo como sendo algo existente entre as chaves { ...} . A chave à direita é chamada de chave de saída do bloco (**outer**) e a chave à esquerda é chamada chave de entrada do bloco (**inner**).

Ao declarar atributos fora de um bloco, eles serão visíveis (usáveis) inclusive pelas linhas da classe dentro do bloco. Entretanto, ao declarar os atributo dentro do bloco, não será possível utilizá-los fora do bloco.

O escopo de um atributo é dito local quando é declarado dentro do bloco. Seu escopo inicia com a sua declaração e vai até a chave de saída do bloco.

Por exemplo, dado o seguinte fragmento de código:

```
public class ScopeExample {
      int i = 0;
      public static void main(String[] args) {
            int j = 0;
            {
                  int k = 0;
                  int m = 0;
                  int n = 0;
            }
      }
}
```

O código acima representa cinco escopos indicado pelas letras. Dados os atributos **i**, **j**, **k**, **m** e **n**, e os cinco escopos **A**, **B**, **C**, **D** e **E**, temos os seguintes escopos para cada atributo:

```
O escopo do atributo i é A.
O escopo do atributo j é B.
O escopo do atributo k é C.
O escopo do atributo m é D.
O escopo do atributo n é E.
```

Dado dois métodos: **main** e **test** teremos o seguinte exemplo:

```
class TestPassByReference {
      public static void main(String[] args) {
            // criar um array de inteiros
            int []ages = {10, 11, 12};
            //exibir os valores do array
            for (int i=0; i < ages.length; i++){
                  System.out.println( ages[i] );
            }
            // chamar o método test e enviar a referência para o array
            test(ages);
            //exibe novamente os valores do array
            for (int i = 0; i < ages.length; i++) {
                  System.out.println( ages[i] );
            }
      }
      public static void test(int[] arr) {
            // modificar os valores do array
            for (int i = 0; i < arr.length; i++) {
                  arr[i] = i + 50;
            }
      }
}
```

Para o método **main**, os escopos dos atributos são:

```
ages[] - escopo A
i em B - escopo B
i em C – escopo C
```

E, no método **test**, os escopos dos atributos são:

```
arr[] - escopo D
i em E - escopo E
```

Quando atributos são declarados, o identificador deve ser único no escopo. Isto significa que se você tiver a seguinte declaração:

```
{
int test = 10;
int test = 20;
}
```

o compilador irá gerar um erro pois deve-se ter um nome único para o atributos dentro do bloco. Entretanto, é possível ter atributos definidos com o mesmo nome, se não estiverem declarados no mesmo bloco. Por exemplo:

```
public class TestBlock {
	int test = 10;

	public void test() {
		System.out.print(test);
		int test = 20;
		System.out.print(test);
	}
	public static void main(String[] args) {
		TestBlock testBlock = new TestBlock();
		testBlock.test();
	}
}
```

Quando a primeira instrução **System.out.print** for invocada, exibirá o valor **10** contido na primeira declaração do atributo **test**. Na segunda instrução **System.out.print**, o valor **20** é exibido, pois é o valor do atributos **test** neste escopo.

Dicas de programação:

1. Evite ter atributos declarados com o mesmo nome dentro de um método para não causar confusão.

5. *Casting*, Conversão e Comparação de Objetos

Nesta seção, vamos aprender como realizar um ***casting***. ***Casting***, ou ***typecasting***, é o processo de conversão de um certo tipo de dado para outro. Também aprenderemos como converter tipos de dados primitivos para objetos e vice-versa. E, finalmente, aprenderemos como comparar objetos.

5.1. Casting de Tipos Primitivos

Casting entre tipos primitivos permite converter o valor de um dado de um determinado tipo para outro tipo de dado primitivo. O *casting* entre primitivos é comum para os tipos numéricos. Há um tipo de dado primitivo que não aceita o *casting*, o tipo de dado **boolean**.

Como demonstração de *casting* de tipos, considere que seja necessário armazenar um valor do tipo **int** em um atributo do tipo **double**. Por exemplo:

```
int numInt = 10;
double numDouble = numInt; // cast implícito
```

uma vez que o atributo de destino é **double**, pode-se armazenar um valor cujo tamanho seja menor ou igual aquele que está sendo atribuído. O tipo é convertido implicitamente. Quando convertemos um atributo cujo tipo possui um tamanho maior para um de tamanho menor, necessariamente devemos fazer um ***casting* explícito**. Esse possui a seguinte forma:

```
(tipoDado)valor
```

onde:

tipoDado é o nome do tipo de dado para o qual se quer converter o valor
valor é um valor que se quer converter.

Por exemplo:

```
double valDouble = 10.12;
int valInt = (int)valDouble; //converte valDouble para o tipo int
double x = 10.2;
int y = 2;
int result = (int)(x/y); //converte o resultado da operação para int
```

Outro exemplo é quando desejamos fazer um *casting* de um valor do tipo **int** para **char**. Um caractere pode ser usado como **int** porque para cada caractere existe um correspondente numérico que representa sua posição no conjunto de caracteres. O *casting* **(char)65** irá produzir a saída 'A'. O código numérico associado à letra maiúscula A é 65, segundo o conjunto de caracteres ASCII. Por exemplo:

```
char valChar = 'A';
System.out.print((int)valChar); //casting explícito produzirá 65
```

5.2. Casting de Objetos

Para objetos, a operação de *casting* também pode ser utilizada para fazer a conversão para outras classes, com a seguinte restrição: a classe de origem e a classe de destino devem ser da mesma família, relacionadas por herança; uma classe deve ser subclasse da outra. Veremos mais em lições posteriores sobre herança.

Analogamente à conversão de valores primitivos para um tipo maior, alguns objetos não necessitam ser convertidos explicitamente. Em conseqüência de uma subclasse conter todas as informações da sua superclasse, pode-se usar um objeto da subclasse em qualquer lugar onde a superclasse é esperada.

Por exemplo, um método que recebe dois argumentos, um deles do tipo **Object** e outro do tipo **Window**. Pode-se enviar um objeto de qualquer classe como argumento **Object** porque todas as classes Java são subclasses de **Object**. Para o argumento **Window**, é possível enviar apenas suas subclasses, tais como **Dialog**, **FileDialog**, **Frame** (ou quaisquer de subclasses de suas subclasses, indefinidamente). Isso vale para qualquer parte da classe, não apenas dentro da chamadas do método. Para um objeto definido como uma classe Window, é possível atribuir objetos dessa classe ou qualquer uma de suas subclasses para esse objeto sem o *casting*.

O contrário também é verdadeiro. Uma superclasse pode usada quando uma subclasse é esperada. Entretanto, nesse caso, o *casting* é necessário porque as subclasses contém mais métodos que suas superclasses, e isso acarreta em perda de precisão. Os objetos das superclasses podem não dispor de todo o comportamento necessário para agir como um objeto da subclasse. Por exemplo, se uma operação faz a chamada a um método de um objeto da classe **Integer**, usando um objeto da classe **Number**, ele não terá muitos dos métodos que foram especificados na classe **Integer**. Erros ocorrerão se você tentar chamar métodos que não existem no objeto de destino.

Para usar objetos da superclasse onde uma subclasse é esperada, é necessário fazer o ***casting* explícito**. Nenhuma informação será perdida no *casting*, entretanto, ganhará todos os atributos e métodos que a subclasse define. Para fazer o *casting* de um objeto para outro, utiliza-se a mesma operação utilizada com os tipos primitivos:

Para fazer o *casting*:

‘ onde:
nomeClasse é o nome da classe destino
objeto é a referência para o objeto origem que se quer converter

Uma vez que *casting* cria uma referência para o antigo objeto de **nomeClasse**; ele continua a existir depois do *casting*.

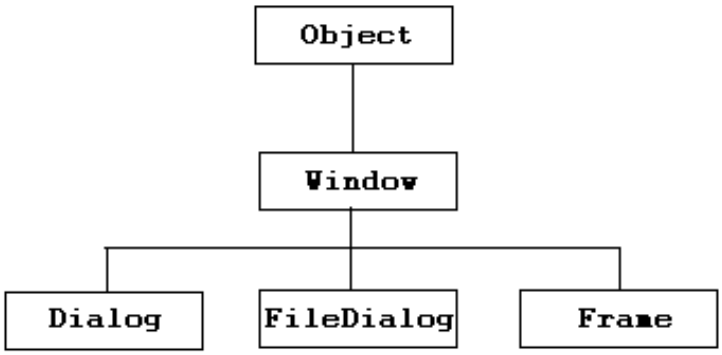


Figura 3: Exemplo de Hierarquia de Classes

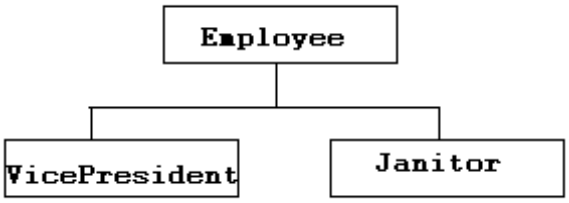


Figura 4: Hierarquia de classes para a superclasse Employee

O exemplo a seguir realiza um *casting* de um objeto da classe **VicePresident** para um objeto da classe **Employee**. **VicePresident** é uma subclasse de **Employee** com mais informações, supondo que tenha sido definido que a classe **VicePresident** tem mais privilégios que um **Employee**.

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // casting não é necessário de baixo para cima
veep = (VicePresident)emp; // necessita do casting explícito
```

5.3. Convertendo Tipos Primitivos para Objetos e Vice- Versa

Não se pode fazer sob qualquer circunstância é um *casting* de um objeto para um tipo de dado primitivo, ou vice-versa. Tipos primitivos e objetos são muito diferentes em Java e não se pode fazer o *casting* automaticamente entre os dois ou intercambiar o seu uso.

Como alternativa, o pacote **java.lang** oferece classes que fazem a correspondência para cada tipo primitivo: Float, Boolean, Byte, dentre outros. Muitas dessas classes têm o mesmo nome dos tipos de dados primitivos, exceto pelo fato que o nome dessas classe começam com letra maiúscula (Short ao invés de short, Double ao invés de double, etc). Há duas classes que possuem nomes que diferem do correspondente tipo de dado primitivo: **Character** é usado para o tipo **char** e **Integer** usado para o tipo **int**. Estas classes são chamadas de **Wrapper Class**.

Versões anteriores a 5.0 de Java tratam os tipos de dados e suas **Wrapper Class** de forma muito diferente e uma classe não será compilada com sucesso se for utilizado uma quando deveria usar a outra. Por exemplo:

```
Integer ten = 10;
Integer two = 2;
System.out.println(ten + two);
```

Usando as classes que correspondem a cada um dos tipos primitivos, é possível criar objetos que armazenam este mesmo valor. Por exemplo:

```
// A declaração seguinte cria um objeto de uma classe Integer
// com o valor do tipo int 7801 (primitivo -> objeto)
Integer dataCount = new Integer(7801);

// A declaração seguinte converte um objeto Integer para um
// tipo de dado primitivo int. O resultado é o valor 7801
int newCount = dataCount.intValue();

// Uma conversão comum de se fazer é de String para um tipo
// numérico (objeto -> primitivo)
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

Dicas de programação:

1. A classe Void representa vazio em Java. Deste modo, não existem motivos para ela ser usada na conversão de valores primitivos e objetos. Ela é um tratador para a palavra-chave *void*, que é utilizada na assinatura de métodos indicando que estes não retornam valor.

5.4. Comparando Objetos

Em lições prévias, aprendemos sobre operadores para comparar valores — igualdade, negação, menor que, etc. Muitos desses operadores trabalham apenas com dados de tipo primitivo, não com objetos. Ao tentar utilizar outros tipos de dados, o compilador produzirá erros.

Uma exceção para esta regra são os operadores de igualdade: == (igual) e != (diferente). Quando aplicados a objetos, estes operadores não fazem o que se poderia supor. Ao invés de verificar se um objeto tem o mesmo valor de outro objeto, eles determinam se os dois objetos comparados pelo operador têm a mesma referência. Por exemplo:

```
String valor1 = new String;
Integer dataCount = new Integer(7801);
```

Para comparar objetos de uma classe e ter resultados apropriados, deve-se implementar e chamar métodos especiais na sua classe. Um bom exemplo disso é a classe String.É possível ter dois objetos String diferentes que contenham o mesmo valor. Caso se empregue o operador **==** para comparar objetos, estes serão considerados diferentes. Mesmo que seus conteúdos sejam iguais, eles não são o mesmo objeto. Para ver se dois objetos String têm o mesmo conteúdo, um método chamado **equals()** é utilizado. O método compara cada caractere presente no conteúdo das Strings e retorna **true** se ambas strings tiverem o mesmo valor.

O código seguinte ilustra essa comparação,

```
class EqualsTest {
      public static void main(String[] args) {
            String str1, str2;
            str1 = "Free the bound periodicals.";
            str2 = str1;
            System.out.println("String1: " + str1);
            System.out.println("String2: " + str2);
            System.out.println("Same object? " + (str1 == str2));
            str2 = new String(str1);
            System.out.println("String1: " + str1);
            System.out.println("String2: " + str2);
            System.out.println("Same object? " + (str1 == str2));
            System.out.println("Same value? " + str1.equals(str2));
      }
}
```

A saída dessa classe é a seguinte:

```
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? true
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? false
Same value? true
```

Vamos entender o processo envolvido.

```
String str1, str2;
str1 = "Free the bound periodicals.";
str2 = str1;
```

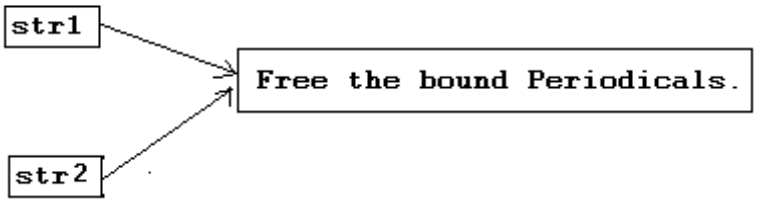


Figura 5: Duas referências apontando para o mesmo objeto

A primeira parte dessa classe declara dois atributos (str1 e str2) e atribui a literal "Free the bound periodicals." a str1, e depois atribui esse valor a str2. Como visto anteriormente, str1 e str2 agora apontam para o mesmo objeto, e o teste de igualdade prova isso.

Em seguida, temos a seguinte instrução:

```
str2 = new String(str1);
```

Na segunda parte da classe, cria-se um novo objeto String com o mesmo valor de str1 e faz-se a atribuição de str2 para esse novo objeto String. Agora temos dois diferentes tipos de objetos na str1 e str2, ambos com o mesmo conteúdo. Testando para ver se eles são o mesmo objeto usando o operador de igualdade obtemos a resposta esperada: **false** — eles não são o mesmo objeto na memória. Utilizando o método **equals()** recebemos a resposta esperada: **true** — eles tem o mesmo conteúdo.

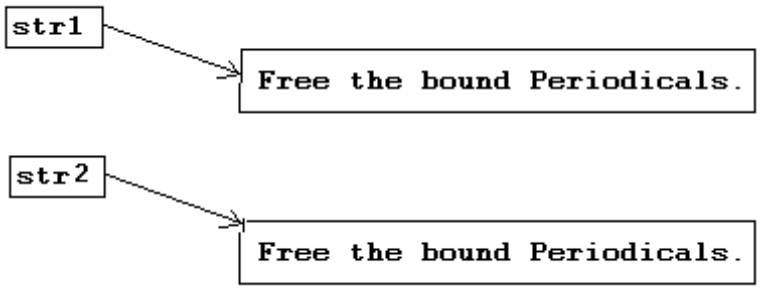


Figura 6: As referências apontam agora para objetos diferentes

Dicas de programação:

1. Porque não se pode ter a mesma literal quando mudamos str2, a não ser quando utilizamos o **new**? Literais String são otimizadas em Java; se uma String é criada utilizando uma literal e cria-se outra String com os mesmos caracteres, Java tem inteligência suficiente para retornar apenas a posição em memória do primeiro objeto String criado. Ou seja, ambas strings se referem ao mesmoobjeto como se fosse um apelido. Para obrigar ao Java criar um novo objeto String, deve-se utilizar o operador new.

5.5. Determinando a Classe de um Objeto

Existem duas maneiras de se descobrir a qual classe determinado objeto pertence:

1. Para obter o nome da classe:

Utiliza-se o método **getClass()** que retorna a classe do objeto (onde Class é a classe em si). Esta, por sua vez, possui o método chamado **getName()** que retorna o nome da classe.

Por exemplo:

```
String name = key.getClass().getName();
```

2. Para testar se um objeto qualquer foi instanciado de uma determinada classe:

Utiliza-se a palavra-chave *instanceof*. Esta palavra-chave possui dois operadores: a referência para o objeto à esquerda e o nome da classe à direita. A expressão retorna um lógico dependendo se o objeto é uma instância da classe declarada ou qualquer uma de suas subclasses.

Por exemplo:

```
String ex1 = "Texas";
System.out.println(ex1 instanceof String); // retorna true
String ex2;
System.out.println(ex2 instanceof String); // retorna false
```

6. Exercícios

6.1. Definindo termos

Em suas palavras, defina os seguintes termos:

1. Classe
2. Objeto
3. Instanciação
4. Atributo de objeto
5. Método de objeto
6. Atributo de classe ou atributos estáticas
7. Construtor
8. Método de classe ou métodos estáticos

6.2. Java Scavenger Hunt

Pipoy é um novato na linguagem de programação Java. Ele apenas ouviu que existem Java APIs (Application Programming Interface) prontas para serem usadas em suas classes e ele está ansioso para fazer uns testes com elas. O problema é que **Pipoy** não tem uma cópia da documentação Java e ele também não tem acesso à Internet, deste modo, não há como ele ver as APIs java.

Sua tarefa é ajudar **Pipoy** a procurar as APIs. Você deve informar as classes às quais os métodos pertencem e como o método deve ser declarado com um exemplo de uso deste.

Por exemplo, se **Pipoy** quer saber qual o método que converte uma String para int, sua resposta deve ser:

Classe: Integer

Declaração do Método: public static int parseInt(String value)

Exemplo de Uso:

```
String strValue = "100";
int value = Integer.parseInt( strValue );
```

Tenha certeza de que o fragmento de código que você escreveu em seu exemplo de uso compila e que produza o resultado correto. Enfim, não deixe **Pipoy** confuso. (**Dica: Todos os métodos estão no package java.lang**). Caso haja mais de um método para atender à tarefa, utilize apenas um.

Agora vamos iniciar a busca! Aqui estão alguns métodos que **Pipoy** necessita:

1. Procure pelo método que verifica se uma String termina com um determinado sufixo. Por exemplo, se a String dada é "Hello", o método deve retornar true se o sufixo informado é "lo", e false se o sufixo for "alp".

2. Procure pelo método que determina a representação do caractere para um dígito e base específicos. Por exemplo, se o dígito informado é 15 e a base é 16, o método retornará o caractere 'F', uma vez que 'F' é a representação hexadecimal para o número 15 em base 10.

3. Procure por um método que retorna a parte inteira de um valor double. Por exemplo, se a entrada for 3.13, o método deve retornar o valor 3.

4. Procure por um método que determina se um certo caractere é um dígito. Por exemplo, se a entrada for '3', retornará o valor true.

5. Procure por um método que interrompe a execução da Java Virtual Machine corrente.

Lição 8

1. Objetivos

Agora que já estudamos como usar as classes existentes na biblioteca de classes do Java, estudaremos como criar nossas próprias classes. Nesta lição, para facilmente entender como criá-las, realizaremos um exemplo de classe no qual adicionaremos dados e funcionalidades à medida em que avançarmos no curso.

Criaremos uma classe com informações de um Estudante e com operações necessárias para seu registro.

Algumas observações devem ser feitas quanto à sintaxe que será usada nesta e nas próximas seções:

* - significa que pode haver nenhuma ou diversas ocorrências na linha em que for aplicada < descrição> - indica a substituição deste trecho por um certo valor, ao invés de digitá-lo como está [] - indica que esta parte é opcional
Ao final desta lição, o estudante será capaz de:

- Criar nossas classes
- Declarar atributos e métodos para as classes
- Usar o objeto **this** para acessar dados de instância
- Utilizar **overloading** de métodos
- Importar e criar pacotes
- Usar modificadores de acesso para controlar o acesso aos elementos de uma classe

2. Definindo nossas classes

Antes de escrever sua classe, primeiro pense onde e como sua classe será usada. Pense em um nome apropriado para a classe e liste todas as informações ou propriedades que deseje que ela tenha. Liste também os métodos que serão usados para a classe.

Para definir uma classe, escrevemos:

```
<modificador>* class <nome> {
<declaraçãoDoAtributo>*
<declaraçãoDoConstrutor>*
<declaraçãoDoMétodo>*
}
```

onde:

```
<modificador> é um modificador de acesso, que pode ser
usado em combinação com outros
<nome> nome da sua classe
<declaraçãoDoAtributo> atributos definidos para a classe
<declaraçãoDoConstrutor> método construtor
```

```
<declaraçãoDoMétodo> métodos da classe
```

Dicas de programação:

1. Lembre-se de que, para a declaração da classe, o único modificador de acesso válido é o **public**. De uso exclusivo para a classe que possuir o mesmo nome do arquivo externo.

Nesta lição, criaremos uma classe que conterá o registro de um estudante. Como já identificamos o objetivo da nossa classe, agora podemos nomeá-la. Um nome apropriado para nossa classe seria StudentRecord.

Para definir nossa classe, escrevemos:

```
public class StudentRecord {
// adicionaremos mais código aqui
}
```

onde:

```
public modificador de acesso e significa que qualquer classe pode acessar esta
```

class palavra-chave usada para criar uma classe StudentRecord identificador único que identifica a classe

Dicas de programação:

1. Pense em nomes apropriados para a sua classe. Não a chame simplesmente de classe XYZ ou qualquer outro nome aleatório.
2. Os nomes de classes devem ser iniciadas por letra MAIÚSCULA.
3. O nome do arquivo de sua classe obrigatoriamente possui o MESMO NOME da sua classe pública.

3. Declarando Atributos

Para declarar um certo atributo para a nossa classe, escrevemos:

```
<modificador>* <tipo> <nome> [= <valorInicial>];
```

onde:

```
modificador tipo de modificador do atributo
tipo tipo do atributo
nome pode ser qualquer identificador válido
valorInicial valor inicial para o atributo
```

Relacionaremos a lista de atributos que um registro de estudante pode conter. Para cada informação, listaremos os tipos de dados apropriados que serão utilizados. Por exemplo, não seria ideal usar um tipo int para o nome do estudante ou String para a nota do estudante.

Abaixo, por exemplo, temos algumas informações que podemos adicionar ao registro do estudante:

```
nome - String
endereço - String
idade - int
nota de matemática - double
nota de inglês - double
nota de ciências - double
```

Futuramente, é possível adicionar mais informações. Para este exemplo, utilizaremos somente estas.

3.1. Atributos de Objeto

Agora que temos uma lista de todos os atributos que queremos adicionar à nossa classe, vamos adicioná-los ao nosso código. Uma vez que queremos que estes atributos sejam únicos para cada objeto (ou para cada estudante), devemos declará-los como atributos de objeto.

Por exemplo:

```
public class StudentRecord {
    private String name;
    private String address;
    private int age;
    private double mathGrade;
    private double englishGrade;
    private double scienceGrade;
}
```

onde:

private significa que os atributos são acessíveis apenas de dentro da classe. Outros objetos não podem acessar diretamente estes atributos.

Dicas de programação:

1. Declare todas os atributos de objeto na parte superior da declaração da classe.
2. Declare cada atributo em uma linha.

3. Atributos de objeto, assim como qualquer outro atributo devem iniciar com letra MINÚSCULA.
4. Use o tipo de dado apropriado para cada atributo declarado.
5. Declare atributos de objetos como **private** de modo que somente os métodos da classe possam acessá-los diretamente.

3.2. Atributos de Classe ou Atributos Estáticos

Além das atributos de objeto, podemos também declarar atributos de classe ou atributos que pertençam à classe como um todo. O valor destes atributos é o mesmo para todos os objetos da mesma classe. Suponha que queiramos saber o número total de registros criados para a classe. Podemos declarar um atributo estático que armazenará este valor. Vamos chamá-lo de studentCount.

Para declarar um atributo estático:

```
public class StudentRecord {
     // atributos de objeto declarados anteriormente
     private static int studentCount;
}
```

usamos a palavra-chave static para indicar que é um atributo estático.

Então, nosso código completo deve estar assim:

```
public class StudentRecord {
     private String name;
     private String address;
     private int age;
     private double mathGrade;
     private double englishGrade;
     private double scienceGrade;
     private static int studentCount;
}
```

4. Declarando Métodos

Antes de discutirmos quais métodos que a nossa classe deverá conter, vejamos a sintaxe geral usada para a declaração de métodos.

Para declararmos métodos, escrevemos:

```
<modificador>* <tipoRetorno> <nome>(<argumento>*) {
<instruções>*
}
```

onde:

```
<modificador> pode ser utilizado qualquer modificador de acesso
<tipoRetorno> pode ser qualquer tipo de dado (incluindo void)
<nome> pode ser qualquer identificador válido
<argumento> argumentos recebidos pelo método separados por vírgulas. São definidos por:
    <tipoArgumento> <nomeArgumento>
```

4.1. Métodos assessores

Para que se possa implementar o princípio do encapsulamento, isto é, não permitir que quaisquer objetos acessem os nossos dados de qualquer modo, declaramos campos, ou atributos, da nossa classe como particulares.Entretanto, há momentos em que queremos que outros objetos acessem estes dados particulares. Para que possamos fazer isso, criamos **métodos assessores**.

Métodos assessores são usados para ler valores de atributos de objeto ou de classe. O método assessor recebe o nome de **get< NomeDoAtributo>** . Ele retorna um valor. Para o nosso exemplo, queremos um método que possa ler o nome, endereço, nota de inglês, nota de matemática e nota de ciências do estudante. Vamos dar uma olhada na implementação deste:

```
public class StudentRecord {
      private String name;
      :
      :
      public String getName() {
            return name;
      }
}
```

onde:

public significa que o método pode ser chamado por objetos externos à classe String é o tipo do retorno do método. Isto significa que o método deve retornar um valor de tipo **String** getName o nome do método() significa que o nosso método não tem nenhum argumento

A instrução:

```
return name;
```

no método, significa que retornará o conteúdo do atributo **name** ao método que o chamou. Note que o tipo do retorno do método deve ser do mesmo tipo do atributo utilizado na declaração **return**. O

seguinte erro de compilação ocorrerá caso o método e o atributo de retorno não tenham o mesmo tipo de dados:

```
StudentRecord.java:14: incompatible types
found : int
required: java.lang.String
return name;
^
1 error
```

Outro exemplo de um método assessor é o método getAverage:

```
public class StudentRecord {
      private String name;
      :
      :
      public double getAverage(){
      double result = 0;
      result =
      (mathGrade+englishGrade+scienceGrade)/3;
      return result;
      }
}
```

O método **getAverage** calcula a média das 3 notas e retorna o resultado.

4.2 Métodos Modificadores

Para que outros objetos possam modificar os nossos dados, disponibilizamos métodos que possam gravar ou modificar os valores dos atributos de objeto ou de classe. Chamamos a estes métodos **modificadores**. Este método é escrito como **set< NomeDoAtributoDeObjeto>** .

Vamos dar uma olhada na implementação de um método modificador:

```
public class StudentRecord {
      private String name;
      :
      :
      public void setName(String temp) {
      name = temp;
      }
}
```

onde:

public significa que o método pode ser chamado por objetos externos à classevoid significa que o método não retorna valor setName o nome do método (String temp) argumento que será utilizado dentro do nosso método

A instrução:

```
name = temp;
```

atribuir o conteúdo de **temp** para **name** e, portanto, alterar os dados dentro do atributo de objeto **name**.

Métodos modificadores não retornam valores. Entretanto, eles devem receber um argumento com o mesmo tipo do atributo no qual estão tratando.

4.3. Múltiplos comandos return

É possível ter vários comandos **return** para um método desde que eles não pertençam ao mesmo bloco. É possível utilizar constantes para retornar valores, ao invés de atributos.

Por exemplo, considere o método:

```
public String getNumberInWords(int num) {
      String defaultNum = "zero";
      if (num == 1) {
      return "one"; // retorna uma constante
      } else if( num == 2) {
      return "two"; // retorna uma constante
      }
      // retorna um atributo
      return defaultNum;
}
```

4.4. Métodos estáticos

Para o atributo estático **studentCount**, podemos criar um método estático para obter o seu conteúdo.

```
public class StudentRecord {
private static int studentCount;
public static int getStudentCount(){
      return studentCount;
}
}
```

onde:

public significa que o método pode ser chamado por objetos externos à classe static significa que o método é estático e deve ser chamado digitando-se [NomeClasse].[nomeMétodo] int é o tipo do retorno do método. Significa que o método deve retornar um valor de tipo **int** getStudentCount nome do método() significa que o método não tem nenhum argumento Por enquanto, **getStudentCount** retornará sempre o valor zero já que ainda não fizemos nada na nossa classe para atribuir o seu valor. Modificaremos o valor de **studentCount** mais tarde, quando discutirmos construtores.

Dicas de programação:

1. Nomes de métodos devem iniciar com letra MINÚSCULA.

2. Nomes de métodos devem conter verbos

3. Sempre faça documentação antes da declaração do método. Use o

estilo **javadoc** para isso.

4.5. Exemplo de Código Fonte para a classe StudentRecord

Aqui está o código para a nossa classe **StudentRecord**:

```
public class StudentRecord {
      private String name;
      private String address;
      private int age;
      private double mathGrade;
      private double englishGrade;
      private double scienceGrade;
      private static int studentCount;
      /**
      * Retorna o nome do estudante
      */
      public String getName(){
      return name;
      }
      /**
      * Muda o nome do estudante
      */
      public void setName( String temp ){
            name = temp;
}
// outros métodos modificadores aqui ....
/**
* Calcula a média das classes de inglês, matemática
* e ciências
*/
public double getAverage(){
      double result = 0;
      result =
      (mathGrade+englishGrade+scienceGrade)/3;
      return result;
}
/**
* Retorna o número de ocorrências em StudentRecords
*/
public static int getStudentCount(){
      return studentCount;
}
}
```

Aqui está um exemplo do código de uma classe que utiliza a nossa classe **StudentRecord**.

```
public class StudentRecordExample {
```

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

        // criar três objetos para StudentRecord
        StudentRecord annaRecord = new StudentRecord();
        StudentRecord beahRecord = new StudentRecord();
        StudentRecord crisRecord = new StudentRecord();
        /
        / enviar o nome dos estudantes
        annaRecord.setName("Anna");
        beahRecord.setName("Beah");
        crisRecord.setName("Cris");

        // mostrar o nome de anna
        System.out.println(annaRecord.getName());

        //mostrar o número de estudantes
        System.out.println("Count=" +
        StudentRecord.getStudentCount());
        }
}
```

A saída desta classe é:

```
Anna
Count = 0
```

5. this

O objeto ***this*** é usado para acessar atributos de objeto ou métodos da classe. Para entender isso melhor, tomemos o método **setAge** como exemplo. Suponha que tenhamos o seguinte método para **setAge**:

```
public void setAge(int age){
        age = age; // Não é uma boa prática
}
```

O nome do argumento nesta declaração é **age**, que tem o mesmo nome do atributo de objeto **age**. Já que o argumento **age** é a declaração mais próxima do método, o valor do argumento **age** será usado. Na instrução:

```
age = age;
```

estamos simplesmente associando o valor do argumento **age** para si mesmo! Isto não é o que queremos que aconteça no nosso código. A fim de corrigir esse erro, usamos o objeto **this**. Para utilizar o objeto **this**, digitamos:

```
this.<nomeDoAtributo>
```

O ideal é reescrever o nosso método do seguinte modo:

```
public void setAge(int age){
        this.age = age;
}
```

Este método irá atribuir o valor do argumento **age** para a atributo de objeto **age** do objeto **StudentRecord**.

6. Overloading de Métodos

Nas nossas classes, podemos necessitar de criar métodos que tenham os mesmos nomes, mas que funcionem de maneira diferente dependendo dos argumentos que informamos. Esta capacidade é chamada de **overloading de métodos**.

Overloading de métodos permite que um método com o mesmo nome, entretanto com diferentes argumentos, possa ter implementações diferentes e retornar valores de diferentes tipos. Ao invés de inventar novos nomes todas as vezes, o overloading de métodos pode ser utilizado quando a mesma operação tem implementações diferentes.

Por exemplo, na nossa classe **StudentRecord**, queremos ter um método que mostre as informações sobre o estudante. Entretanto, queremos que o método **print** mostre dados diferentes dependendo dos argumentos que lhe informamos. Por exemplo, quando não enviamos qualquer argumento queremos que o método **print** mostre o nome, endereço e idade do estudante.

Quando passamos 3 valores **double**, queremos que o método mostre o nome e as notas do estudante.

Temos os seguintes métodos dentro da nossa classe **StudentRecord**:

```
public void print(){
      System.out.println("Name:" + name);
      System.out.println("Address:" + address);
      System.out.println("Age:" + age);
}



public void print(double eGrade, double mGrade, double sGrade) {
      System.out.println("Name:" + name);
      System.out.println("Math Grade:" + mGrade);
      System.out.println("English Grade:" + eGrade);
      System.out.println("Science Grade:" + sGrade);
}
```

Quando tentamos chamar estes métodos no método **main**, criado para a classe **StudantRecordExample:**

```
public static void main(String[] args) {
      StudentRecord annaRecord = new StudentRecord();
      annaRecord.setName("Anna");
      annaRecord.setAddress("Philippines");
      annaRecord.setAge(15);
```

```
        annaRecord.setMathGrade(80);
        annaRecord.setEnglishGrade(95.5);
        annaRecord.setScienceGrade(100);

        // overloading de métodos
        annaRecord.print();
        annaRecord.print(
        annaRecord.getEnglishGrade(),
        annaRecord.getMathGrade(),
        annaRecord.getScienceGrade());
    }
```

teremos a saída para a primeira chamada ao método **print**:

```
Name:Anna
Address:Philippines
Age:15
```

e, em seguida, a saída para a segunda chamada ao método **print**:

```
Name:Anna
Math Grade:80.0
English Grade:95.5
Science Grade:100.0
```

Lembre-se sempre que métodos **overload** possuem as seguintes

propriedades:

1. o mesmo nome
2. argumentos diferentes
3. tipo do retorno igual ou diferente

7. Declarando Construtores

Discutimos anteriormente o conceito de construtores. Construtores são importantes na criação de um objeto. É um método onde são colocadas todas as inicializações.

A seguir, temos as propriedades de um construtor:

1. Possuem o **mesmo nome da classe**

2. Construtor é um método, entretanto, somente as seguintes informações podem ser colocadas no cabeçalho do construtor:

    - Escopo ou identificador de acessibilidade (como **public**)

    - Nome do construtor

- Argumentos, caso necessário

3. Não retornam valor

4. São executados automaticamente na utilização do operador **new** durante a instanciação da classe

Para declarar um construtor, escrevemos:

```
[modificador] <nomeClasse> (<argumento>*) {
<instrução>*
}
```

7.1. Construtor Padrão (default)

Toda classe tem o seu construtor padrão. O **construtor padrão** é um construtor público e sem argumentos. Se não for definido um construtor para a classe, então, implicitamente, é assumido um construtor padrão.

Por exemplo, na nossa classe **StudentRecord**, o construtor padrão é definido do seguinte modo:

```
public StudentRecord() {
}
```

7.2. Overloading de Construtores

Como mencionamos, construtores também podem sofrer **overloading**, por exemplo, temos aqui quatro construtores:

```
public StudentRecord() {
      // qualquer código de inicialização aqui
}

public StudentRecord(String temp){
      this.name = temp;
}
public StudentRecord(String name, String address) {
      this.name = name;
      this.address = address;
}
public StudentRecord(double mGrade, double eGrade, double sGrade) {
      mathGrade = mGrade;
      englishGrade = eGrade;
      scienceGrade = sGrade;
}
```

7.3. Usando Construtores

Para utilizar estes construtores, temos as seguintes instruções:

```
public static void main(String[] args) {
      // criar três objetos para o registro do estudante
      StudentRecord annaRecord = new
```

```
        StudentRecord("Anna");
        StudentRecord beahRecord =
        new StudentRecord("Beah", "Philippines");
        StudentRecord crisRecord =
        new StudentRecord(80,90,100);
        // algum código aqui
    }
```

Antes de continuarmos, vamos retornar o atributo estático **studentCount** que declaramos agora a pouco. O objetivo de **studentCount** é contar o número de objetos que são instanciados com a classe **StudentRecord**. Então, o que desejamos é incrementar o valor de **studentCount** toda vez que um objeto da classe **StudentRecord** é instanciado. Um bom local para modificar e incrementar o valor de **studentCount** é nos construtores, pois são sempre chamados toda vez que um objeto é instanciado. Por exemplo:

```
public StudentRecord() {
        studentCount++; // adicionar um estudante
}
public StudentRecord(String name) {
        studentCount++; // adicionar um estudante
        this.name = name;
}
public StudentRecord(String name, String address) {
        studentCount++; // adicionar um estudante
        this.name = name;
        this.address = address;
}
public StudentRecord(double mGrade, double eGrade, double sGrade) {
        studentCount++; // adicionar um estudante
        mathGrade = mGrade;
        englishGrade = eGrade;
        scienceGrade = sGrade;
}
```

7.4. Utilizando o this()

Chamadas a construtores podem ser cruzadas, o que significa ser possível chamar um construtor de dentro de outro construtor. Usamos a chamada **this()** para isso. Por exemplo, dado o seguinte código,

```
public StudentRecord() {
        this("some string");
}
public StudentRecord(String temp) {
        this.name = temp;
}
public static void main( String[] args ) {
        StudentRecord annaRecord = new StudentRecord();
}
```

Dado o código acima, quando se executa a instrução do método **main**, será chamado o **primeiro** construtor. A instrução inicial deste construtor resultará na chamada ao **segundo** construtor.

Há algum detalhes que devem ser lembrados na utilização da chamada ao construtor por **this()**:

1. A chamada ao construtor **DEVE SEMPRE OCORRER NA PRIMEIRA LINHA DE INSTRUÇÃO**

2. **UTILIZADO PARA A CHAMADA DE UM CONSTRUTOR**. A chamada ao **this()** pode ser seguida por outras instruções.

Como boa prática de programação, é ideal nunca construir métodos que repitam as instruções. Buscamos a utilização de **overloading** com o objetivo de evitarmos essa repetição. Deste modo, reescreveremos os construtores da classe **StudentRecord** para:

```
public StudentRecord() {
      studentCount++; // adicionar um estudante
}
public StudentRecord(String name) {
      this();
      this.name = name;
}
public StudentRecord(String name, String address) {
      this(name);
      this.address = address;
}
public StudentRecord(double mGrade, double eGrade, double sGrade) {
      this();
      mathGrade = mGrade;
      englishGrade = eGrade;
      scienceGrade = sGrade;
}
```

8. Pacotes

São utilizados para agrupar classes e interfaces relacionadas em uma única unidade (discutiremos interfaces mais tarde). Esta é uma característica poderosa que oferece um mecanismo para gerenciamento de um grande grupo de classes e interfaces e evita possíveis conflitos de nome.

8.1. Importando Pacotes

Para utilizar classes externas ao pacote da classe, é necessário importar os pacotes dessas classes. Por padrão, todos as suas classes Java importam o pacote **java.lang**. É por isso que é possível utilizar classes como *String* e *Integer* dentro da sua classe, mesmo não tendo importado nenhum pacote explicitamente.

A sintaxe para importar pacotes é como segue:

```
import <nomeDoPacote>.<nomeDaClasse>;
```

Por exemplo, necessitar utilizar a classe **Color** dentro do pacote **awt**, é necessário a seguinte instrução:

```
import java.awt.Color;
```

ou:

```
import java.awt.*;
```

A primeira linha de instrução importa especificamente a classe **Color** enquanto que a seguinte importa todas as classes do pacote **java.awt**.

Outra maneira de importar classes de outros pacotes é através da referência explícita ao pacote. Isto é feito utilizando-se o nome completo do pacote para declaração do objeto na classe:

```
java.awt.Color color;
```

8.2. Criando pacotes

Para criar os nossos pacotes, escrevemos:

```
package <nomeDoPacote>;
```

Suponha que desejamos criar um pacote onde colocaremos a nossa classe **StudentRecord** juntamente com outras classes relacionadas. Chamaremos o nosso pacote de **schoolClasses**.

A primeira coisa que temos que fazer é criar uma pasta chamada **schoolClasses**. Em seguida, copiar para esta pasta todas as classes que pertençam a este pacote. Adicione a seguinte instrução no arquivo da classe, esta linha deve ser colocada antes da definição da classe. Por exemplo:

```
package schoolClasses;
      public class StudentRecord {
      // instruções da classe
}
```

Pacotes podem ser aninhados. Neste caso, o interpretador espera que a estrutura de diretórios contendo as classes combinem com a hierarquia dos pacotes.

8.3. Definindo a variável de ambiente CLASSPATH

Suponha que colocamos o pacote **schoolClasses** sob o diretório C:\.

Precisamos que a **classpath** aponte para este diretório de tal forma que quando executemos a classe, a JVM seja capaz de enxergar onde está armazenada.

Antes de discutirmos como ajustar a variável **classpath**, vamos ver um exemplo sobre o que aconteceria se esta não fosse ajustada.

Suponha que sigamos os passos para compilar e executar a classe **StudentRecord** que escrevemos:

```
C:\schoolClasses>javac StudentRecord.java
C:\schoolClasses>java StudentRecord
Exception in thread "main" java.lang.NoClassDefFoundError: StudentRecord
(wrong name: schoolClasses/StudentRecord)
```

```
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(Unknown Source)
at java.security.SecureClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.defineClass(Unknown Source)
at java.net.URLClassLoader.access$100(Unknown Source)
at java.net.URLClassLoader$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClass(Unknown Source)
at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

Surge o erro **NoClassDefFoundError**, que significa que o Java desconhece onde procurar por esta classe. A razão disso é que a sua classe **StudentRecord** pertence a um pacote denominado **schoolClasses**. Se desejamos executar esta classe, teremos que dizer ao Java o seu nome completo **schoolClasses.StudentRecord**. Também teremos que dizer à JVM onde procurar pelos nossos pacotes, que, neste caso, é no C:\. Para fazer isso, devemos definir a variável **classpath**.

Para definir a variável **classpath** no Windows, digitamos o seguinte na linha de comando:

```
C:\schoolClasses>set classpath=C:\
```

onde C:\ é o diretório onde colocamos os pacotes. Após definir a variável **classpath**, poderemos executar a nossa classe em qualquer pasta, digitando:

```
C:\schoolClasses>java schoolClasses.StudentRecord
```

Para sistemas baseados no Unix, suponha que as nossas classes estejam no diretório **/usr/local/myClasses**, escrevemos:

```
export classpath=/usr/local/myClasses
```

Observe que é possível definir a variável **classpath** em qualquer lugar. É possível definir mais de um local de pesquisa; basta separá-los por ponto-evírgula (no Windows) e dois-pontos (nos sistemas baseados em Unix). Por exemplo:

```
set classpath=C:\myClasses;D:\;E:\MyPrograms\Java
```

e para sistemas baseados no Unix:

```
export classpath=/usr/local/java:/usr/myClasses
```

9. Modificadores de Acesso

Quando estamos criando as nossas classes e definindo as suas propriedades e métodos, queremos implementar algum tipo de restrição para se acessar esses dados. Por exemplo, ao necessitar que um certo atributo seja modificado apenas pelos métodos dentro da classe, é possível esconder isso dos outros objetos que estejam usando a sua classe. Para implementar isso, no Java, temos os **modificadores de acesso**.

Existem quatro diferentes tipos de modificadores de acesso: **public**, **private**, **protected** e **default**. Os três primeiros modificadores são escritos explicitamente no código para indicar o acesso, para o tipo **default**, não se utiliza nenhuma palavra-chave.

9.1. Acesso padrão

Especifica que os elementos da classe são acessíveis somente aos métodos internos da classe e às suas subclasses. Não há palavra chave para o modificador **default**; sendo aplicado na ausência de um modificador de acesso. Por exemplo :

```
public class StudentRecord {
      // acesso padrão ao atributo
      int name;
      // acesso padrão para o método
      String getName(){
            return name;
      }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados somente por métodos internos à classe e por subclasses de **StudentRecord**.

Falaremos sobre subclasses em próximas lições.

9.2. Acesso público

Especifica que os elementos da classe são acessíveis tanto internamente quanto externamente à classe. Qualquer objeto que interage com a classe pode ter acesso aos elementos públicos da classe. Por exemplo:

```
public class StudentRecord {
      // acesso público o atributo
      public int name;
      // acesso público para o método
      public String getName(){
            return name;
      }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados a partir de outros objetos.

9.3. Acesso protegido

Especifica que somente classes no mesmo pacote podem ter acesso aos atributos e métodos da classe. Por exemplo:

```
public class StudentRecord {
      //acesso protegido ao atributo
      protected int name;
      //acesso protegido para o método
      protected String getName(){
      return name;
      }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados por outros objetos, desde que o objetos pertençam ao mesmo pacote da classe **StudentRecord**.

9.4. Acesso particular

Especifica que os elementos da classe são acessíveis somente na classe que o definiu. Por exemplo:

```
public class StudentRecord {
      // acesso particular ao atributo
      private int name;
      // acesso particular para o método
      private String getName(){
            return name;
      }
}
```

O atributo de objeto **name** e o método **getName()** podem ser acessados somente por métodos internos à classe.

Dicas de programação:

1. Normalmente, os atributos de objeto de uma classe devem ser declarados particulares e a classe pode fornecer métodos assessores e modificadores para estes.

10. Exercícios

10.1. Registro de Agenda

Sua tarefa é criar uma classe que contenha um Registro de Agenda. A **tabela 1** descreve as informações que um Registro de Agenda deve conter:

Atributos/Propriedades	Descrição
Nome	Nome da pessoa
Endereço	Endereço da pessoa
Número de Telefone	Número de telefone da pessoa
email	Endereço eletrônico da pessoa

Tabela 1: Atributos e Descrições dos Atributos

Crie os seguintes métodos:

1. Forneça todos os métodos assessores e modificadores necessários para todos os atributos.

2. Construtores.

10.2. Agenda

Crie uma classe Agenda que possa conter entradas de objetos tipo Registro de Agenda (utilize a classe criada no primeiro exercício). Devem ser oferecidos os seguintes métodos para a agenda:

1. Adicionar registro
2. Excluir registro
3. Visualizar registros
4. Modificar um registro

Lição 9

1. Objetivos

Nesta lição será discutido como uma classe pode herdar as propriedades de outra classe jáexistente. Uma classe que faz isso é chamada de subclasse, e sua classe pai é chamada de superclasse. Será também discutida a aplicação automática dos métodos de cada objeto, independente da subclasse deste. Esta propriedade é conhecida como polimorfismo. E, por último, discutiremos sobre interfaces, que ajudam a reduzir no esforço de programação.

Ao final desta lição, o estudante será capaz de:

- Definir superclasses e subclasses
- Criar override de métodos das superclasses
- Criar métodos final e classes final

2. Herança

Todas as classes, incluindo as que compõem a API Java, são subclasses da classe Object. Um exemplo de hierarquia de classes é mostrado com a figura 1. A partir de uma determinada classe, qualquer classe acima desta na hierarquia de classes é conhecida como uma **superclasse** (ou classe Pai). Enquanto que qualquer classe abaixo na hierarquia de classes é conhecia como uma **subclasse** (ou classe Filho).

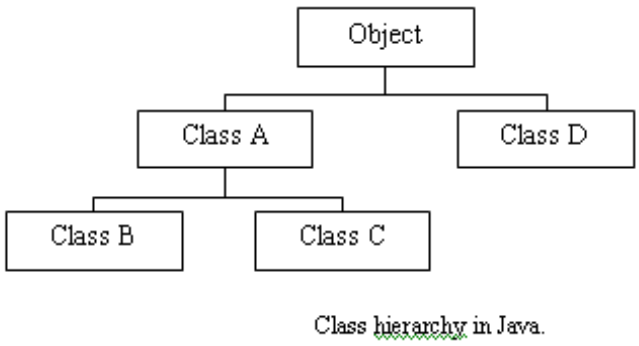


Figura 1: Hierarquia de Classes

Herança é um dos principais princípios em orientação a objeto. Um comportamento (método) é definido e codificado uma única vez em uma única classe e este comportamento é herdado por todas suas subclasses. Uma subclasse precisa apenas implementar as diferenças em relação a sua classe pai, ou seja, adaptar-se ao meio em que vive.

2.1. Definindo Superclasses e Subclasses

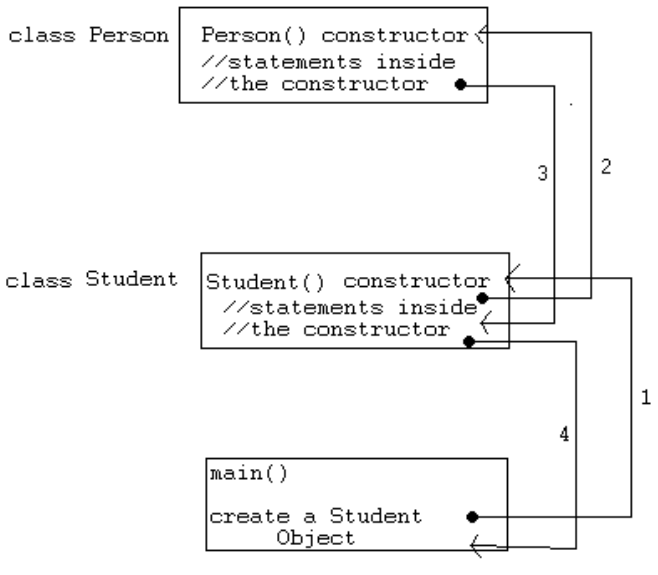
Para herdar uma classe usamos a palavra-chave **extends.** Ilustraremos criando uma classe pai de exemplo. Suponha que tenhamos uma classe pai chamada **Person**.

```
public class Person {
	protected String name;
	protected String address;
	/**
	* Construtor Padrão
	*/
	public Person(){
		System.out.println("Inside Person:Constructor");
		name = "";
		address = "";
	}
	/**
	* Construtor com 2 parâmetros
	*/
	public Person( String name, String address ){
		this.name = name;
		this.address = address;
	}
	/**
	* Métodos modificadores e acessores
	*/
	public String getName(){
		return name;
	}
	public String getAddress(){
		return address;
	}
	public void setName( String name ){
		this.name = name;
	}
	public void setAddress( String add ){
		this.address = add;
	}
}
```

Os atributos **name** e **address** são declarados como **protected**. A razão de termos feito isto é que queremos que estes atributos sejam acessíveis às subclasses dessa classe. Se a declararmos com o modificador **private**, as subclasses não estarão aptas a usá-los. Todas as propriedades de uma superclasse que são declaradas como **public**, **protected** e **default** podem ser acessadas por suas subclasses.

Vamos criar outra classe chamada **Student**. E, como um estudante também é uma pessoa, concluímos que iremos estender a classe **Person**, então, poderemos herdar todas as propriedades existêntes na classe **Person**. Para isto, escrevemos:

```
public class Student extends Person {
	public Student(){
		System.out.println("In-
	side Student:Constructor");
		//Algum código aqui
	}
	// Algum código aqui
```



```
}
```

O fluxo de controle é mostrado na figura 2.

Quando a classe **Student** for instanciada, o construtor padrão da superclasse **Person** é invocado implicitamente para fazer as inicializações necessárias. Após isso, as instruções dentro do construtor da subclasse são executadas. Para ilustrar, considere o seguinte código:

```
public static void main( String[] args ){
      Student anna = new Student();
}
```

No código, criamos um objeto da classe **Student**. O resultado da execução deste programa é:

```
Inside Person:Constructor
Inside Student:Constructor
```

2.2. super

Uma subclasse pode, explicitamente, chamar um construtor de sua superclasse imediata. Isso é feito utilizando uma chamada ao objeto **super**. Uma chamada ao **super** no construtor de uma subclasse irá resultar na execução de um construtor específico da superclasse baseado nos argumentos passados.

Por exemplo, dada a seguinte instrução para a classe **Student**:

```
public Student(){
      super( "SomeName", "SomeAddress" );
      System.out.println("Inside Student:Constructor");
}
```

Este código chama o segundo construtor de sua superclasse imediata (a classe **Person**) e a executa. Outro código de exemplo é mostrado abaixo:

```
public Student(){
      super();
      System.out.println("Inside Student:Constructor");
}
```

Este código chama o construtor padrão de sua superclasse imediata (a classe **Person**) e o executa. Devemos relembrar, quando usamos uma chamada ao objeto **super**:

1. A instrução **super()** DEVE SER A PRIMEIRA INSTRUÇÃO EM UM CONSTRUTOR.

2. As instruções **this()** e **super()** não podem ocorrer simultaneamente no mesmo construtor. O objeto **super** é uma referência aos membros da superclasse (assim como o objeto **this** é da sua própria classe). Por exemplo:

```
public Student() {
      super.name = "person name"; // Nome da classe pai
      this.name = "student name"; // Nome da classe atual
}
```

2.3. Override de Métodos

Se, por alguma razão, uma classe derivada necessita que a implementação de algum método seja diferente da superclasse, o **polimorfismo por override** pode vir a ser muito útil. Uma subclasse pode modificar um método definido em sua superclasse fornecendo uma nova implementação para aquele método.

Supondo que tenhamos a seguinte implementação para o método **getName** da superclasse **Person**:

```
public class Person {
      ...
      public String getName(){
            System.out.println("Parent: getName");
            return name;
      }
...
}
```

Para realizar um polimorfismo por override no método **getName** da subclasse **Student**, escrevemos:

```
public class Student extends Person {
      ...
      public String getName(){
            System.out.println("Student: getName");
            return name;
      }
      ...
}
```

Então, quando invocarmos o método **getName** de um objeto da classe **Student**, o método chamado será o de **Student**, e a saída será:

```
Student: getName
```

É possível chamar o método **getName** da superclasse, basta para isso:

```
public class Student extends Person {
...
public String getName() {
super.getName();
System.out.println("Student: getName");
```

```
return name;
}
...
}
```

Inserimos uma chamada ao objeto super e a saída será:

```
Parent: getName
Student: getName
```

2.4. Métodos final e Classes final

Podemos declarar classes que não permitem a herança. Estas classes são chamadas classes finais. Para definir que uma classe seja final, adicionamos a palavra-chave **final** na declaração da classe (na posição do modificador). Por exemplo, desejamos que a classe **Person** não possa ser herdada por nenhuma outra classe, escrevemos:

```
public final class Person {
      // Código da classe aqui
}
```

Muitas classes na API Java são declaradas **final** para certificar que seu comportamento não seja herdado e, possivelmente , modificado. Exemplos, são as classes Integer, Double e Math.

Também é possível criar métodos que não possam ser modificados pelos filhos, impedindo o polimorfismo por override. Estes métodos são o que chamamos de métodos finais. Para declarar um método final, adicionamos a palavra-chave final na declaração do método (na posição do modificador). Por exemplo, se queremos que o método getName da classe **Person** não possa ser modificado, escrevemos:

```
public final String getName(){
      return name;
}
```

Caso o programador tente herdar uma classe final, ocorrerá um erro de compilação. O mesmo acontecerá ao se tentar fazer um **override** de um método final.

3. Polimorfismo

Considerando a classe pai **Person** e a subclasse **Student** do exemplo anterior, adicionaremos outra subclasse a **Person**, que se chamará **Employee**. Abaixo está a hierarquia de classes que ilustra o cenário:

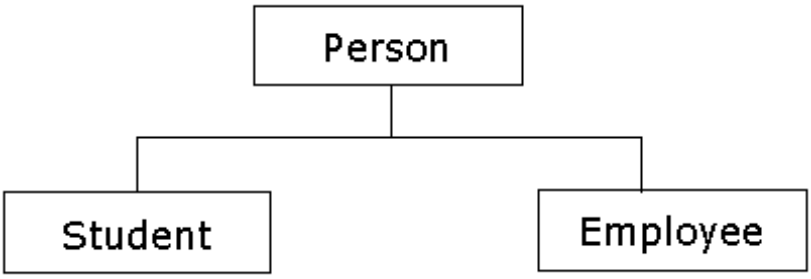


*Figura 3: Hieraquia para a classe **Person** e suas subclasses.*

Podemos criar uma referência do tipo da superclasse para a subclasse. Por exemplo:

```
public static main( String[] args ) {
      Person ref;
      Student studentObject = new Student();
      Employee employeeObject = new Employee();
      ref = studentObject; //Person ref: ponteiro para um Student
      // algum código aqui
}
```

Supondo que tenhamos um método **getName** em nossa superclasse **Person**, iremos realizar uma modificação deste nas subclasses **Student** e **Employee**:

```
public class Person {
      public String getName(){
            System.out.println("Person Name:" + name);
            return name;
      }
}
public class Student extends Person {
      public String getName(){
            System.out.println("Student Name:" + name);
            return name;
      }
}
public class Employee extends Person {
      public String getName(){
            System.out.println("Employee Name:" + name);
            return name;
      }
}
```

Voltando ao método **main**, quando tentamos chamar o método **getName** da referência **ref** do tipo **Person**, o método **getName** do objeto **Student** será chamado. Agora, se atribuirmos **ref** ao objeto **Employee**, o método **getName** de **Employee** será chamado.

```
public static main(String[] args) {
      Person ref;
      Student studentObject = new Student();
      Employee employeeObject = new Employee();
      ref = studentObject; //ponteiro de referência para um Student
```

```
        String temp = ref.getName(); //getName de Student é chamado
        System.out.println(temp);
        ref = employeeObject; // ponteiro de referência Person para um
        // objeto Employee
        String temp = ref.getName(); // getName de Employee
        // classe é chamada
        System.out.println(temp);
}
```

A capacidade de uma referência mudar de comportamento de acordo com o objeto a que se refere é chamada de **polimorfismo**. O polimorfismo permite que múltiplos objetos de diferentes subclasses sejam tratados como objetos de uma única superclasse, e que automaticamente sejam selecionados os métodos adequados a serem aplicados a um objeto em particular, baseado na subclasse a que ele pertença.

Outro exemplo que demonstra o polimorfismo é realizado ao passar uma referência a métodos. Supondo que exista um método estático **printInformation** que recebe como parâmetro um objeto do tipo **Person**, pode-se passar uma referência do tipo **Employee** e do tipo **Student**, porque são subclasses do tipo **Person**.

```
public static main(String[] args) {
      Student studentObject = new Student();
      Employee employeeObject = new Employee();
      printInformation(studentObject);
      printInformation(employeeObject);
}
public static printInformation(Person p){
...
}
```

4. Classes Abstratas

Para criar métodos em classes devemos, necessariamente, saber qual o seu comportamento. Entretanto, em muitos casos não sabemos como estes métodos se comportarão na classe que estamos criando, e, por mera questão de padronização, desejamos que as classes que herdem desta classe possuam, obrigatoriamente, estes métodos.

Por exemplo, queremos criar uma superclasse chamada **LivingThing**. Esta classe tem certos métodos como **breath**, **sleep** e **walk**. Entretanto, existem tantos métodos nesta superclasse que não podemos generalizar este comportamento. Tome por exemplo, o método **walk** (andar). Nem todos os seres vivos andam da mesma maneira. Tomando os humanos como exemplo, os humanos andam sobre duas pernas, enquanto que outros seres vivos como os cães andam sobre quatro. Entretanto, existem muitas características que os seres vivos têm em comum, isto é o que nós queremos ao criar uma superclasse geral.

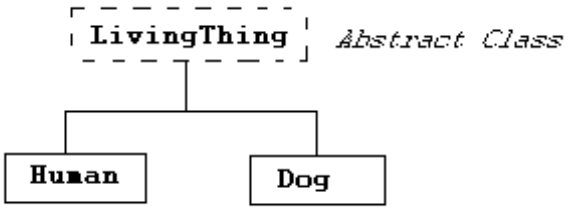


Figura 4: Classe Abstrata

Para realizarmos isto, teremos que criar uma superclasse que possua alguns métodos com implementações e outros não. Este tipo de classe é chamada de **classe abstrata**.

Uma **classe abstrata** é uma classe que não pode gerar um objeto. Freqüentemente aparece no topo de uma hierarquia de classes no modelo de programação orientada a objetos.

Os métodos nas **classes abstratas** que não têm implementação são chamados de **métodos abstratos**. Para criar um método abstrato, apenas escreva a assinatura do método sem o corpo e use a palavra-chave **abstract**. Por exemplo:

```
public abstract void someMethod();
```

Agora, vamos criar um exemplo de classe abstrata:

```
public abstract class LivingThing {
      public void breath(){
            System.out.println("Living Thing breathing...");
      }
public void eat(){
      System.out.println("Living Thing eating...");
}
/**
* método abstrato walk
* Queremos que este método seja criado pela
* subclasse de LivingThing
*/
public abstract void walk();
}
```

Quando uma classe estende a classe abstrata **LivingThing**, ela é obrigada a implementar o método abstrato **walk**. Por exemplo:

```
      public class Human extends LivingThing {
            public void walk(){
            System.out.println("Human walks...");
            }
```

```
}
```

Se a classe Human não implementar o método **walk**, será mostrada a seguinte mensagem de erro de compilação:

```
Human.java:1: Human is not abstract and does not override abstract
method walk() in LivingThing
public class Human extends LivingThing
^
1 error
```

Dicas de programação:
1. Use classes abstratas para definir muitos tipos de comportamentos no topo de uma hierarquia de classes de programação orientada a objetos. Use suas subclasses para prover detalhes de implementação da classe abstrata.

5. Interfaces

Uma interface é um tipo especial de classe que contém unicamente métodos abstratos ou atributos finais. Interfaces, por natureza, são abstratas.

Interfaces definem um padrão e o caminho público para especificação do comportamento de classes. Permitem que classes, independente de sua localização na estrutura hierárquica, implementem comportamentos comuns.

5.1. Porque utilizar Interfaces?

Utilizamos interfaces quando queremos classes não relacionadas que implementem métodos similares. Através de interfaces, podemos obter semelhanças entre classes não relacionadas sem forçar um relacionamento artificial entre elas.

Tomemos como exemplo a classe **Line**, contém métodos que obtém o tamanho da linha e compara o objeto **Line** com objetos de mesma classe. Considere também que tenhamos outra classe, **MyInteger**, que contém métodos que comparam um objeto **MyInteger** com objetos da mesma classe. Podemos ver que ambas classes têm os mesmos métodos similares que os comparam com outros objetos do mesmo tipo, entretanto eles não são relacionados. Para se ter certeza de que essas classes implementem os mesmos métodos com as mesmas assinaturas, utilizamos as interfaces. Podemos criar uma interface **Relation** que terá declarada algumas assinaturas de métodos de comparação. A interface **Relation** pode ser implementada da seguinte forma:

```
public interface Relation {
      public boolean isGreater(Object a, Object b);
      public boolean isLess(Object a, Object b);
      public boolean isEqual(Object a, Object b);
}
```

Outra razão para se utilizar interfaces na programação de objetos é revelar uma interface de programação de objeto sem revelar essas classes. Como veremos mais adiante, podemos utilizar uma interface como tipo de dados.

Finalmente, precisamos utilizar interfaces como mecanismo alternativo para herança múltipla, que permite às classes em ter mais de uma superclasse. A herança múltipla não está implementada em Java.

5.2. Interface vs. Classe Abstrata

A principal diferença entre uma **interface** e uma **classe abstrata** é que a classe abstrata pode possuir métodos implementados (reais) ou não implementados (abstratos). Na interface, todos os métodos são obrigatoriamente abstratos e públicos, tanto que para esta, a palavrachave **abstract** ou **public** é opcional.

5.3. Interface vs. Classe

Uma característica comum entre uma interface e uma classe é que ambas são tipos. Isto significa que uma interface pode ser usada no lugar onde uma classe é esperada. Por exemplo, dadas a classe **Person** e a interface **PersonInterface**, as seguintes declarações são válidas:

```
PersonInterface pi = new Person();
Person pc = new Person();
```

Entretanto, não se pode criar uma instância de uma interface sem implementá-la. Um exemplo disso é:

```
PersonInterface pi = new PersonInterface(); //ERRO DE
//COMPILAÇÃO !!!
```

Outra característica comum é que ambas, interfaces e classes, podem definir métodos, embora uma interface não possa tê-los implementados. Já uma classe pode.

5.4. Criando Interfaces

Para criarmos uma interface, utilizamos:

```
[public] [abstract] interface <NomeDaInterface> {
      < [public] [final] <tipoAtributo> <atributo> = <valorInicial>; >*
      < [public] [abstract] <retorno> <nomeMetodo>(<parametro>*); >*
}
```

Como exemplo, criaremos uma interface que define o relacionamento entre dois objetos de acordo com a "ordem natural" dos objetos:

```
interface Relation {
      boolean isGreater(Object a, Object b);
      boolean isLess(Object a, Object b);
      boolean isEqual( Object a, Object b);
}
```

Para implementar esta **interface**, usaremos a palavra chave "implements". Por exemplo:

```
/**
* Esta classe define um segmento de linha
*/
public class Line implements Relation {
      private double x1;
      private double x2;
      private double y1;
      private double y2;
      public Line(double x1, double x2, double y1, double y2) {
            this.x1 = x1;
            this.x2 = x2;
            this.y2 = y2;
            this.y1 = y1;
      }
      public double getLength(){
            double length = Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
            return length;
      }
      public boolean isGreater( Object a, Object b){
            double aLen = ((Line)a).getLength();
            double bLen = ((Line)b).getLength();
            return (aLen > bLen);
      }
      public boolean isLess( Object a, Object b){
            double aLen = ((Line)a).getLength();
            double bLen = ((Line)b).getLength();
            return (aLen < bLen);
      }
      public boolean isEqual( Object a, Object b){
            double aLen = ((Line)a).getLength();
            double bLen = ((Line)b).getLength();
            return (aLen == bLen);
      }
}
```

Quando a classe implementa uma interface, deve-se implementar **todos** os métodos desta, caso contrário será mostrado o erro:

```
Line.java:4: Line is not abstract and does not override abstract
method isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
^
1 error
```

Dicas de programação:

1. Use interface para criar uma definição padrão de métodos em classes diferentes. Uma vez que o conjunto de definições de métodos é criado, e pode ser escrito um método simples para manipular todas as classes que implementam a interface.

5.5. Relacionamento de uma Interface para uma Classe

Como vimos nas seções anteriores, a classe pode implementar uma interface e para isso prover o código de implementação para todos os métodos definidos na interface. Outro detalhe a se notar na relação entre uma interface e uma classe. A classe pode apenas estender uma única superclasse, mas pode implementar diversas interfaces. Um exemplo de uma classe que implementa diversas interfaces:

```
public class Person implements PersonInterface, LivingThing, WhateverInterface
{
      //algumas linhas de código
}
```

Outro exemplo de uma classe que estende de outra superclasse e implementa interfaces:

```
public class ComputerScienceStudent extends Student
      implements PersonInterface, LivingThing {
      // algumas linhas de código
}
```

Uma interface não é parte de uma hierarquia de classes. Classes não relacionadas podem implementar a mesma interface.

5.6. Herança entre Interfaces

Interfaces não são partes de uma hierarquia de classes. Entretanto, interfaces podem ter relacionamentos entre si. Por exemplo, suponha que tenhamos duas interfaces, **StudentInterface** e **PersonInterface**. Se **StudentInterface** estende **PersonInterface**, esta herda todos os métodos declarados em **PersonInteface**.

```
public interface PersonInterface {
...
}

public interface StudentInterface extends PersonInterface {
...
}
```

6. Exercícios

6.1. Estendendo StudentRecord

Neste exercício, queremos criar um registro mais especializado de **Student** que contém informações adicionais sobre um estudante de Ciência da Computação. Sua tarefa é estender a classe **StudentRecord** que foi implementada nas lições anteriores e acrescentar atributos e métodos que são necessários para um registro de um estudante de Ciência da Computação.

Utilize **override** para modificar alguns métodos da superclasse **StudentRecord**, caso seja necessário.

6.2. A classe abstrata Shape

Crie uma classe abstrata chamada **Shape** com os métodos abstratos **getArea()** e **getName()**. Escreva duas de suas subclasses **Circle** e **Square**. E acrescente métodos adicionais a estas subclasses.

Lição 10

1. Objetivos

Nesta lição, iremos aprender uma técnica utilizada em Java para tratar condições incomuns que interrompem a operação normal da classe. Esta técnica é chamada de tratamento de exceção.

Ao final desta lição, o estudante será capaz de:

- Definir o que são exceções
- Tratar exceções utilizando **try- catch- finally**

2. O que são Exceções (Exception)?

Uma exceção é um evento que interrompe o fluxo normal de processamento de uma classe. Este evento é um erro de algum tipo. Isto causa o término anormal da classe. Estes são alguns dos exemplos de exceções que podem ter ocorridos em exercícios anteriores:

- **ArrayIndexOutOfBoundsException**, ocorre ao acessar um elemento inexistente de um array.

- **NumberFormatException**, ocorre ao enviar um parâmetro não-numérico para o método **Integer.parseInt()**.

3. Tratando Exceções

Para tratar exceções em Java utilizamos a declaração **try- catch- finally**. O que devemos fazer para proteger as instruções passíveis de gerar uma exceção, é inserí-las dentro deste bloco.

A forma geral de um **try- catch- finally** é:

```
try{
      // escreva as instruções passíveis de gerar uma exceção
      // neste bloco
} catch (<exceptionType1> <varName1>){
      // escreva a ação que o seu programa fará caso ocorra
      // uma exceção de um determinado
} . . .
} catch (<exceptionTypen> <varNamen>){
      // escreva a ação que o seu programa fará caso ocorra
      // uma exceção de um determinado tipo
} finally {
      // escreva a ação que o seu programa executará caso ocorra
      // ou não um erro ou exceção
}
```

Exceções geradas durante a execução do bloco **try** podem ser detectadas e tratadas num bloco **catch**. O código no bloco **finally** é sempre executado, ocorrendo ou não a exceção.

A seguir são mostrados os principais aspectos da sintaxe da construção de um **try- catchfinally**:

- A notação de bloco é obrigatória.
- Para cada bloco **try**, pode haver um ou mais blocos **catch**, mas somente um bloco **finally**.
- Um bloco **try** deve que ser seguido de PELO MENOS um bloco **catch** OU um bloco **finally**, ou ambos.
- Cada bloco **catch** define o tratamento de uma exceção.
- O cabeçalho do bloco **catch** recebe somente um argumento, que é a exceção (Exception) que este bloco pretende tratar.
- A exceção deve ser da classe **Throwable** ou de uma de suas subclasses. Para um melhor entendimento, observe a figura 1 que demonstra o fluxo seguido pelo **trycatch- finally**:

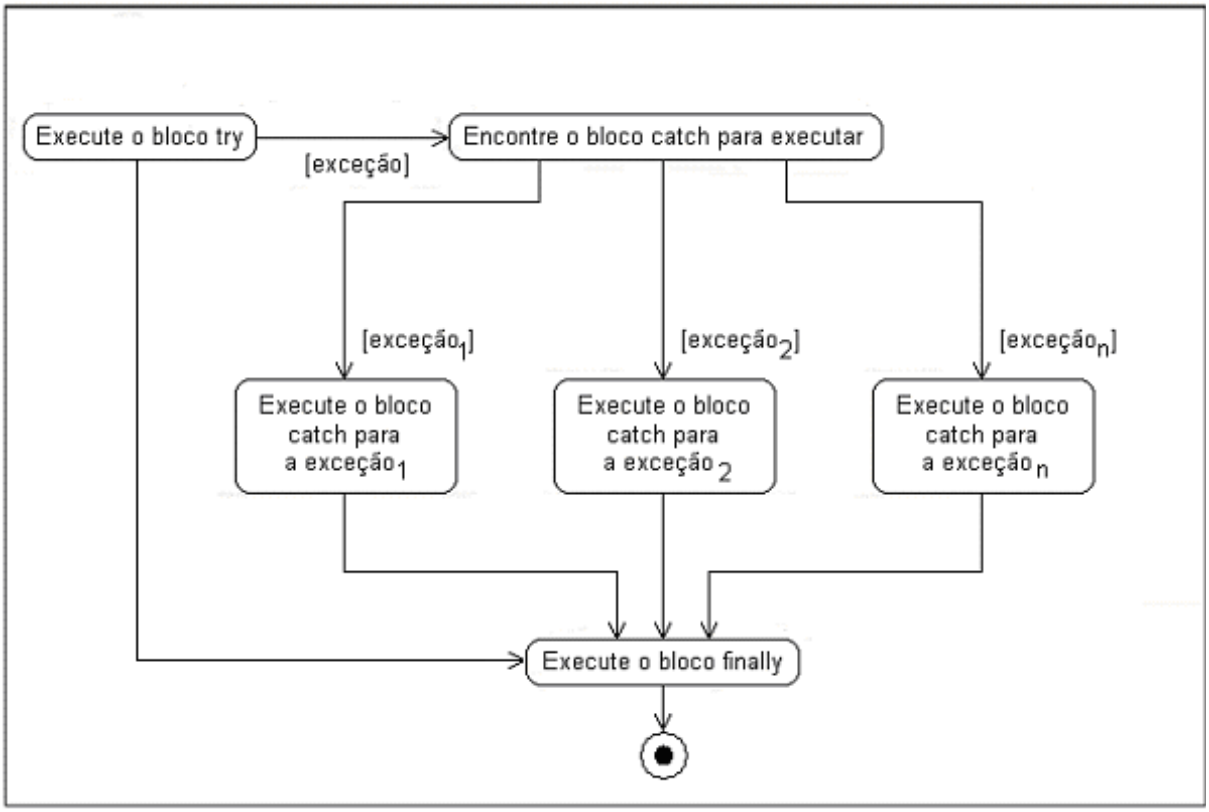


Figura 1: Fluxo em um **try- catch- finally**

Tomemos, por exemplo, uma classe que imprime o segundo argumento passado através da linha de comandos. Supondo que não há verificação no código para o número de argumentos.

```
public class ExceptionExample {
      public static void main( String[] args ) {
            System.out.println(args[1]);
            System.out.println("Finish");
      }
}
```

Ao executar esta classe sem informar nenhum argumento e, ao tentar acessar diretamente, conforme o exemplo descrito, o segundo argumento args[1], uma exceção é obtida que interromperá a execução normal do programa, e a seguinte mensagem será mostrada:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
at ExceptionExample.main(ExceptionExample.java:5)
```

Para prevenir que isto ocorra, podemos colocar o código dentro de um bloco **try- catch**. O bloco **finally** é opcional. Neste exemplo, não utilizaremos o bloco **finally**.

```
public class ExceptionExample{
      public static void main( String[] args ){
      try {
            System.out.println( args[1] );
      } catch (ArrayIndexOutOfBoundsException exp) {
            System.out.println("Exception caught!");
      }
            System.out.println("Finish");
      }
}
```

Assim, quando tentarmos rodar o programa novamente sem a informação dos argumentos, a saída trataria a exceção e o fluxo do programa não seria interrompido, mostrando o resultado:

```
Exception caught!
Finish
```

4. Exercícios

4.1. Capturando Exceções 1

Dada a seguinte classe:

```
public class TestException {
      public static void main(String[] args) {
            for (int i=0; true; i++) {
                  System.out.println("args["+i+"]="+ args[i]);
            }
            System.out.println("Quiting...");
      }
}
```

Compile e rode a classe **TestException**. E como saída será:

```
java TestExceptions one two three
      args[0]=one
      args[1]=two
      args[2]=three
      Exception in thread "main"
      java.lang.ArrayIndexOutOfBoundsException: 3
      at TestExceptions.main(1.java:4)
```

Modifique a classe **TestException** para tratar esta exceção. A saída depois do tratamento da exceção deverá ser:

```
java TestExceptions one two three
      args[0]=one
      args[1]=two
      args[2]=three
      Exception caught: java.lang.ArrayIndexOutOfBoundsException: 3
      Quiting...
```

4.2. Capturando Exceções 2

Há uma boa chance de que algumas classes escritas anteriormentes tenham disparados exceções. Como as exceções não foram tratadas, simplesmente interromperam a execução.

Retorne a estes programas e implemente o tratamento de exceções.

Apêndice A

Instalação do Java e do NetBeans

1. Objetivos

Neste apêndice, veremos como instalar o Java, versão JDK, e o NetBeans no seu sistema (Ubuntu Dapper/Windows). Efetue os downloads na página web da Sun Microsystems no endereço http://java.sun.com para o JDK 5.0.x e na página web do projeto NetBeans no endereço http://www.netbeans.org/downloads para o NetBeans 5.5. Antes de começar a instalação, copie, primeiramente, os arquivos instaladores para seu disco rígido.

Para Ubuntu Dapper:

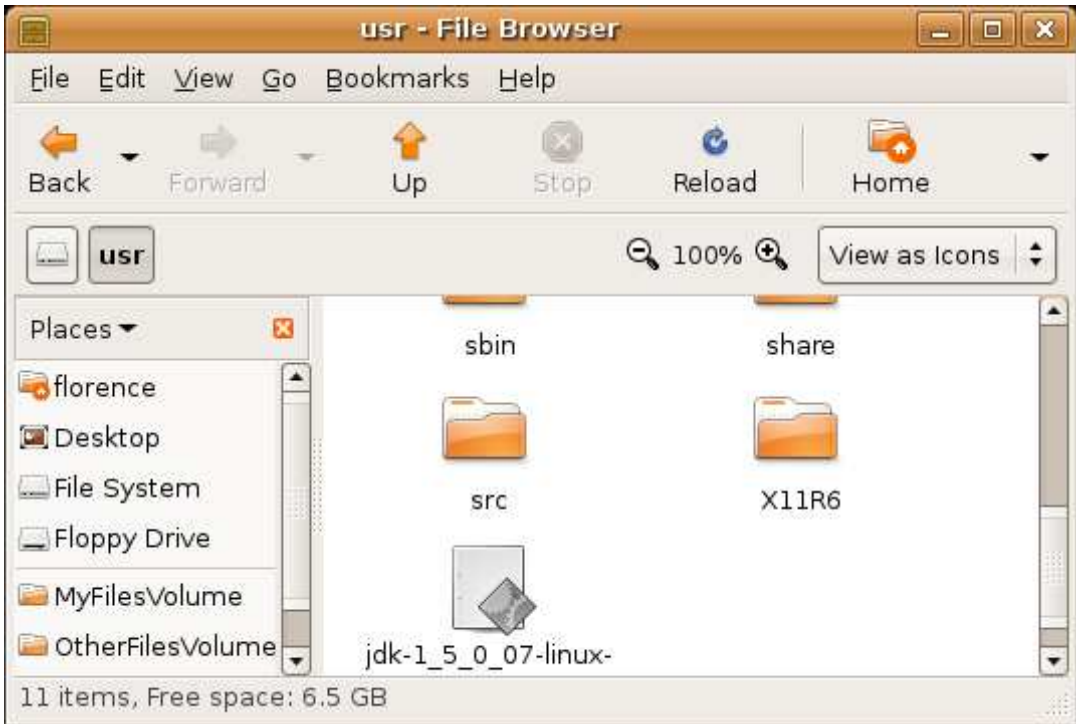
Copie todos os instaladores para a pasta /usr.

Para Windows:

Copie os instaladores em qualquer diretório temporário.

2. Instalando Java no Ubuntu Dapper

Passo 1: No diretório onde foi efetuado o download dos instaladores.



Passo 2: Antes de executar o instalador, assegure-se de que o arquivo seja um executável. Para tanto, pressione o botão direito do mouse no ícone do instalador, e em seguida selecione **Properties**. Selecione na aba **Permissions**, e então marque a opção **Execute**. Feche a janela.

Passo 3: Duplo-clique no arquivo **jdk-1_5_0_07-linux-i586.bin**. A caixa de diálogo abaixo será mostrada. Pressione o botão **Run in Terminal**.

No console será mostrado o contrato de licença do software. Pressione ENTER até ser mostrada a pergunta: **Do you agree to the above license terms? [yes or no]**. Caso concorde com os termos apresentados digite a palavra **yes** e pressione a tecla **ENTER**. Aguarde que o instalador termine de descompactar e instale o Java.

Passo 4: Devemos um caminho de pesquisa a fim de permitir a execução de comandos java em qualquer local. Para isto, entraremos na pasta `/usr/local/bin`. Digitando:

```
cd /usr/local/bin
```

Para criar os links simbólicos para os comandos, tecle:

```
sudo ln -s /usr/java/jdk1.5.0_07/bin/* .
```

3. Instalando Java no Windows

Passo 1: Utilizando o Windows Explorer, vá até a pasta onde estiver o instalador Java.

Passo 2: Para executar o instalador, duplo-clique no ícone. A caixa de diálogo do instalador J2SE será mostrada com o contrato. Selecione a opção **I accept the terms in the license agreement** caso concorde com os termos apresentados e pressione o botão **Next >**.

Figura 2: Contrato de Licença

Esta próxima janela define o que será instalado. Pressione o botão **Next >** para continuar a instalação. O processo de instalação poderá demorar um pouco, ao término a seguinte janela será mostrada, pressione o botão **Finish** para completar a instalação.

4. Instalando NetBeans no Ubuntu Dapper

Passo 1: Vá para a pasta onde estiver o instalador do NetBeans.

Passo 2: Antes de executar o instalador, assegure-se de que o arquivo seja executável. Para tanto, utilize o botão direito do mouse no ícone do instalador e, em seguida selecione **Properties**. Selecione a aba **Permissions**, e marque a opção **Execute**. Encerre a janela.

Passo 3: Duplo-clique no arquivo de instalação do NetBeans. Pressione o botão **Run in Terminal**. Será mostrada uma caixa de diálogo do NetBeans 5.5. Pressione o botão **Next >**.

Na próxima janela o termos da licença serão mostrados, caso concorde selecione a opção **I accept the terms in the license agreement**, e então pressione o botão **Next >**.

Modifique o nome do diretório para: /usr/java/netbeans-5.5, então pressione o botão **Next >**. Na pasta do JDK, selecione /usr/java/jdk1.5.0_07, e então pressione o botão **Next >**.

A próxima caixa de diálogo mostra apenas informações sobre o NetBeans que você está instalando. Pressione o botão **Next >**. Aguarde o NetBeans terminar o processo de instalação. Pressione o botão **Finish** para completar a instalação.

Passo 4: A fim de possibilitar a execução do NetBeans a partir de qualquer pasta no computador, precisamos criar um caminho de pesquisa. Para isso, entramos na pasta :`/usr/local/bin`. com o comando:

```
cd /usr/local/bin
```

Crie um caminho de pesquisa para o NetBeans, digitando:

```
sudo ln -s /usr/java/netbeans-5.5 .
```

É possível executar o NetBeans a partir de qualquer pasta, digitando:

```
netbeans &
```

5. Instalando NetBeans no Windows

Passo 1: Através do Windows Explorer, localize a pasta do instalador do NetBeans.

Passo 2: Para executar o instalador, dê um duplo-clique no ícone do instalador. O assistente de instalação do NetBeans será mostrado. Pressione o botão **Next >** para iniciar o processo de instalação.

A página do contrato será mostrada. Caso concorde com os termos selecione a opção **I accept the terms in the license agreement** e pressione o botão **Next >** para continuar.

O instalador solicitará a pasta que deve ser instado o NetBeans. Se desejar modifique a pasta sugerida pressionando o botão **Browse**, e ao término pressione o botão **Next >**.

O próximo passo é selecionar uma JDK existente em seu computador. Pressione o botão **Next>** para continuar.

Em seguida, o instalador informará a localização e o espaço em disco que o NetBeans irá ocupar depois de instalado no seu computador. Pressione o botão **Next >** e aguarde o término da instalação. No momento que o NetBeans terminar de ser instalado em seu computador. Pressione o botão

Finish para encerrar o processo.

Hino Nacional

Ouviram do Ipiranga as margens plácidas
De um povo heróico o brado retumbante,
E o sol da liberdade, em raios fúlgidos,
Brilhou no céu da pátria nesse instante.

Se o penhor dessa igualdade
Conseguimos conquistar com braço forte,
Em teu seio, ó liberdade,
Desafia o nosso peito a própria morte!

Ó Pátria amada,
Idolatrada,
Salve! Salve!

Brasil, um sonho intenso, um raio vívido
De amor e de esperança à terra desce,
Se em teu formoso céu, risonho e límpido,
A imagem do Cruzeiro resplandece.

Gigante pela própria natureza,
És belo, és forte, impávido colosso,
E o teu futuro espelha essa grandeza.

Terra adorada,
Entre outras mil,
És tu, Brasil,
Ó Pátria amada!
Dos filhos deste solo és mãe gentil,
Pátria amada,Brasil!

Deitado eternamente em berço esplêndido,
Ao som do mar e à luz do céu profundo,
Fulguras, ó Brasil, florão da América,
Iluminado ao sol do Novo Mundo!

Do que a terra, mais garrida,
Teus risonhos, lindos campos têm mais flores;
"Nossos bosques têm mais vida",
"Nossa vida" no teu seio "mais amores."

Ó Pátria amada,
Idolatrada,
Salve! Salve!

Brasil, de amor eterno seja símbolo
O lábaro que ostentas estrelado,
E diga o verde-louro dessa flâmula
- "Paz no futuro e glória no passado."

Mas, se ergues da justiça a clava forte,
Verás que um filho teu não foge à luta,
Nem teme, quem te adora, a própria morte.

Terra adorada,
Entre outras mil,
És tu, Brasil,
Ó Pátria amada!
Dos filhos deste solo és mãe gentil,
Pátria amada, Brasil!

Hino do Estado do Ceará

Poesia de Thomaz Lopes
Música de Alberto Nepomuceno
Terra do sol, do amor, terra da luz!
Soa o clarim que tua glória conta!
Terra, o teu nome a fama aos céus remonta
Em clarão que seduz!
Nome que brilha esplêndido luzeiro
Nos fulvos braços de ouro do cruzeiro!

Mudem-se em flor as pedras dos caminhos!
Chuvas de prata rolem das estrelas...
E despertando, deslumbrada, ao vê-las
Ressoa a voz dos ninhos...
Há de florar nas rosas e nos cravos
Rubros o sangue ardente dos escravos.
Seja teu verbo a voz do coração,
Verbo de paz e amor do Sul ao Norte!
Ruja teu peito em luta contra a morte,
Acordando a amplidão.
Peito que deu alívio a quem sofria
E foi o sol iluminando o dia!

Tua jangada afoita enfune o pano!
Vento feliz conduza a vela ousada!
Que importa que no seu barco seja um nada
Na vastidão do oceano,
Se à proa vão heróis e marinheiros
E vão no peito corações guerreiros?

Se, nós te amamos, em aventuras e mágoas!
Porque esse chão que embebe a água dos rios
Há de florar em meses, nos estios
E bosques, pelas águas!
Selvas e rios, serras e florestas
Brotem no solo em rumorosas festas!
Abra-se ao vento o teu pendão natal
Sobre as revoltas águas dos teus mares!
E desfraldado diga aos céus e aos mares
A vitória imortal!
Que foi de sangue, em guerras leais e francas,
E foi na paz da cor das hóstias brancas!

GOVERNO DO
ESTADO DO CEARÁ

Secretaria da Educação