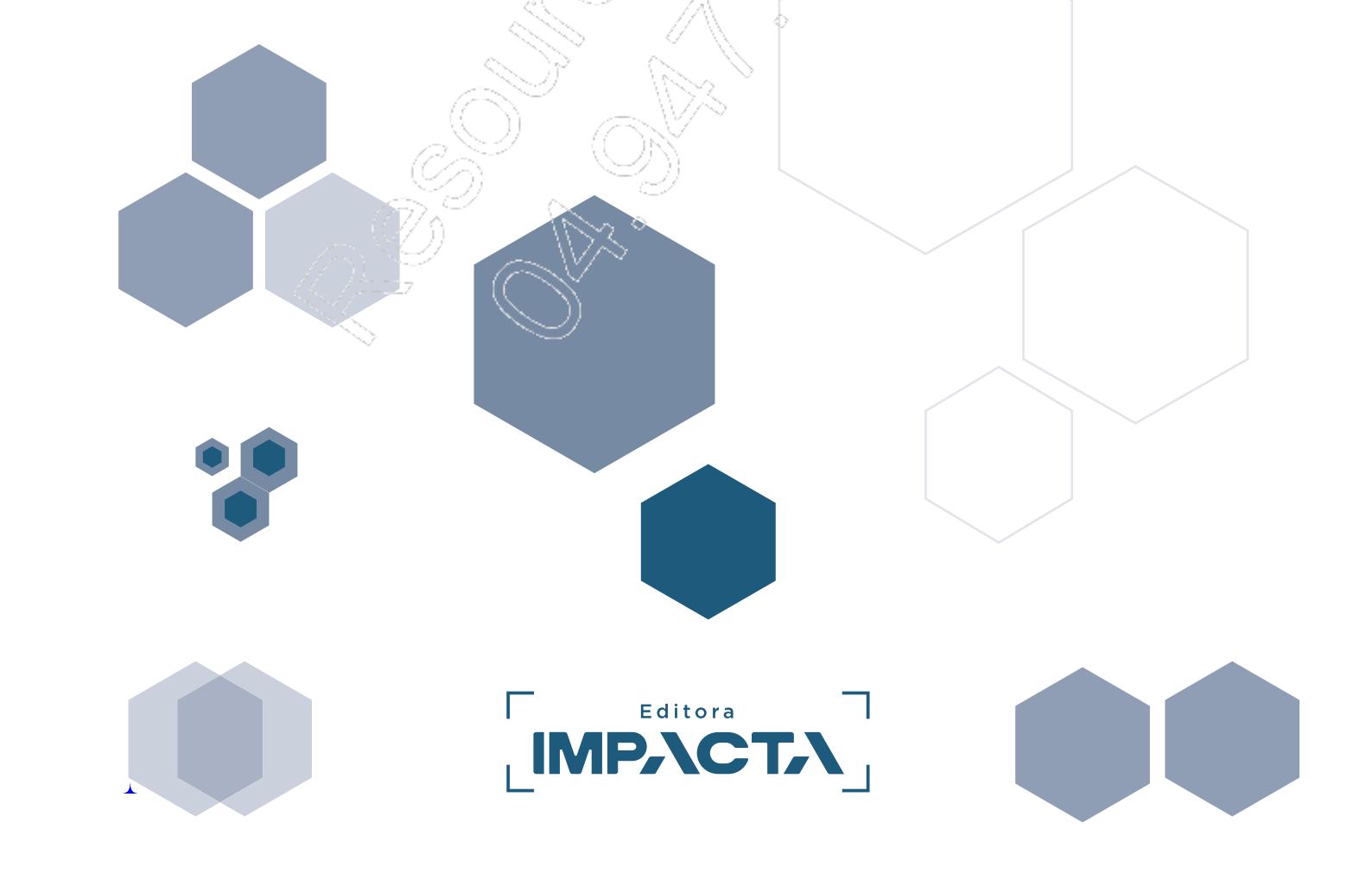


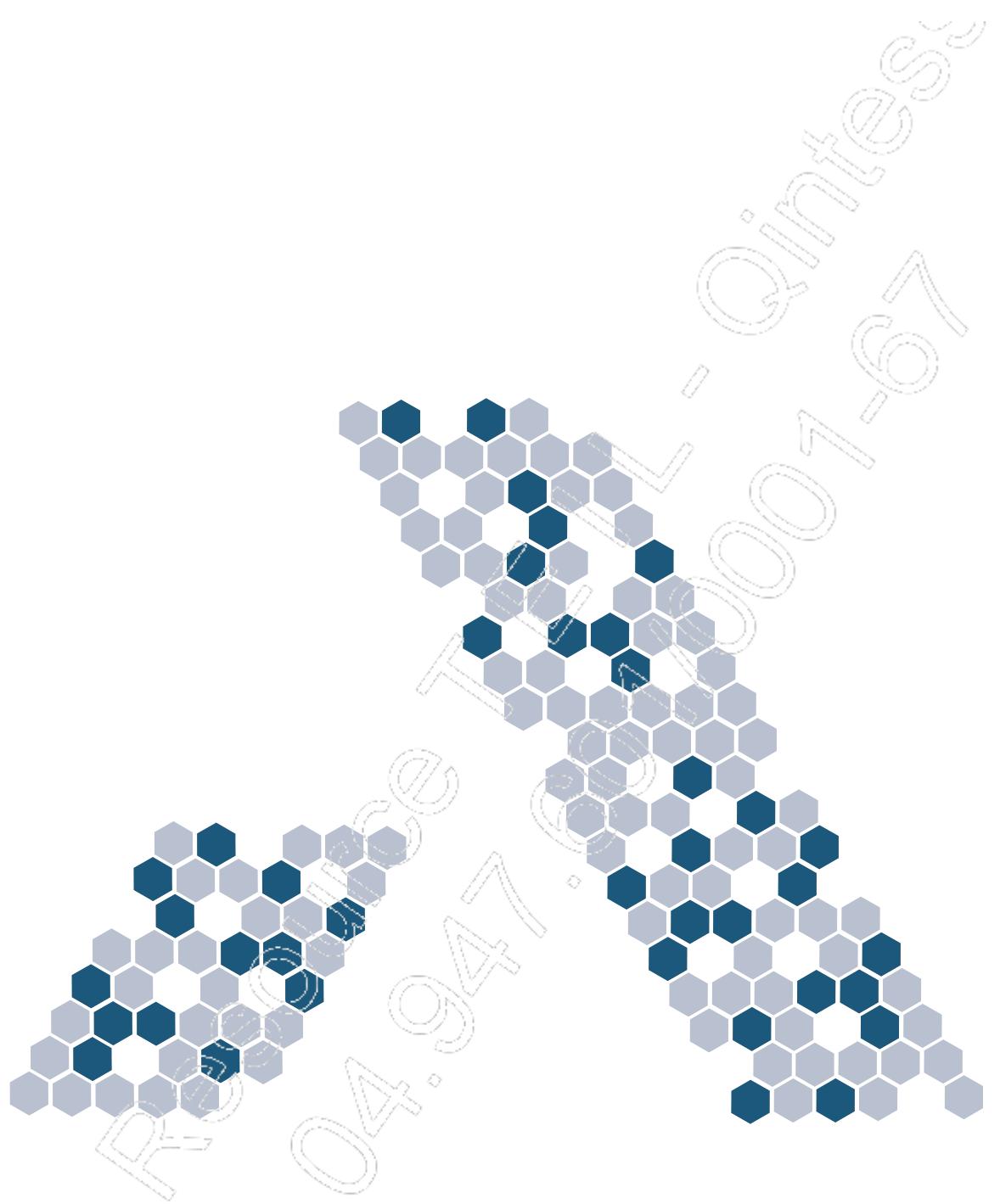


Java Programmer

Parte I



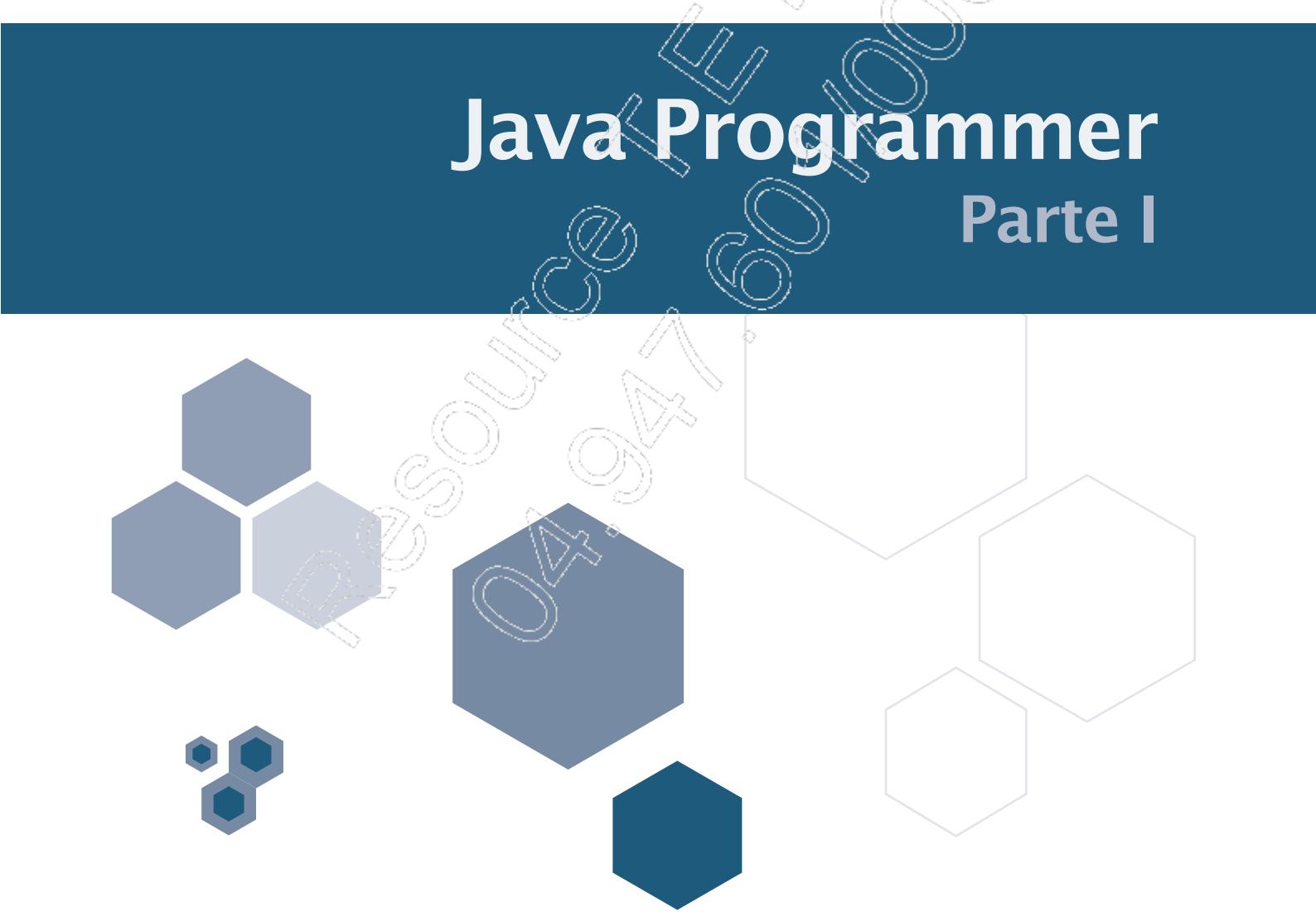
Editora
IMPACTA





Java Programmer

Parte I



Editora
IMPACTA



Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Java Programmer

Parte I

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Autoria

Braulio Consani Moura

Revisão Ortográfica e Gramatical

Marcos Cesar dos Santos Silva

Diagramação

Paloma da Silva Teixeira

Edição nº 1 | 1841_0

Março/ 2018

Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Set/2014: Java Programmer.

Autoria: Sandro Luiz de Souza Vieira

Sumário

Capítulo 1 - Introdução à linguagem Java.....	9
1.1. Histórico	10
1.2. Características	11
1.3. Edições disponíveis	13
1.4. Java Development Kit (JDK).....	13
1.4.1. Java Virtual Machine (JVM).....	14
1.5. Ambientes de desenvolvimento (IDEs)	14
1.6. Estrutura básica de um programa Java	15
1.7. Características do código	16
1.7.1. Case sensitive	16
1.7.2. Nomes de arquivo	16
1.7.3. Nomenclatura	16
1.7.4. Estrutura.....	16
1.7.5. Comentários	17
1.7.6. Palavras reservadas	17
1.8. Compilando e executando um programa.....	18
1.9. JShell (Java Interativo).....	23
1.9.1. Utilização básica	24
Pontos principais	26
Teste seus conhecimentos.....	27
Capítulo 2 - Tipos de dados, literais e variáveis	29
2.1. Introdução	30
2.2. Tipos de dados primitivos	30
2.3. Literais.....	32
2.3.1. Literais inteiros.....	32
2.3.2. Literais de ponto flutuante.....	34
2.3.3. Literais booleanos	34
2.3.4. Literais de caracteres	34
2.3.4.1. Caracteres de escape	35
2.4. Variáveis.....	36
2.4.1. Definindo uma variável	37
2.4.2. Declarando uma variável	37
2.4.2.1. Usando o qualificador final	39
2.4.3. Escopo de variáveis	39
2.5. Casting	40
2.6. Tipos de referência comuns	41
2.6.1. String.....	41
2.6.2. Enum	42
2.6.3. Classes Wrapper	43
Pontos principais	44
Teste seus conhecimentos.....	45

Java Programmer - Parte I

Capítulo 3 - Operadores	47
3.1. Introdução	48
3.2. Operador de atribuição	48
3.3. Operadores aritméticos	50
3.3.1. Operadores aritméticos de atribuição reduzida	51
3.4. Operadores incrementais e decrementais	52
3.5. Operadores relacionais	53
3.6. Operadores lógicos	54
3.7. Operador ternário	57
3.8. Precedência dos operadores	58
Pontos principais	60
 Teste seus conhecimentos.....	61
 Mãos à obra!.....	63
Capítulo 4 - Estruturas de controle.....	65
4.1. Introdução	66
4.2. Estruturas de desvios condicionais	66
4.2.1. if / else	67
4.2.2. switch	69
4.3. Estruturas de repetição	73
4.3.1. While	73
4.3.2. Do / while	75
4.3.3. For	76
4.4. Outros comandos	80
4.4.1. Break	80
4.4.1.1. Instruções rotuladas	81
4.4.2. Continue	83
Pontos principais	84
 Teste seus conhecimentos.....	85
 Mãos à obra!.....	87
Capítulo 5 - Introdução à orientação a objetos.....	91
5.1. Apresentação	92
5.2. Classes	92
5.3. Objeto	93
5.3.1. Instanciação	94
5.4. Atributos	94
5.5. Tipos construídos	95
5.5.1. Atribuição entre objetos de tipos construídos	96
5.5.2. Variáveis não inicializadas	99
5.5.3. O uso do this	100
5.6. Encapsulamento	101
5.7. Pacotes	102
5.7.1. Criando um pacote	103
5.7.2. Acessando uma classe em outro pacote	105
5.8. UML – Diagramas de casos de uso, classes e pacotes	106
5.8.1. Diagrama de casos de uso	107
5.8.2. Diagrama de classes	108
5.8.3. Diagrama de pacotes	109
Pontos principais	110
 Teste seus conhecimentos.....	111
 Mãos à obra!.....	115
Projeto Prático - Fase 1	119

Sumário

Capítulo 6 - Métodos.....	121
6.1. Introdução	122
6.2. Estrutura de um método	123
6.3. Comando return.....	124
6.4. Chamando um método (mensagens)	124
6.5. Passagem de parâmetros	127
6.6. Varargs	129
6.7. Métodos assessores	131
6.7.1. Método getter	132
6.7.2. Método setter	132
6.8. Modificadores de métodos	133
6.9. Modificador static	134
6.9.1. Atributos estáticos	134
6.9.2. Métodos estáticos	135
6.9.3. Exemplos práticos de membros estáticos.....	136
6.10. Método main().....	138
6.11. Sobrecarga de métodos	140
6.12. UML – Diagrama de sequência.....	142
Pontos principais	143
Teste seus conhecimentos.....	145
Mãos à obra!	149
Capítulo 7 - Construtores	155
7.1. Introdução	156
7.2. Construtor padrão	156
7.3. Considerações sobre os construtores.....	159
Pontos principais	160
Teste seus conhecimentos.....	161
Mãos à obra!	163
Projeto Prático - Fase 2	167
Capítulo 8 - Arrays.....	169
8.1. Introdução	170
8.2. Tipos de array	170
8.2.1. Array unidimensional	170
8.2.2. Array bidimensional	172
8.2.3. Array multidimensional	173
8.3. Acessando elementos de um array	174
8.3.1. Acesso aos elementos em um for tradicional.....	176
8.3.2. Acesso aos elementos usando enhanced for (foreach)	177
8.4. Modos de inicializar e construir um array	178
8.4.1. Por meio de uma única instrução	178
8.4.2. Por meio de um array anônimo	181
8.5. Passando um array como parâmetro	182
8.5.1. Variáveis de referência para arrays unidimensionais	183
8.5.2. Variáveis de referência para arrays multidimensionais	184
8.6. Array de argumentos	184
Pontos principais	189
Teste seus conhecimentos.....	191
Mãos à obra!	193



Java Programmer - Parte I

Capítulo 9 - Herança, classes abstratas e polimorfismo	197
9.1. Introdução	198
9.2. Herança e generalização	200
9.3. Estabelecendo herança entre classes.....	201
9.3.1. Acesso aos membros da superclasse.....	203
9.3.2. O operador super.....	205
9.3.3. Chamada ao construtor da superclasse	207
9.4. Herança e classes.....	209
9.4.1. Classes finais	209
9.4.2. Classe Object	211
9.5. Classes abstratas	212
9.5.1. Métodos abstratos	212
9.6. Polimorfismo	214
9.6.1. Ligaçāo tardia (late binding).....	215
9.6.2. Polimorfismo em métodos declarados na superclasse	216
9.6.3. Operador instanceof	218
9.7. UML - Associações entre classes	219
9.7.1. Tipos de associação	220
9.7.1.1. Associação Simples	220
9.7.1.2. Agregação	221
9.7.1.3. Composição	221
9.7.1.4. Herança	223
9.7.2. Herança x Composição	224
Pontos principais	226
Teste seus conhecimentos.....	227
Mãos à obra!.....	231
Capítulo 10 - Interfaces	237
10.1. O conceito de interface	238
10.1.1. Variáveis de referência	239
10.1.2. Constantes	240
10.2. Métodos em interfaces	241
10.2.1. Métodos estáticos	241
10.2.2. Métodos default	242
10.2.3. Métodos privados	242
Pontos principais	243
Teste seus conhecimentos.....	245
Mãos à obra!.....	247

1

Introdução à linguagem Java

- ◆ Histórico;
- ◆ Características;
- ◆ Edições;
- ◆ Java Development Kit (JDK);
- ◆ Ambientes de desenvolvimento (IDEs);
- ◆ Estrutura básica de um programa Java;
- ◆ Características do código;
- ◆ Compilando e executando um programa;
- ◆ JShell (Java Interativo).

1.1. Histórico

Java é uma linguagem de programação orientada a objetos, desenvolvida em 1991 por um grupo de profissionais da empresa Sun Microsystems, em um projeto chamado Green. O objetivo inicial do projeto era desenvolver uma linguagem para dispositivos portáteis inteligentes. A linguagem foi lançada oficialmente em 1995, não só como uma linguagem, mas como uma plataforma de desenvolvimento. Assim, Java começou a ser largamente utilizada na criação de páginas Web e se disseminou até ser uma das linguagens mais utilizadas no mundo.

Os profissionais que desenvolveram a linguagem utilizaram grande parte da estrutura da C++, por também ser uma linguagem orientada a objetos, e conceitos de segurança da linguagem SmallTalk. Partindo daí, os criadores de Java obtiveram: uma linguagem de fácil aprendizado, por ser relativamente simples e com poucas construções; uma linguagem um tanto familiar aos programadores, pela semelhança com C e C++; e uma linguagem de grande eficácia, que permite a criação maciça de aplicações, applets e sistemas embutidos.

Desde seu lançamento, Java teve diferentes versões. Veja um breve quadro dessas versões e suas principais características:

Versão	Características
1.02	Lançada em 1996, a primeira versão estável de Java tinha como destaque os applets, programas que não podem ser executados sozinhos, mas só no contexto de outra aplicação. Possuía 250 classes.
1.1	Lançada em 1997, possuía 500 classes.
1.2	Lançada em 1998, possuía 2300 classes e passou a ser chamada Java 2. Era disponibilizada em duas edições: Standard Edition (J2SE), uma edição básica, e Enterprise Edition (J2SEE), mais robusta, voltada para o desenvolvimento de aplicações empresariais.
1.3	Lançada em 2000, introduziu a tecnologia HotSpot no modo Java Virtual Machine.
1.4	Lançada em 2002, introduziu a diretiva assert .
5	Versão de 2004, que passou a ser chamada de Java 5, inseriu recursos importantes na API, como Generics , Autoboxing , Enum , Var-args , entre outros.
6	Lançada em 2006, não difere muito da versão anterior, tendo introduzido suporte a scriptings.

Versão	Características
7	Lançada em 2011, o Java 7 passou a adotar o OpenJDK como padrão, uma versão livre e open source da máquina virtual Java, o que permite colaboração no desenvolvimento da linguagem. As principais melhorias nesta versão estão relacionadas ao desempenho, estabilidade, segurança, ao Java Plug-in e à Java Virtual Machine.
8	Lançada em 2014, foi introduzida na linguagem a programação funcional através das expressões Lambda , uma de suas maiores inovações, além de uma nova API para manipulação de data e hora e extensões importantes nas APIs de I/O e coleções.
9	Versão mais atual da linguagem, o Java 9 trouxe modificações estruturais importantes para a plataforma como um todo e seu ambiente de execução. Destaque para o novo esquema de modularização (JIGSAW), e o console de programação interativa (o JShell), além da consolidação do paradigma funcional nas bibliotecas da linguagem.

Java 9 é a versão que tomaremos como base para este treinamento.

1.2. Características

Conheça as principais características da linguagem java, que fazem dela uma ferramenta importante para o desenvolvimento de aplicações:

- **Linguagem orientada a objetos**

Uma linguagem orientada a objetos permite a criação de programas constituídos por objetos que se comunicam entre si e são manipulados por meio de propriedades e métodos. É um paradigma de programação muito bem consolidado.

- **Multiplataforma**

Java é uma linguagem interpretada e compilada. Isso significa que, para um programa ser executado, ele deve ser compilado, isto é, transformado em bytecode (um arquivo binário que contém o código do programa) e, posteriormente, interpretado por um interpretador Java. Este interpretador se encontra na máquina virtual Java (ou JVM, Java Virtual Machine), que é responsável por executar o bytecode.

Como cada execução do programa depende da JVM, o bytecode é independente de plataforma e a execução pode ser feita em qualquer plataforma que suporte a JVM. Essa característica é fundamental para as aplicações que são distribuídas por meio de redes heterogêneas, especialmente a Internet.

- **Robustez e confiabilidade**

A programação em Java confere um alto grau de robustez e confiabilidade às aplicações. Isso se deve a alguns fatores: a ausência de ponteiros (característicos do C++), evitando falhas comuns a esse tipo de recurso; checagens realizadas em tempo de execução; a declaração explícita dos tipos das variáveis; a tipagem forte, que permite verificar, em tempo de compilação, eventuais problemas na combinação de tipos; o tratamento de exceções, que simplifica a manipulação e recuperação de erros; e o gerenciamento automático de memória, por meio do garbage collector, que faz a remoção de objetos não utilizados da memória, disponibilizando espaço para outras operações.

- **Segurança**

A segurança é uma característica fundamental de Java, que possui diversas camadas de controle como proteção contra possíveis códigos maliciosos. Essas camadas são as seguintes: a impossibilidade de acesso direto à memória, já que os programas não podem manipular ponteiros nem acessar memória que esteja além de arrays e strings; a assinatura digital, que é anexada ao código Java e permite protegê-lo contra falsificações e definir certos geradores de código (pessoas ou instituições) como confiáveis ou não; e a verificação de bytecodes carregados, que evita a implementação de códigos corrompidos.

- **Multithread**

Multithread consiste na possibilidade de um programa executar simultaneamente diversas threads (linhas de execução). Isso permite, portanto, a realização simultânea de diversas tarefas distribuídas para todos os processadores disponíveis na máquina.

1.3. Edições disponíveis

A linguagem Java pode ser aplicada a diversos ambientes. Por conta disso, possui diferentes divisões, ou edições. As principais são as seguintes:

- **JSE (Java Standard Edition)**: A edição básica e principal da linguagem. Utilizada primordialmente em estações de trabalho. Possibilita o desenvolvimento de aplicações cliente-servidor;
- **JEE (Java Enterprise Edition)**: Edição destinada ao desenvolvimento de aplicações complexas multicamadas. Possibilita a integração entre sistemas de diferentes plataformas e o desenvolvimento de aplicações distribuídas;
- **JME (Java Micro Edition)**: Edição para dispositivos com menor poder de processamento e memória, sendo de aplicação primordial para dispositivos usados em IoT (Internet of Things).

1.4. Java Development Kit (JDK)

Para desenvolver aplicativos em linguagem Java, você precisa do conjunto de ferramentas JDK (Java Development Kit) instalado na sua máquina. O pacote JDK contém o compilador, a JVM, algumas bibliotecas, ferramentas como **jar**, **javadoc** e códigos utilizados como exemplos.

O JDK contém também o JRE (Java Runtime Environment), um ambiente de execução que simula uma máquina responsável por realizar as interpretações Java.

Como não é possível realizar compilação de código Java por meio dessa máquina JRE, ela é indicada somente àqueles que necessitam executar aplicações Java e contam com um pacote que já contenha a máquina virtual e as bibliotecas necessárias para essa execução.

1.4.1. Java Virtual Machine (JVM)

JVM (Java Virtual Machine) refere-se a uma máquina virtual que cada uma das máquinas deve possuir para executar classes, depois de realizado o processo de compilação. Cada sistema operacional possui sua respectiva JVM (há JVM para Solaris, para Linux e para Windows).

1.5. Ambientes de desenvolvimento (IDEs)

A linguagem Java já possui as ferramentas e recursos necessários para o desenvolvimento de programas. Mas conta também com IDEs (Integrated Development Environments), que são ambientes gráficos de desenvolvimento que podem auxiliar na criação de sistemas de maneira mais produtiva. A seguir, relacionamos alguns dos principais ambientes de desenvolvimento de Java:

- **NetBeans IDE**

Este ambiente permite escrever, compilar, depurar e instalar programas. Por ser escrito em Java, pode ser executado em diversas plataformas (Windows, Linux, Solaris, Apple Mac OS, entre outras). É uma ferramenta de código aberto, ou seja, com as fontes do programa disponíveis para uso. Pode ser baixado em <http://www.netbeans.org>.

- **JCreator**

Também permite escrita, compilação e depuração de programas. É uma ferramenta leve e fácil de usar, mas disponibilizada apenas para Windows. Possui uma versão livre, JCreator LE, e uma comercial, JCreator Pro. Disponível em <http://www.jcreator.com/>.

- **JBuilder**

Ferramenta bastante completa para o desenvolvimento Java. É disponibilizada sob licença comercial, mas possui uma versão para testes por tempo limitado. Hoje faz parte da suíte Embarcadero e pode ser baixada em <http://www.embarcadero.com/>.

- **Eclipse**

Desenvolvido pela IBM em 2001 e disponibilizado para a comunidade Java, possui código aberto e permite escrever, compilar e depurar programas. Pode ser executado em Windows, Linux e Solaris e está disponível em <http://www.eclipse.org>.

Este será o IDE utilizado para desenvolver os exemplos e laboratórios neste treinamento.

- **IntelliJ IDEA**

IDE com ampla gama de fãs e adeptos, é desenvolvido pela empresa JetBrains. Sua licença é comercial, porém possui uma versão Community gratuita, para projetos Open Source. Dotada de inúmeros recursos e plugins, é largamente utilizada por empresas dos mais diversos gêneros. Pode ser baixada em <http://www.jetbrains.com/idea>.

1.6. Estrutura básica de um programa Java

A estrutura básica de um programa executável Java é exibida adiante e é importante que você a compreenda bem, pois será usada ao longo de todo o treinamento e também na criação de todos os programas que você fará:

```
public class <nome> {  
    public static void main(String args[]) {  
        // declarações;  
        // comandos;  
    }  
}
```

Veja a descrição dessa estrutura básica:

- Na primeira linha, temos a definição do nome da classe e, em seguida, a primeira chave, que especifica o corpo (início do bloco) da classe. Tudo o que fizer referência à classe deve estar entre a primeira e a última chave (última linha do bloco);
- Na terceira linha, temos a definição do método **main()** (as instruções **public**, **static**, **void**, **String**, **args[]** serão explicadas posteriormente) e encontramos a abertura da chave (bloco) referente aos comandos dentro do método **main()**;
- Na quinta linha, encontram-se as declarações de variáveis dentro do método **main()**;
- Na sexta linha, encontram-se os comandos dentro do método **main()**;
- Na sétima linha, a chave (bloco) do método **main()** é fechada;
- Na oitava linha, fecha-se a chave (bloco) da classe.

1.7. Características do código

É essencial que você se familiarize com algumas particularidades ao escrever códigos em Java. A seguir, descrevemos as principais.

1.7.1. Case sensitive

Java é uma linguagem **case sensitive**, ou seja, uma linguagem capaz de distinguir letras em caixa alta (maiúsculas) e em caixa baixa (minúsculas).

1.7.2. Nomes de arquivo

O nome de um arquivo feito em Java tem a extensão **.java** e, preferencialmente, deve ser o mesmo nome de uma das classes declaradas dentro dele. Assim, para a class **Cliente**, por exemplo, teremos o arquivo com o nome **Cliente.java**. Seria um erro definir a classe como **class Cliente** e o nome do arquivo como **cliente.java**.

1.7.3. Nomenclatura

Os nomes atribuídos aos identificadores devem ser **iniciados** pelos caracteres underline (_), cifrão (\$), ou por letras. Para os demais caracteres, pode-se incluir, na lista anterior, os números. Dentre os identificadores, temos as classes, os métodos e as variáveis, entre outros.

! Não é permitido definir um identificador com um único caractere underline (_).

1.7.4. Estrutura

O início de um bloco de código ou de uma classe é sempre marcado pela abertura de chaves ({}, e o término, pelo fechamento de chaves (}). Os comandos, geralmente, terminam com sinal de ponto e vírgula (;).

1.7.5. Comentários

As linhas de comentário podem ser iniciadas de formas distintas, dependendo dos seguintes fatores:

- **Comentário de uma linha**

Comentários que não ultrapassam uma linha devem ser iniciados por duas barras (//).

- **Comentário de mais de uma linha**

Comentários que ultrapassam uma linha ou, ainda, blocos que não devem ser executados são marcados por barra e asterisco no início /*) e asterisco e barra no final (*/).

- **Comentários para o Javadoc**

O Java possui uma ferramenta, chamada **Javadoc**, para criar documentação, em HTML, sobre as funcionalidades dos pacotes, classes, métodos e atributos. Os comentários em Javadoc iniciam com uma barra e dois asteriscos (/**) e terminam com um asterisco e uma barra (*/).

1.7.6. Palavras reservadas

Em Java, há palavras especiais, reservadas para uma funcionalidade específica no compilador. Elas não podem ser utilizadas como nome de classes, métodos ou variáveis.

As palavras reservadas são escritas em caixa baixa.

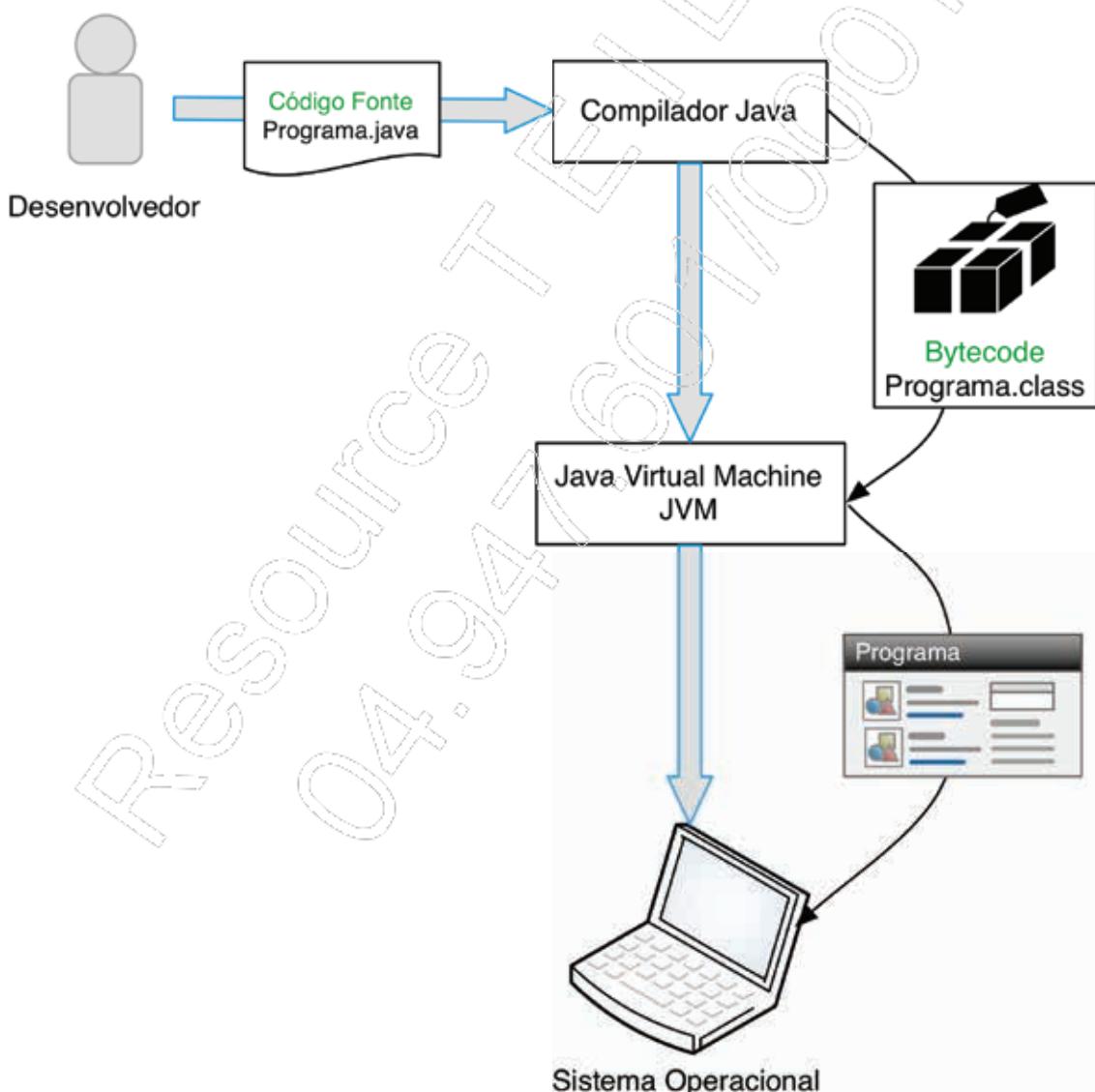
A seguir, as palavras reservadas em Java 8:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	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

- As palavras reservadas **const** e **goto** não são atualmente usadas.
- As palavras **true** e **false** são, na verdade, literais booleanos.
- A palavra **null** também é um literal.
- O caractere **underline** (_) também é uma palavra reservada.

1.8. Compilando e executando um programa

O processo necessário para se executar um programa em Java segue duas etapas bem definidas: a compilação e a interpretação. Veja, em um diagrama, como funciona cada estágio do processo:

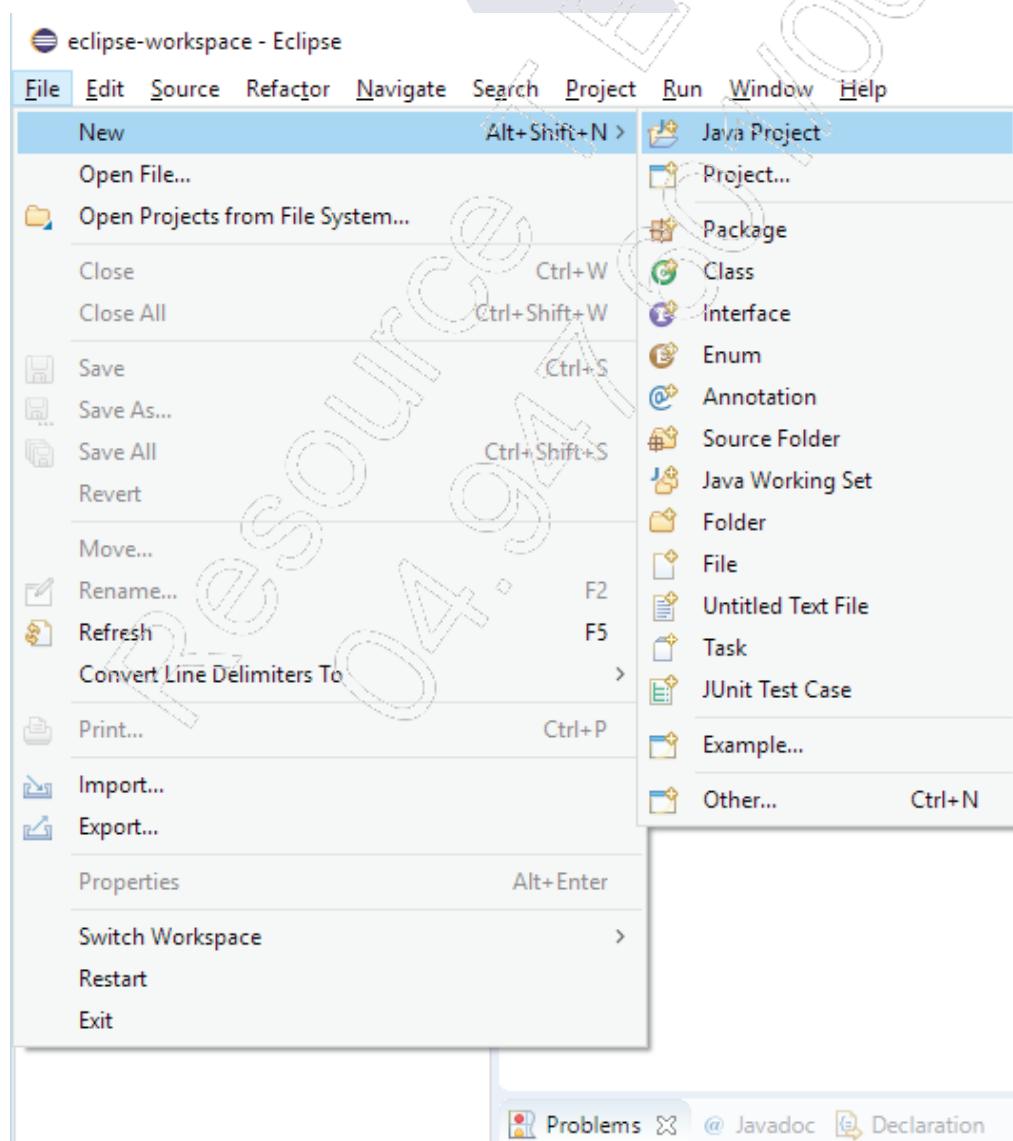


Note que, ao passar pelo compilador Java, também conhecido como “javac”, o código-fonte existente no arquivo **Programa.java** é inteiramente analisado e, caso não haja erros, o compilador gera um arquivo de mesmo nome, porém com a extensão **.class**: **Programa.class**. Esse arquivo contém instruções conhecidas como “bytecode”, que representam símbolos intermediários entre o código-fonte e a linguagem de máquina. Caso existam erros, o compilador gerará instruções para o desenvolvedor, indicando qual o erro e a sua localização.

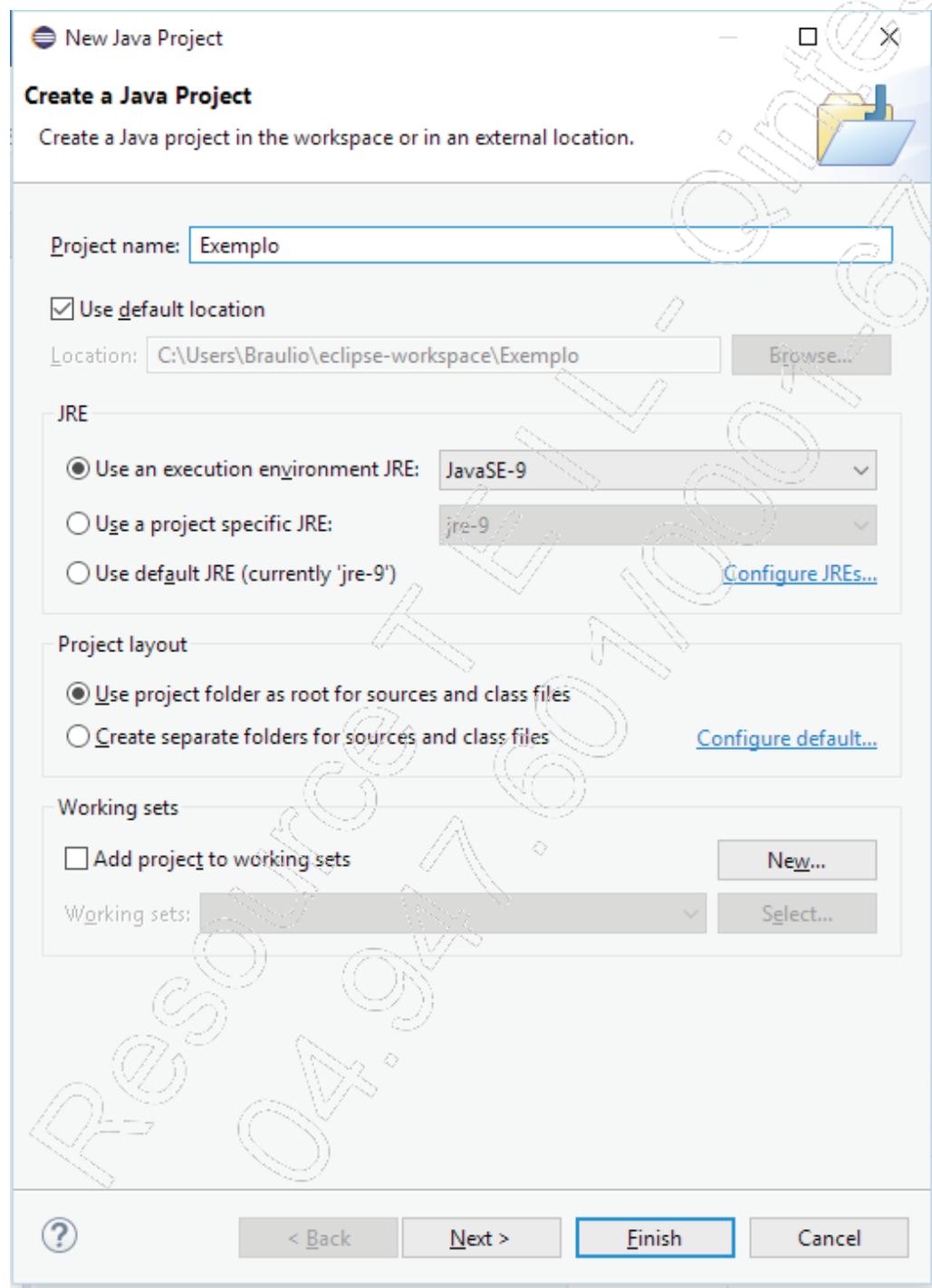
Após o processo de compilação, o bytecode gerado é executado pela JVM, que o transforma em binário executável para o Sistema Operacional de base. Nessa etapa, acontece a interpretação do código, feita linha a linha e, na incidência de erros, a JVM lança as chamadas exceções, que serão abordadas posteriormente.

Para compreender o processo de compilação e execução de uma classe (programa) em Java, vale considerar os seguintes procedimentos no Eclipse:

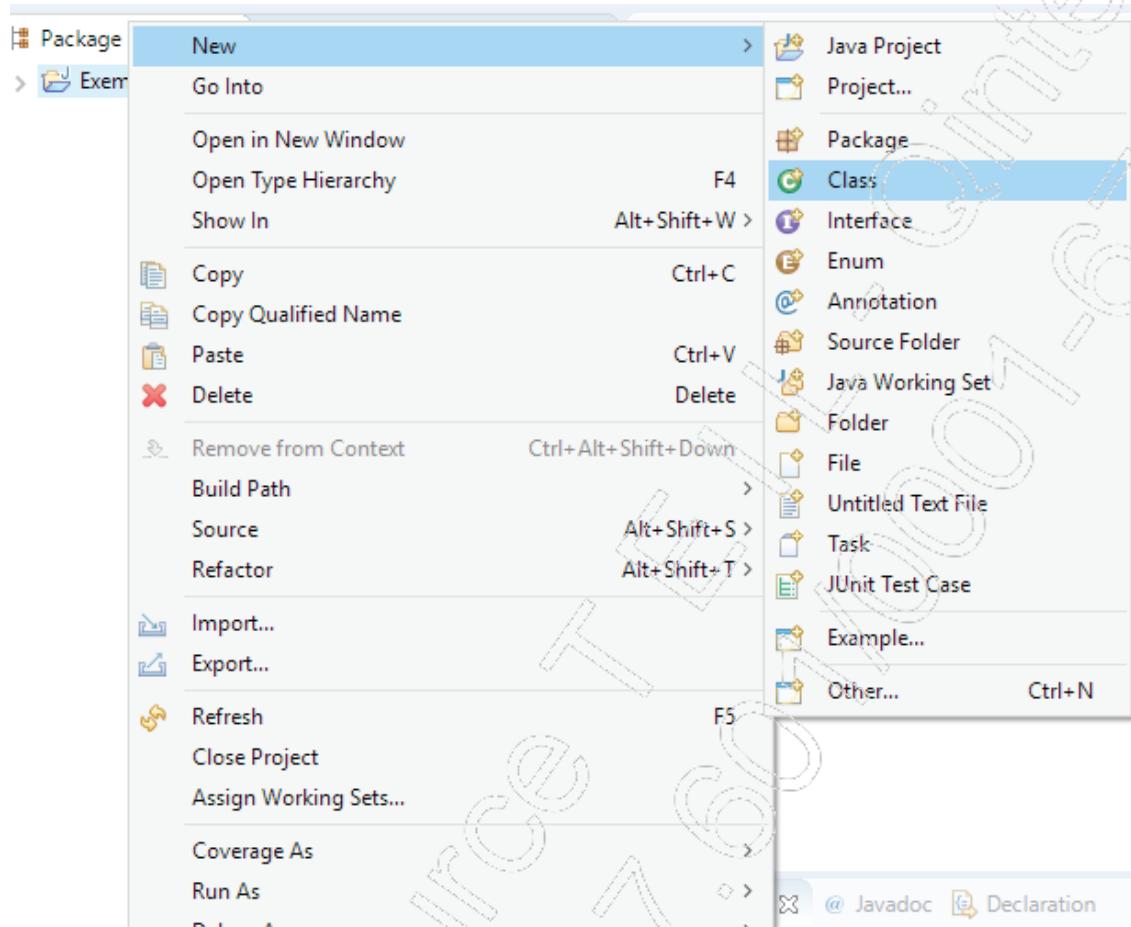
1. No menu **File**, selecione **New / Java Project**:



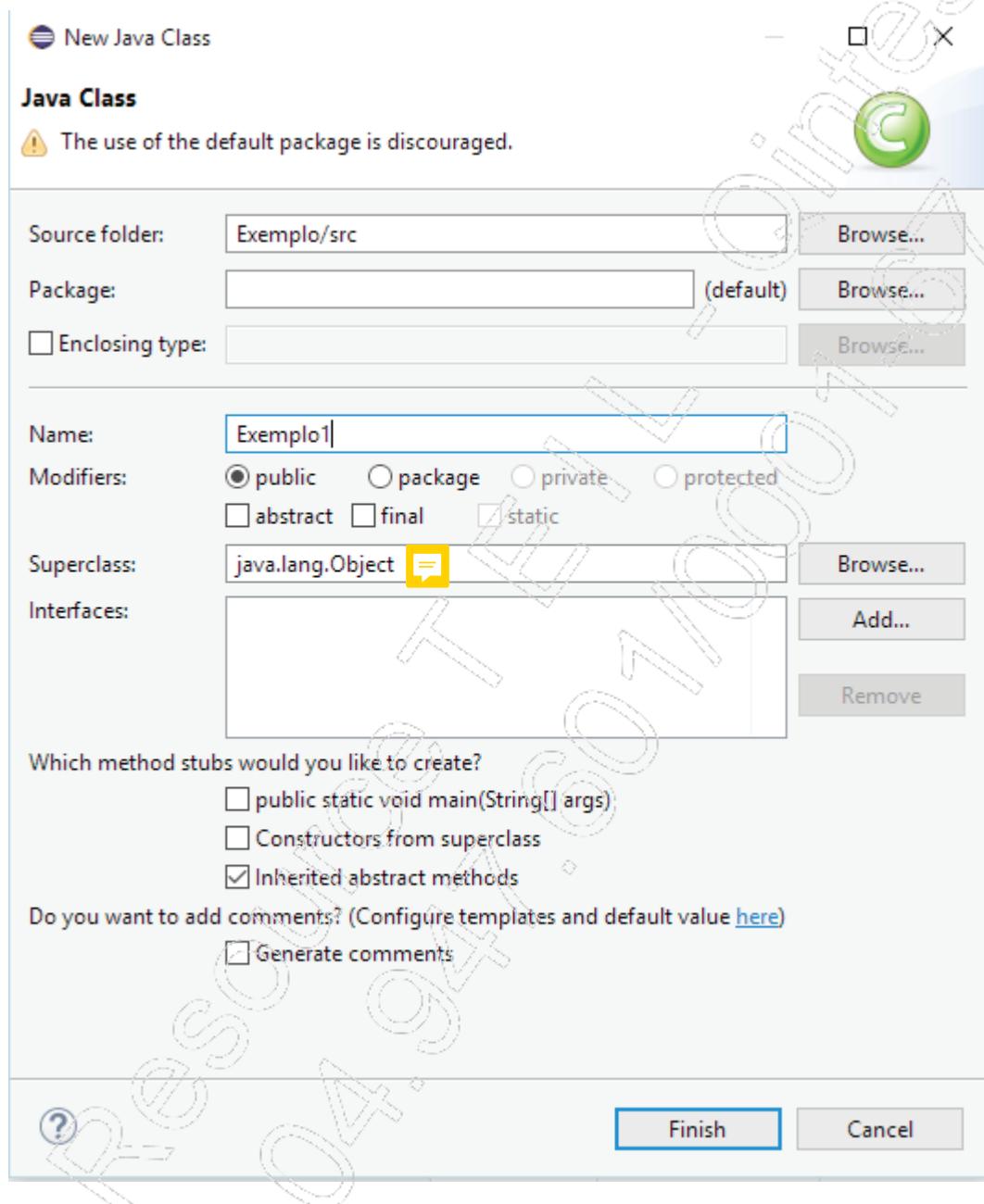
2. Na janela **New Java Project**, nomeie o projeto e clique em **Finish**:



3. Clique com o botão direito do mouse no projeto criado e escolha New Class:



4. Na janela New Java Class, adicione o nome desejado e clique em **Finish**:



A classe será criada e aparecerá da seguinte forma no editor de código:

```
1 public class Exemplo1 {  
2  
3 }  
4
```

5. Digite o seguinte código no editor:

```
Exemplo1.java X
1 public class Exemplo1 {
2
3     public static void main(String[] args) {
4         System.out.println("Meu primeiro programa Java!");
5     }
6
7 }
8
```

6. Pressione **CTRL + S** para salvar o código digitado;

7. Para compilar e executar o código, clique no botão **Run**, na seção **Launch** da barra de ferramentas padrão do Eclipse:



Caso não haja erros após o processo de compilação, o resultado será exibido na janela **Console** do Eclipse:

A screenshot of the Eclipse IDE 'Console' view. The title bar shows tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console window displays the text: '<terminated> Exemplo1 [Java Application] C:\Program Files\Java\jre-9\bin\javaw.exe | Meu primeiro programa Java!'

1.9. JShell (Java Interativo)

JShell é um programa auxiliar do JDK que faz avaliações de declarações, comandos e expressões à medida que são inseridas na linha de comando.

Usando JShell, pode-se entrar com elementos do programa verificando o resultado imediatamente, realizando, se necessário, os ajustes. JShell é uma ferramenta interativa para aprendizado e prototipagem rápida de códigos Java.

Como será visto no decorrer dos estudos da programação Java, para escrever um trecho de código e executá-lo, são necessários os seguintes passos:

1. Escrever um programa completo;
2. Compilar e corrigir todos os erros;

3. Executar o programa;
4. Verificar possíveis pontos de ajustes;
5. Editar o programa para ajustar eventuais pontos do código;
6. Repetir este processo.

JShell ajuda a testar variações de código à medida que se desenvolve o programa. Permite testar trechos de instruções, variações de métodos, dentre outras opções. JShell não substitui a IDE. A forma como, na prática, é utilizado é passando pequenos trechos de código da IDE para o JShell, testando variações e implementações diferentes, verificando o resultado dinamicamente e, assim que se chega à versão considerada ideal, integrando o código do JShell ao código presente na IDE.

1.9.1. Utilização básica

JShell foi incluído no JDK a partir da versão 9. Para iniciar esta ferramenta, deve-se abrir o prompt de comando. Se, na variável de ambiente do sistema operacional PATH, o caminho `[java-home]/jdk-9/bin` não estiver presente, é necessário iniciar a ferramenta a partir desse diretório.

Vejamos, adiante, alguns exemplos da utilização básica da ferramenta:

- Iniciando o JShell:

```
% jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
jshell>
```

- Saindo do JShell:

```
jshell> /exit
| Goodbye
```

- Iniciando o JShell no modo **verbose**, para obter mais informações de feedback dos comandos:

```
% jshell -v
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

- Declarando uma variável e atribuindo a ela um valor (é retornado um feedback do que foi processado neste comando):

```
jshell> int x = 45  
x ==> 45  
| created variable x : int
```

Expressões criadas e que não tenham variáveis declaradas fazem com que o JShell crie variáveis anônimas. Métodos podem ser criados e executados dinamicamente (métodos, variáveis e declarações serão detalhados em capítulos posteriores).

```
jshell> 2 + 2  
$3 ==> 4  
| created scratch variable $3 : int  
  
jshell> String twice(String s) {  
...>     return s + s;  
...> }  
| created method twice(String)  
  
jshell> twice("Java")  
$5 ==> "JavaJava"  
| created scratch variable $5 : String
```

! Esta foi apenas uma visão introdutória da ferramenta JShell. Para explorar todas as possibilidades de utilização disponíveis, acesse a página oficial: <https://docs.oracle.com/javase/9/jshell/>.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

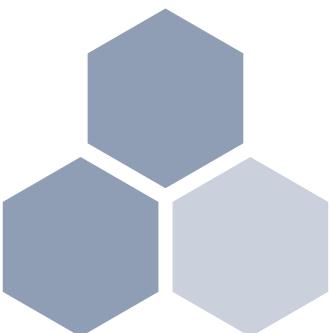
- Java é uma linguagem de programação orientada a objetos desenvolvida em 1991 pela Sun Microsystems. Entre suas principais características, destacamos a orientação a objetos, a possibilidade de ser executada em diversas plataformas, a robustez e a confiabilidade, a segurança e o fato de ser multithread;
- Para desenvolver aplicativos em linguagem Java, é preciso ter o conjunto de ferramentas JDK (Java Development Kit) instalado na máquina. Ele contém o compilador, a JVM, algumas bibliotecas, ferramentas como **jar**, **javadoc** e códigos utilizados como exemplos;
- Os principais ambientes de desenvolvimento para Java são: **NetBeans IDE**, **JCreator**, **JBuilder**, **Eclipse** e **IntelliJ IDEA**. O Eclipse será usado nos exemplos e laboratórios deste treinamento;
- Todo programa executável em Java possui uma estrutura básica, que consiste na definição do nome da classe, no corpo (bloco) da classe, na definição do método **main** e seus comandos e variáveis;
- A execução de todo programa Java passa por duas etapas bem definidas: a compilação e a interpretação. Na primeira, o compilador gera o código bytecode, que, em seguida, é interpretado pela JVM e executado no Sistema Operacional de base da máquina.



1

Introdução à linguagem Java

Teste seus conhecimentos



1. Quais são as principais características de Java?

- a) Linguagem orientada a objeto, multiplataforma, robustez e confiabilidade, segurança e monothread.
- b) Linguagem orientada a objeto, monoplataforma, robustez e confiabilidade, segurança e multithread.
- c) Linguagem orientada a objeto, multiplataforma, robustez e confiabilidade, segurança e multithread.
- d) Multiplataforma, multithread, sustentabilidade e viabilidade.
- e) Nenhuma das alternativas anteriores está correta.

2. Qual é o método principal de um programa Java?

- a) start()
- b) run()
- c) play()
- d) main()
- e) java()

3. Qual o recurso responsável por executar as classes de Java, depois de compilado?

- a) IDE
- b) JVM
- c) JSE
- d) JavaFX
- e) Nenhuma das alternativas anteriores está correta.

4. Representa uma palavra reservada da linguagem Java, exceto:

- a) struct
- b) const
- c) static
- d) private
- e) class

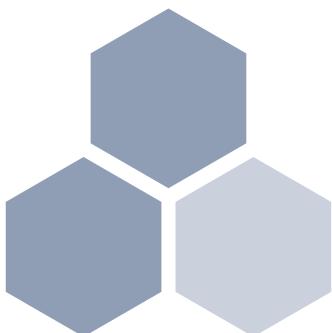
5. Quais são as possibilidades de uma linguagem orientada a objetos?

- a) A criação de programas constituídos por objetos que não se comunicam entre si e não podem ser manipulados.
- b) A criação de programas constituídos por objetos que se comunicam entre si e são manipulados por meio de propriedades e métodos.
- c) A criação de programas constituídos por objetos que se comunicam entre si e são manipulados por meio de outros objetos.
- d) A criação de programas constituídos por objetos que se comunicam entre si e não podem ser manipulados.
- e) Nenhuma das alternativas anteriores está correta.

2

Tipos de dados, literais e variáveis

- Tipos de dados primitivos;
- Literais;
- Variáveis;
- Casting;
- Tipos de referência comuns.



2.1. Introdução

Neste capítulo, você conhecerá alguns elementos essenciais em qualquer trabalho de programação com Java. São os tipos de dados, os literais e as variáveis.

2.2. Tipos de dados primitivos

Java é uma linguagem estaticamente tipada, ou seja, cada expressão e variável possui seu tipo em tempo de compilação. Ela também é fortemente tipada, o que significa que os valores das variáveis e expressões são limitados, bem como as operações relativas aos tipos de valor, e que erros em tempo de compilação são mais facilmente identificados.

Há duas categorias de tipos em Java:

- **Tipos primitivos:** Incluem tipos numéricos (**byte**, **short**, **int**, **long**, **float** e **double**), textuais (**char**) e booleanos (**boolean**), que serão apresentados mais detalhadamente adiante.
- **Tipos de referência:** Os valores desse tipo são referências a objetos. Os tipos de referência incluem tipos de classe, de interface e de array.

 Neste capítulo, abordaremos os tipos primitivos com detalhes e daremos uma visão geral dos tipos de referências mais comuns da linguagem.

Apresentamos, a seguir, os tipos primitivos:

Tipo	Tamanho	Abrangência	Tipo de valor	Utilização
byte	1 byte / 8 bits	-128 a 127 (-2 ⁷ a 2 ⁷ - 1)	Números inteiros	Pode economizar memória em arrays extensas.
short	2 bytes / 16 bits	-32.768 a 32.767 (-2 ¹⁵ a 2 ¹⁵ - 1)	Números inteiros	Também pode ser usado para economizar memória em arrays extensas.
int	4 bytes / 32 bits	-2.147.483.648 a 2.147.483.647 (-2 ³¹ a 2 ³¹ - 1)	Números inteiros	Geralmente, é o tipo padrão para valores inteiros. Possui extensão suficiente para a maior parte dos valores usados em um programa.
long	8 bytes / 64 bits	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (-2 ⁶³ a 2 ⁶³ - 1)	Números inteiros	Quando for preciso usar valores maiores do que é possível em int , pode-se usar long , que é mais extenso.
float	4 bytes / 32 bits	-3.40292347e+38 a 3.40292347e+38	Ponto flutuante	Pode ser usado para economizar memória em arrays de ponto flutuante, mas não funciona para valores precisos.
double	8 bytes / 64 bits	Aproximadamente 1.79769313486231570E+308	Ponto flutuante	Geralmente, é o tipo padrão para valores decimais. Também não funciona para valores precisos.
char	2 bytes / 16 bits	“\u0000” a “\uffff”	Caracteres (Textual)	Representa qualquer caractere alfanumérico, além de símbolos, a partir do padrão Unicode.
boolean	1 bit (pode variar conforme a implementação da JVM)	true ou false	Booleano	Indica condições falsas ou verdadeiras. Representa um bit de informação, mas não tem tamanho definido.

2.3. Literais

Um literal é uma representação, em código-fonte, de um valor fixo. Eles não necessitam de nenhuma operação computacional para serem representados, isso acontece de forma direta no código. Os literais podem ser atribuídos a variáveis de um tipo primitivo, como mostra o seguinte exemplo:

```
public class Literais {  
    public static void main(String args[]){  
        byte b = 100;  
        short s = 0100;  
        int i = 0x100;  
        long l = 100L;  
        float f = 0.000123f;  
        double d = 123d;  
        char c = '\u0022';  
        boolean bo = true;  
  
        System.out.println(b);  
        System.out.println(s);  
        System.out.println(i);  
        System.out.println(l);  
        System.out.println(f);  
        System.out.println(d);  
        System.out.println(c);  
        System.out.println(bo);  
    }  
}
```

Para otimizar a leitura dos códigos, o Java introduziu um separador de dígitos para literais numéricos, que consiste simplesmente em um caractere underline (_). Assim, o valor real 1000000000000 pode ser representado da seguinte forma:

```
int numero = 100_000_000_000_000;
```

Veja, a seguir, os diferentes tipos de literais.

2.3.1. Literais inteiros

Um literal inteiro representa qualquer valor numérico inteiro, que pode ser expresso em base binária (base 2), octal (base 8), decimal (base 10) ou hexadecimal (base 16).

Um literal inteiro, normalmente de tipo **int**, permite criar tipos **byte**, **short** e **long**, além do referido **int**. Literais inteiros de tipo **long** servem para criar valores que excedem o alcance de **int**. Para isso, basta terminar o literal com um caractere **L** (que pode ser minúsculo ou, de preferência, maiúsculo, para não ser confundido com o dígito **1**).

- **Decimal**

Um número decimal é composto por um numeral entre **1** e **9** que pode ou não ser seguido por outros numerais. O numeral **0** não deve ser usado na primeira posição, apenas quando representa mesmo o valor zero, não sendo seguido por nenhum outro numeral. Um decimal representa sempre um inteiro positivo.

Veja um exemplo:

```
int decimal = 100;
```

- **Hexadecimal**

Um número hexadecimal representa inteiros positivos, negativos e com valor zero, e é composto por um **0x** inicial, seguido de valores que vão de **0** a **15**. Os valores de **0** a **9** são representados normalmente por numerais e os de **10** a **15** são representados, respectivamente, pelas letras **A** (**a**), **B** (**b**), **C** (**c**), **D** (**d**), **E** (**e**), **F** (**f**).

Veja um exemplo:

```
int hexadecimal = 0x64; // igual a 100 decimal
```

- **Octal**

Um número octal é composto por um **0** seguido de um ou mais numerais entre **0** e **7**. Representa inteiros positivos, negativos e com valor zero.

Veja um exemplo:

```
int octal = 0144; // igual a 100 decimal
```

- **Binário**

Um número binário é composto por um **0B** (ou **0b**) seguido de um ou mais numerais **0** ou **1**. Pode representar um inteiro positivo, negativo ou com valor zero. Essa forma de declarar literais binários foi introduzida com o Java 7.

Veja um exemplo:

```
int binario = 0b1100100; // igual a 100 decimal
```

2.3.2. Literais de ponto flutuante

Os números de ponto flutuante representam valores decimais compostos de fração. Podem ser representados por um numeral inteiro, seguido de um ponto (e não vírgula) e, então, pela parte fracionária, como em 3.692. Essa é a chamada notação comum. Outra possibilidade é a notação científica, que consiste em um número de ponto flutuante em notação comum acompanhado de um sufixo indicando a potência de 10, pela qual o número será multiplicado. A letra E (e) representa o expoente e é seguida de um número decimal, positivo ou negativo, como em 4.0E+05 (o mesmo que 4*10^5).

Por padrão, literais de ponto flutuante em Java são tratados como **double**. Para representar um número de tipo **float**, você deve anexar um caractere **f** ou **F** (preferencialmente maiúsculo) ao fim do número. Para o caso do tipo **double**, um caractere **D** ou **d** podem ser usados para a mesma finalidade, porém, diferentemente do caso anterior, não são obrigatórios.

Veja um exemplo de código com literal de ponto flutuante:

```
double pontoFlutuante = 123e-3; // igual a 0.123
```

2.3.3. Literais booleanos

Os literais booleanos envolvem os valores lógicos **true** (verdadeiro) e **false** (falso), que não podem ser convertidos em representação numérica. Os literais booleanos só podem ser utilizados em expressões com operadores **boolean** ou, então, atribuídos a variáveis declaradas como **boolean**.

Veja um exemplo de literal booleano:

```
boolean ligado = true;  
boolean desligado = false;
```

2.3.4. Literais de caracteres

O Unicode é um padrão internacional unificado capaz de representar os caracteres de todas as línguas humanas, o que faz do Java uma linguagem de compatibilidade global.

A fim de possibilitar que dezenas de conjuntos de caracteres – dentre os quais o grego, o hebraico, o cirílico e o katakana (uma das representações gráficas do japonês) – sejam reunidos ao Unicode, o tipo **char**, em Java, foi criado com 16 bits. Esse tipo possui alcance de 0 a 65.536 e não adota valores negativos. Os literais de caracteres são, na verdade, índices de 16 bits que apontam para os caracteres Unicode. São manipuláveis e passíveis de conversão.

Para representar um valor literal de caractere, utilizamos um par de aspas simples (''). Podemos digitar entre aspas simples todos os caracteres ASCII visíveis, por exemplo, 'f' e '?'.

Veja um exemplo de literal de caractere:

```
char caractere = 'a';
char caractereUnicode = '\u0061'; // igual ao literal 'a'
```

2.3.4.1. Caracteres de escape

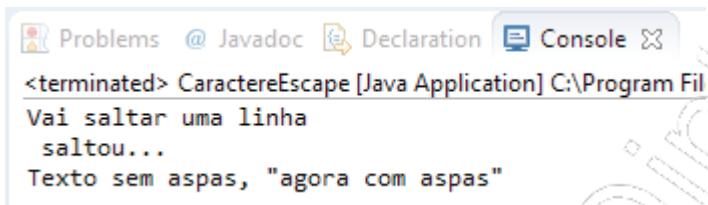
Existem, contudo, caracteres especiais que não podem ser representados diretamente dentro de aspas simples. Não podem ser apresentados visualmente, mas servem para executar alguma ação predefinida. Eles são conhecidos como caracteres ou sequências de escape. Eles são compostos por uma barra invertida e um caractere ou sequência de caracteres, sendo usados com bastante frequência em textos, geralmente armazenados em objetos **String**, além de outros locais onde se emprega conteúdo textual. Veja os principais na tabela adiante:

Caractere (ou sequência) de escape	Descrição
\'	Aspas simples (apóstrofo)
\\"	Aspas duplas
\\\	Barra invertida
\r	Caractere de retorno
\n	Nova linha (alimentação de linha)
\f	Alimentação de formulário
\t	Tabulação
\v	Tabulação vertical
\b	Backspace

A seguir, temos um exemplo do uso de caracteres de escape:

```
public class CaractereEscape {
    public static void main(String args[]){
        System.out.println("Vai saltar uma linha \n saltou...");
        System.out.println("Texto sem aspas, \"agora com aspas\"");
    }
}
```

O resultado é o seguinte:



```
Problems @ Javadoc Declaration Console X
<terminated> CaractereEscape [Java Application] C:\Program Fil
Vai saltar uma linha
saltou...
Texto sem aspas, "agora com aspas"
```

2.4. Variáveis

Variável é um espaço da memória utilizado para armazenar temporariamente um valor. Identificada por um nome, a variável contém um identificador e um tipo, podendo ter um inicializador opcional. A visibilidade e a duração de tempo de uma variável são definidas pelo escopo. O valor que uma variável recebe é determinado pelo programa. Elas podem possuir modificadores ou qualificadores opcionais que conferem comportamento diferenciado, conforme a necessidade do programa.

As variáveis podem ser simples, se armazenarem apenas um valor por vez, ou compostas, se armazenarem dois ou mais valores, identificados em posições por um índice.

Por suas características, as variáveis possuem duas funções básicas:

- Podem servir para ação, já que uma variável pode ser modificada para obter determinado resultado durante o processamento do programa;
- Podem servir para controle, já que durante o processamento é possível acompanhar ou vigiar uma variável.

2.4.1. Definindo uma variável

Para que uma variável seja identificada e possa ser usada em um programa, ela deve ser definida por meio de um nome. Considere as seguintes observações ao nomear, ou definir, uma variável:

- Ela deve iniciar, obrigatoriamente com um caractere alfabético (letra) maiúsculo ou minúsculo, underline (_) ou cífrão (\$);

! No caso de underline (_), não pode este caractere sozinho definir o nome da variável, por ser um símbolo reservado da linguagem.

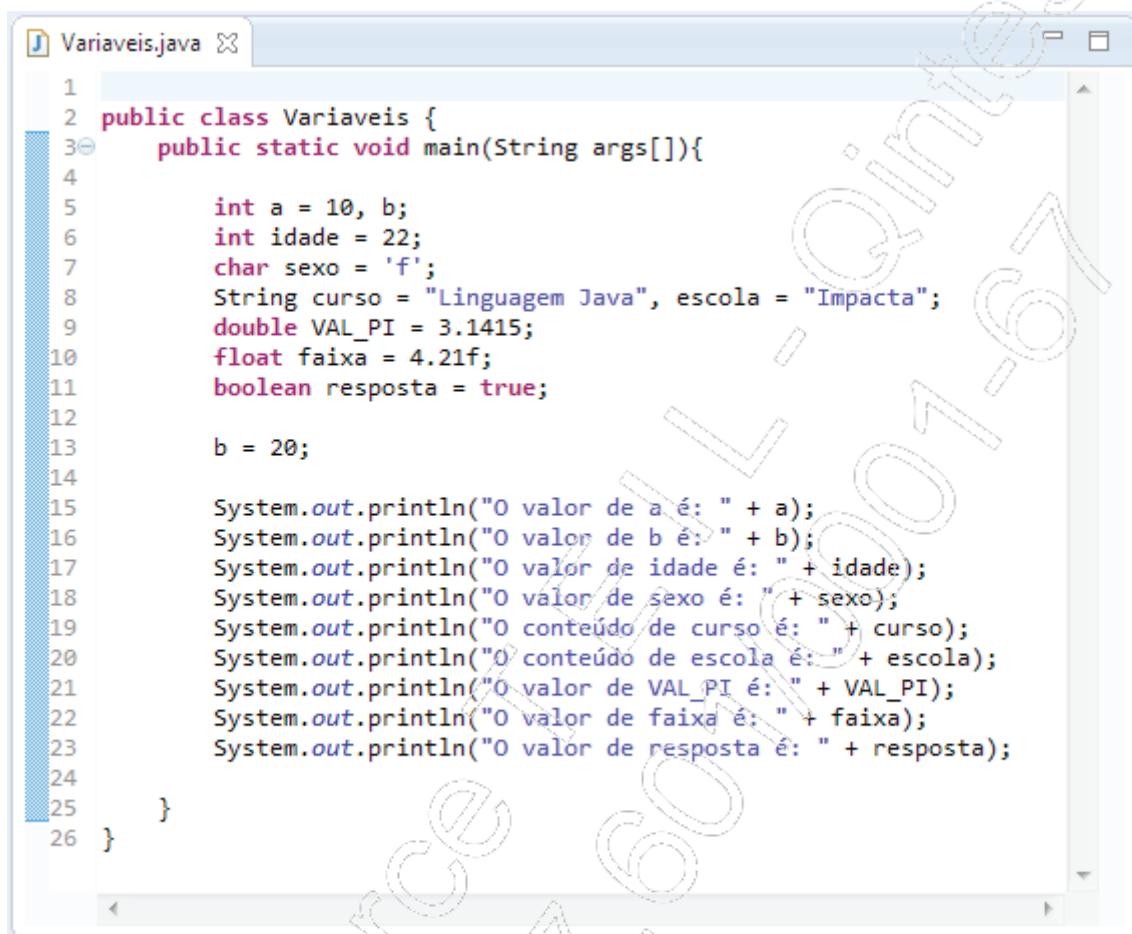
- Os demais caracteres do identificador podem ser letras maiúsculas ou minúsculas, números, underlines (_) ou cífrões (\$);
- Não se pode usar espaços em branco nem quaisquer caracteres de pontuação ou traços;
- A linguagem diferencia entre caracteres maiúsculos e minúsculos;
- Não se pode usar o nome de outra variável;
- Não se pode usar uma palavra reservada da linguagem;
- As variáveis devem ser nomeadas de forma consistente, com a mesma regra valendo para todos os programas desenvolvidos.

2.4.2. Declarando uma variável

Para declarar uma variável em um programa e torná-la, assim, válida para uso, você deve observar a seguinte sintaxe:

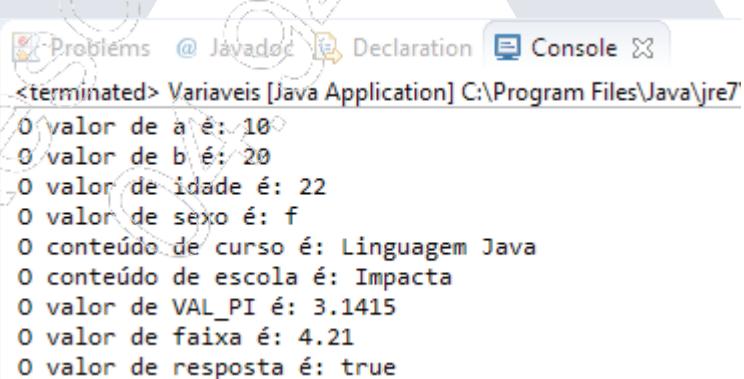
```
<tipo> <nomeVariável>;  
<tipo> <nomeVariável> = <valor>;  
<tipo> <nomeVariável1>, <nomeVariável2>;  
<tipo> <nomeVariável1> = <valor>, <nomeVariável2> = <valor>;
```

Veja alguns exemplos de declaração de variáveis:



```
1 public class Variaveis {
2     public static void main(String args[]){
3
4         int a = 10, b;
5         int idade = 22;
6         char sexo = 'f';
7         String curso = "Linguagem Java", escola = "Impacta";
8         double VAL_PI = 3.1415;
9         float faixa = 4.21f;
10        boolean resposta = true;
11
12        b = 20;
13
14        System.out.println("O valor de a é: " + a);
15        System.out.println("O valor de b é: " + b);
16        System.out.println("O valor de idade é: " + idade);
17        System.out.println("O valor de sexo é: " + sexo);
18        System.out.println("O conteúdo de curso é: " + curso);
19        System.out.println("O conteúdo de escola é: " + escola);
20        System.out.println("O valor de VAL_PI é: " + VAL_PI);
21        System.out.println("O valor de faixa é: " + faixa);
22        System.out.println("O valor de resposta é: " + resposta);
23
24    }
25
26 }
```

Segue o resultado:



```
Problèmes @ Javadoc Declaration Console
<terminated> Variaveis [Java Application] C:\Program Files\Java\jre7
0 valor de a é: 10
0 valor de b é: 20
0 valor de idade é: 22
0 valor de sexo é: f
0 conteúdo de curso é: Linguagem Java
0 conteúdo de escola é: Impacta
0 valor de VAL_PI é: 3.1415
0 valor de faixa é: 4.21
0 valor de resposta é: true
```

2.4.2.1. Usando o qualificador final

Uma variável também pode ser declarada com um qualificador **final**, cuja sintaxe é a seguinte:

```
[final] <tipo> <nome> [=<valor>];
```

Quando utilizamos o modificador final, estamos, na verdade, criando uma constante em Java. Estamos dizendo ao compilador que não queremos que o valor da variável seja alterado após sua atribuição.

Constantes devem ser nomeadas usando um padrão adotado por convenção e largamente utilizado na linguagem Java: todas as letras de seu identificador devem ser maiúsculas, com underlines (_) separando as palavras que o compõem.

Veja um exemplo do qualificador final:

```
final double ACELERACAO_GRAVIDADE = 9.80665;
```

O qualificador final também pode ser usado em outros elementos da linguagem. Sua aplicação nestes outros elementos será abordada posteriormente.

2.4.3. Escopo de variáveis

Escopo refere-se ao limite ou ciclo de vida de um objeto. Um escopo é criado sempre que um novo bloco é iniciado e permite que outros programas visualizem ou não certos objetos, além de estipular o seu tempo de duração.

A definição de escopos das variáveis será mais detalhada quando forem abordadas as estruturas básicas da linguagem Java e a definição de métodos.

2.5. Casting

O **casting**, ou conversão entre variável de tipos primitivos, ocorre de forma natural sempre que precisamos utilizar um valor de tipo “pequeno” em uma variável ou parâmetro de tipo “maior”.

No exemplo a seguir, na segunda linha do código, estamos atribuindo um valor **int** em uma variável **long** e, na quarta linha, atribuindo um valor **float** em uma variável **double**:

```
int ano = 2014;
long anoAtual = ano;

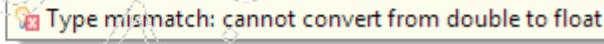
float salarioMinimo = 2000.15f;
double salarioTotal = salarioMinimo;
```

Esta conversão de tipos ocorre naturalmente no Java e é chamada de **autocast**.

Ao tentar realizar uma operação oposta, o Java exibe erros de compilação:

```
long ano = 2014;
int anoAtual = ano;

double salarioMinimo = 2000.15;
float salarioTotal = salarioMinimo;
```



No exemplo anterior, o compilador entende que as atribuições utilizadas estão sendo realizadas com variáveis pequenas demais para aquele tipo.

Podemos facilmente corrigir este código utilizando o **cast** (tipagem) de forma explícita:

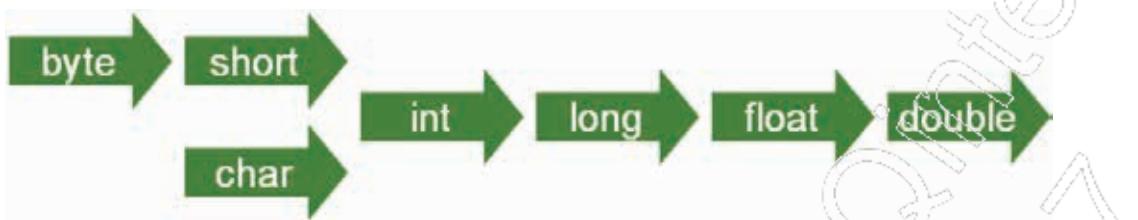
```
long ano = 2014;
int anoAtual = (int) ano;

double salarioMinimo = 2000.15;
float salarioTotal = (float) salarioMinimo;
```

De forma geral, podemos realizar o cast explícito para quaisquer variáveis numéricas primitivas da seguinte forma:

```
variavel = (tipo da variavel) <valor>;
```

O cast explícito é exigido sempre que precisamos realizar atribuições entre variáveis numéricas primitivas que **não** sigam o seguinte fluxo de tipos:



2.6. Tipos de referência comuns

Além dos tipos primitivos, a linguagem Java traz diversas outras classes que são utilizadas como tipos de variáveis: os tipos de referência.

Alguns destes tipos são muito utilizados e fazem parte de quase todo o programa Java.

Apresentaremos neste tópico: **String**, **Enums** e classes **Wrapper**.

2.6.1. String

String representa uma classe que encapsula sequências de caracteres. Quando você coloca um conjunto de caracteres entre aspas duplas, está criando um objeto String, que é um objeto imutável, ou seja, não pode ter seus valores modificados após ser criado.

Um “literal” String é composto por zero ou mais caracteres entre aspas duplas, que podem ser representados também por sequências de escape.

Veja um exemplo de literal String:

```
String texto = "Impacta Tecnologia";
```

Um objeto String possui métodos e atributos que podem ser utilizados para manipulação e pesquisa do texto que ela encapsula. Mais detalhes são apresentados no Apêndice III desta apostila e também podem ser consultados na documentação oficial da linguagem Java em: <https://docs.oracle.com/javase/9/docs/api/java/lang/String.html>.

2.6.2. Enum

Enum é um tipo de dados especial que permite que uma variável assuma qualquer valor de um conjunto predefinido de constantes. A variável deve assumir, obrigatoriamente, um dos valores definidos neste conjunto.

Exemplos comuns de domínios que podem ser assumidos como enums são: pontos cardinais (Norte, Sul, Leste, Oeste), dias da semana, meses do ano e assim por diante.

Convencionalmente, por serem constantes, os nomes dos campos de uma enum são representados com letras maiúsculas.

Para definir uma enum, basta utilizar a palavra-chave **enum**. Por exemplo, para definição de uma enum que represente os dias da semana:

```
public enum DiaDaSemana {  
    SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO, DOMINGO;  
}
```

A seguir, um exemplo de utilização da enum definida:

```
public class UsandoDiaDaSemana {  
  
    public static void main(String[] args) {  
  
        DiaDaSemana diaBom = DiaDaSemana.SABADO;  
        System.out.println("Dia bom é " + diaBom.name());  
    }  
}
```

Para explorar todo o conteúdo relacionado à enum, basta acessar a documentação oficial a respeito em: <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>.

2.6.3. Classes Wrapper

Cada um dos oito tipos primitivos da linguagem possui uma classe correspondente. Estas classes são chamadas de classes **Wrapper** (envelopes), pois encapsulam o tipo primitivo em um objeto.

As classes Wrapper são bastante úteis para dois propósitos essenciais:

- Prover um mecanismo de encapsulamento do valor primitivo em objetos, em contextos onde somente objetos são considerados, como em coleções (listas, conjuntos e tabelas hash);
- Para se utilizar funções utilitárias associadas ao tipo primitivo que são executadas a partir do objeto da classe correspondente.

As duas linhas de código adiante ilustram a diferença entre um tipo de dados primitivo e um objeto de uma classe Wrapper:

```
int x = 13;
Integer y = Integer.valueOf(27);
```

A seguir, os tipos primitivos e suas classes Wrapper correspondentes:

Tipo Primitivo	Classe Wrapper
boolean	Boolean
byte	Byte
char	Character
int	Integer
float	Float
double	Double
long	Long
short	Short

Para obter mais informações a respeito de classes Wrapper, consulte o Apêndice III desta apostila. Além disso, consulte a documentação na página oficial da linguagem: <https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

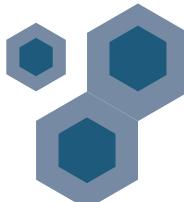
- Java é uma linguagem estaticamente tipada, ou seja, cada expressão e variável possui seu tipo em tempo de compilação. Ela também é fortemente tipada, o que significa que os valores das variáveis e expressões são limitados, bem como as operações relativas aos tipos de valor, e que erros em tempo de compilação são mais facilmente identificados. Os tipos primitivos são; **byte, short, int, long, float, double, char e boolean**;
- Um literal é uma representação, em código-fonte, de um valor fixo. Eles são representados diretamente no código, sem mediação de nenhuma operação computacional. Os literais podem ser inteiros, booleanos, de ponto flutuante, de caracteres ou de strings (cadeias de caracteres);
- Variável é um espaço da memória utilizado para armazenar temporariamente um valor. Uma variável é identificada por um nome, contém um identificador e um tipo e pode ter um inicializador e modificador opcionais. A visibilidade e a duração de tempo de uma variável são definidas pelo escopo. As variáveis podem ser simples ou compostas;
- A conversão (casting) de valores é permitida entre os tipos **byte, short, int, long, float, long e char** e precisa ser explícita quando atribuímos valores de tipos maiores em variáveis de tipos menores;
- A linguagem Java possui tipos não primitivos, mas que fazem parte do núcleo básico da linguagem: **Strings, enums e classes Wrapper**.



2

Tipos de dados, literais e variáveis

Teste seus conhecimentos



1. Qual das seguintes alternativas não é um tipo primitivo?

- a) byte
- b) int
- c) char
- d) String
- e) boolean

2. O literal 42.23f representa que tipo primitivo?

- a) long
- b) float
- c) double
- d) int
- e) boolean

3. Qual sequência de caracteres de escape tem a função de pular uma linha?

- a) \\
- b) \r
- c) \t
- d) \n
- e) \b

4. Qual das seguintes declarações de variáveis não está correta?

- a) int a = 10, b = 30
- b) boolean a, b, c, d, e
- c) double a, b = 10
- d) numero int = 10
- e) Nenhuma das alternativas anteriores está correta.

5. Qual é a função do qualificador final em uma variável?

- a) Finalizar a variável.
- b) Impedir que o valor da variável seja alterado.
- c) Finalizar o programa.
- d) Impedir que o programa utilize a variável.
- e) Nenhuma das alternativas anteriores está correta.

3

Operadores

- ◆ Operador de atribuição;
- ◆ Operadores aritméticos;
- ◆ Operadores incrementais e decrementais;
- ◆ Operadores relacionais;
- ◆ Operadores lógicos;
- ◆ Operador ternário;
- ◆ Precedência dos operadores.

3.1. Introdução

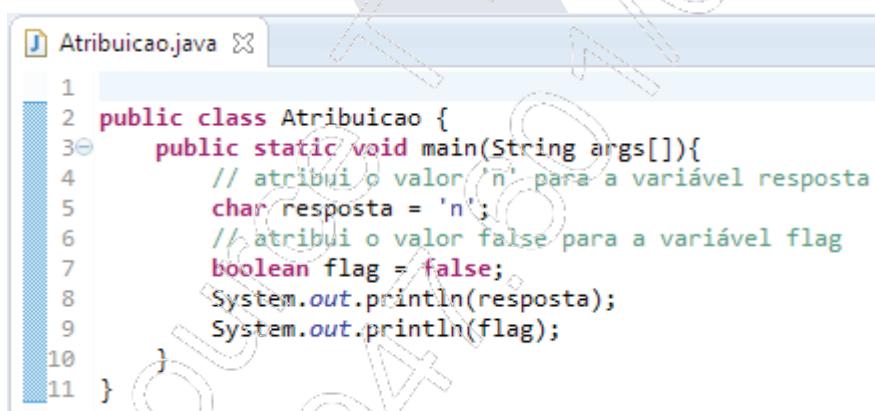
Em Java, encontramos um grande número de operadores, os quais permitem formar diversas expressões e se destinam a realizar as mais variadas operações. Ao longo deste capítulo, veremos os principais tipos de operadores.

3.2. Operador de atribuição

O símbolo de igualdade utilizado em matemática (=) representa a atribuição de um valor a uma variável ou constante, sendo necessário que a variável ou constante e o valor atribuído sejam de tipos compatíveis. Assim, uma variável do tipo **char** não pode receber um valor do tipo **boolean** (true ou false). Observe:

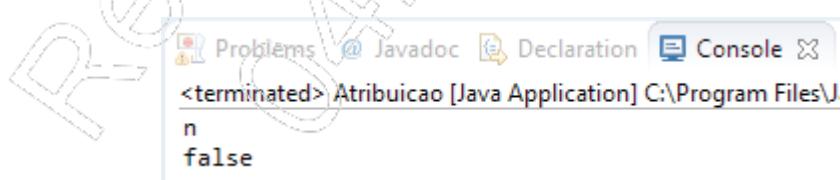
```
<tipo> <nomeVariávelOuConstante> = <valorAtribuição>;
```

Exemplo:



```
1  public class Atribuicao {
2      public static void main(String args[]){
3          // atribui o valor 'n' para a variável resposta
4          char resposta = 'n';
5          // atribui o valor false para a variável flag
6          boolean flag = false;
7          System.out.println(resposta);
8          System.out.println(flag);
9      }
10 }
11 }
```

Depois de compilado e executado o código, o resultado será o seguinte:



```
Problems Javadoc Declaration Console
<terminated> Atribuicao [Java Application] C:\Program Files\J
n
false
```

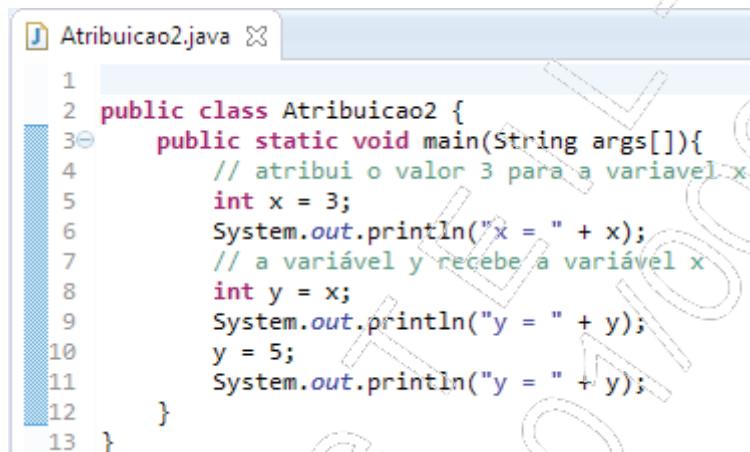
Uma das possibilidades que temos com esse operador é atribuir uma variável primitiva a outra variável primitiva. Veja como isso ocorre no exemplo a seguir:

```
int x = 3;
int y = x;
```

Em que:

- **x** recebe o valor **3**;
- **y** recebe a variável **x**, logo **y** contém o valor **3**.

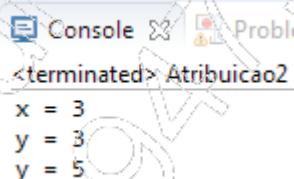
Neste momento, as duas variáveis (**x**, **y**) têm o mesmo valor, porém, se alterarmos o valor de uma delas, a outra não será alterada. Isso ocorre porque o operador de atribuição realiza a cópia do valor de uma variável à outra. Nesse caso, **y** recebe uma cópia do valor armazenado em **x**. Veja o exemplo adiante:



```

1
2 public class Atribuicao2 {
3     public static void main(String args[]){
4         // atribui o valor 3 para a variavel x
5         int x = 3;
6         System.out.println("x = " + x);
7         // a variável y recebe a variável x
8         int y = x;
9         System.out.println("y = " + y);
10        y = 5;
11        System.out.println("y = " + y);
12    }
13 }
```

A compilação e a execução desse código resultarão no seguinte:



```

<terminated> Atribuicao2
x = 3
y = 3
y = 5
```

Toda atribuição possui um valor de retorno que é exatamente o valor da atribuição. Em outras palavras, a seguinte expressão:

```
endereço = "Rua X";
```

Elá está atribuindo o valor “Rua X” à variável **endereço** e está retornando o próprio valor “Rua X”. Este valor de retorno pode ser reutilizado em outra expressão, por exemplo, em outra atribuição:

```
novoEndereco = (endereço = "Rua X");
```

Os parênteses são desnecessários. A ideia desta expressão é: realize a atribuição de “Rua X” na variável **endereco** e coloque o resultado disto (que é o texto “Rua X”) na variável **novoEndereco**.

Desta forma, podemos realizar sucessivas atribuições, como mostrado adiante:

```
a = b = c = d = e = f = 8;
```

Aqui, o Java vai colocar o valor 8 em todas as variáveis, na seguinte ordem: f, e, d, c, b, a.

3.3. Operadores aritméticos

Os operadores aritméticos são utilizados para cálculos matemáticos e são os seguintes:

Operador aritmético	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)

Os operadores + e - também podem significar, respectivamente, positivo e negativo, ou manutenção e inversão de sinal, em casos como $+x$ e $-y$, em que x e y são as variáveis.

O exemplo a seguir mostra como podemos usar esses operadores:

```
1  public class OperadoresAritmeticos {
2      public static void main(String args[]){
3          int a = 10, b = 25, c = 40, d = 55;
4          System.out.println(a + b);
5          System.out.println(b - a);
6          System.out.println(c * d);
7          System.out.println(c / a);
8          System.out.println(d % a);
9      }
10 }
```

Após a compilação e a execução do código, o resultado será:

```

Problems @ Javadoc Declaration Console X
<terminated> OperadoresAritmeticos [Java Application] C:\Pro
35
15
2200
4
5

```

3.3.1. Operadores aritméticos de atribuição reduzida

Os operadores aritméticos de atribuição reduzida são utilizados para compor uma operação aritmética e uma atribuição. Eles são os seguintes:

Operador aritmético	Operação equivalente	Descrição
<code>x += 3</code>	<code>x = x + 3</code>	Mais igual
<code>x -= 3</code>	<code>x = x - 3</code>	Menos igual
<code>x *= 3</code>	<code>x = x * 3</code>	Vezes igual
<code>x /= 3</code>	<code>x = x / 3</code>	Dividido igual
<code>x %= 3</code>	<code>x = x % 3</code>	Módulo igual

No exemplo a seguir, utilizamos um desses operadores para acrescentar o valor 3 à variável x:

```

AtribuicaoReduzida.java X
1
2 public class AtribuicaoReduzida {
3     public static void main(String args[]){
4         int x = 4;
5         x += 3; // é o mesmo que x = x + 3;
6         System.out.println("Valor de x = " + x);
7     }
8 }

```

Após a compilação e a execução do código, o resultado será o seguinte:

```

Problems @ Javadoc Declaration Console X
<terminated> AtribuicaoReduzida [Java Application] C:\Program
Valor de x = 7

```

Ao utilizar operadores de atribuição, você tem dois benefícios: além de serem implementados eficientemente em tempo de execução, o trabalho gasto com a digitação do código é reduzido. Isso ocorre porque os operadores de atribuição funcionam como se fossem uma versão mais reduzida das operações equivalentes.

3.4. Operadores incrementais e decrementais

Os operadores incrementais e decrementais têm a função de aumentar ou diminuir o valor de uma variável em 1 (um). Eles podem ser pré-/pós-incrementais ou pré-/pós-decrementais. Veja:

- **Incremental (++)**
 - **Pré-incremental ou prefixo:** Se o sinal for colocado antes da variável, seu valor será aumentado em um antes da expressão ser resolvida;
 - **Pós-incremental ou sufixo:** Se o sinal for colocado após a variável, a expressão (adição, subtração, multiplicação etc.) será resolvida primeiro e, em seguida, o valor da variável será aumentado em um.
- **Decremental (--)**
 - **Pré-decremental ou prefixo:** Se o sinal for colocado antes da variável, seu valor será diminuído em um, antes da expressão ser resolvida;
 - **Pós-decremental ou sufixo:** Se o sinal for colocado após a variável, a expressão (adição, subtração, multiplicação etc.) será resolvida primeiro e, em seguida, o valor da variável será diminuído em um.

O exemplo a seguir ilustra o uso dos operadores incrementais e decrementais:

```
J OperadoresIncrementaisDecrementais.java X
1
2 public class OperadoresIncrementaisDecrementais {
3     public static void main(String args[]){
4         int x = 4;
5         System.out.println( x++ );
6         // Aqui o x é igual a 5
7         System.out.println( ++x );
8         // Aqui o x é igual a 6
9         System.out.println( x-- );
10        // Aqui o x é igual a 5
11        System.out.println( --x );
12        // Aqui o x é igual a 4
13    }
14 }
```

O resultado é o seguinte:

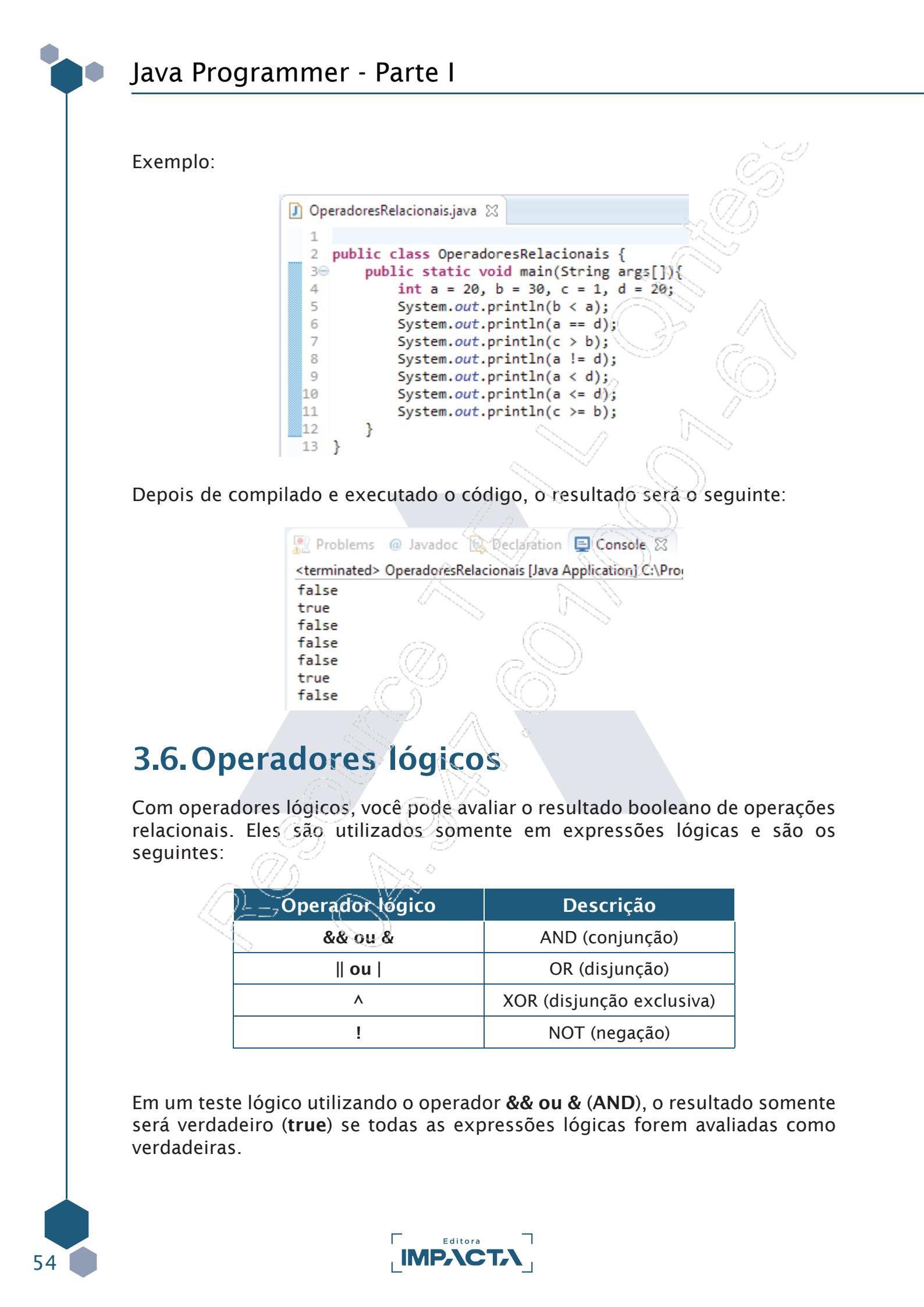
```
Problems @ Javadoc Declaration Console X
<terminated> OperadoresIncrementaisDecrementais [Java App]
4
6
6
4
```

3.5. Operadores relacionais

Os operadores relacionais compararam dois valores numéricos ou expressões que avaliem resultados numéricos e retornam um resultado booleano (**true** ou **false**). Veja quais são eles:

Operador relacional	Descrição
<code>==</code>	Igual a
<code>!=</code>	Diferente de
<code>></code>	Maior que
<code><</code>	Menor que
<code>>=</code>	Maior ou igual a
<code><=</code>	Menor ou igual a

Exemplo:



```
J OperadoresRelacionais.java X
1
2 public class OperadoresRelacionais {
3     public static void main(String args[]){
4         int a = 20, b = 30, c = 1, d = 20;
5         System.out.println(b < a);
6         System.out.println(a == d);
7         System.out.println(c > b);
8         System.out.println(a != d);
9         System.out.println(a < d);
10        System.out.println(a <= d);
11        System.out.println(c >= b);
12    }
13 }
```

Depois de compilado e executado o código, o resultado será o seguinte:



```
Problems @ Javadoc Declaration Console X
<terminated> OperadoresRelacionais [Java Application] C:\Pro...
false
true
false
false
false
true
false
```

3.6. Operadores lógicos

Com operadores lógicos, você pode avaliar o resultado booleano de operações relacionais. Eles são utilizados somente em expressões lógicas e são os seguintes:

Operador lógico	Descrição
&& ou &	AND (conjunção)
ou	OR (disjunção)
^	XOR (disjunção exclusiva)
!	NOT (negação)

Em um teste lógico utilizando o operador **&& ou &** (AND), o resultado somente será verdadeiro (**true**) se todas as expressões lógicas forem avaliadas como verdadeiras.

Se o operador utilizado for **|| ou | (OR)**, basta que uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro.

O operador **\wedge (XOR)** retorna verdadeiro (**true**) somente se as duas expressões tiverem resultados distintos entre si, ou seja, se uma for verdadeira, a outra tem que ser falsa e vice-versa.

Já o operador lógico **! (NOT)** é utilizado para gerar uma negação, invertendo a lógica de uma expressão.

Para ilustrar o comportamento de cada um dos operadores, observe as seguintes tabelas-verdade:

- Operador **&& ou & (AND)** – Utiliza dois operandos:

Operando 1	Operando 2	Resultado
false	false	false
false	true	false
true	false	false
true	true	true

- Operador **|| ou | (OR)** – Utiliza dois operandos:

Operando 1	Operando 2	Resultado
false	false	false
false	true	true
true	false	true
true	true	true

- Operador **\wedge (XOR)** – Utiliza dois operandos:

Operando 1	Operando 2	Resultado
false	false	false
false	true	true
true	false	true
true	true	false

- Operador ! (NOT) – Utiliza somente um operando:

Operando	Resultado
true	false
false	true

A seguir, temos um exemplo de como utilizar operadores lógicos:

```
1  public class OperadoresLogicos {  
2      public static void main(String args[]){  
3          int a = 20, b = 30, c = 1, d = 20;  
4          System.out.println(b < a || c == 1);  
5          System.out.println(a == d && d != a);  
6          System.out.println(!(c > b));  
7      }  
8  }
```

A compilação e a execução do código terão o seguinte resultado:

```
Problems @ Javadoc Declaration Console  
<terminated> OperadoresLogicos [Java Application] C:\Progra  
true  
false  
true
```

Um ponto importante a respeito dos operadores **AND** e **OR** é a diferença na utilização dos símbolos de forma dupla (**&&** e **||**), chamada **curto-circuito** em relação à sua utilização de forma unitária (**&** e **|**).

No curto-circuito, dependendo do resultado de uma das expressões, já se retorna o resultado. Por exemplo: se uma das expressões for **false**, o resultado de um **AND** sempre será falso, independentemente das demais expressões; já usando um **OR**, a situação se inverte: uma das expressões sendo verdadeira, o resultado é verdadeiro, independentemente das demais (isso pode ser verificado nas tabelas-verdade anteriores); já utilizando a forma unitária, todas as expressões são avaliadas independentemente dos resultados parciais.

3.7. Operador ternário

O operador ternário ou operador condicional é composto por três operandos separados pelos sinais ? e :. Seu objetivo é atribuir determinado valor a uma variável de acordo com o resultado de um teste lógico. Sua sintaxe é a seguinte:

```
<variávelOuConstante> = <testeLogico> ? <valorSeVerdadeiro> :  
<valorSeFalso>;
```

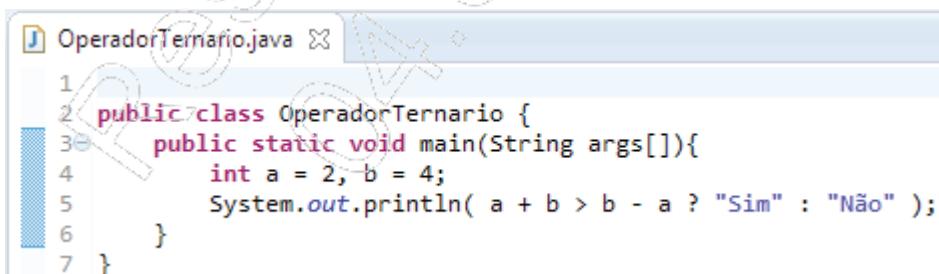
Esse operador simplifica uma operação de desvio condicional (assunto que será tratado posteriormente) e possui a seguinte estrutura:

```
if (<teste lógico>){  
    <variávelOuConstante> = <valorSeVerdadeiro>;  
} else {  
    <variávelOuConstante> = <valorSeFalso>  
}
```

Em que:

- **teste lógico** é qualquer valor ou expressão que pode ser avaliado como verdadeiro ou falso;
- **valor se verdadeiro** é o valor atribuído se o teste lógico for avaliado como verdadeiro;
- **valor se falso** é o valor atribuído se o teste lógico for avaliado como falso.

Veja um exemplo:



```
OperadorTernario.java  
1  
2 public class OperadorTernario {  
3     public static void main(String args[]){  
4         int a = 2, b = 4;  
5         System.out.println( a + b > b - a ? "Sim" : "Não" );  
6     }  
7 }
```

O resultado da compilação e execução do código será o seguinte:



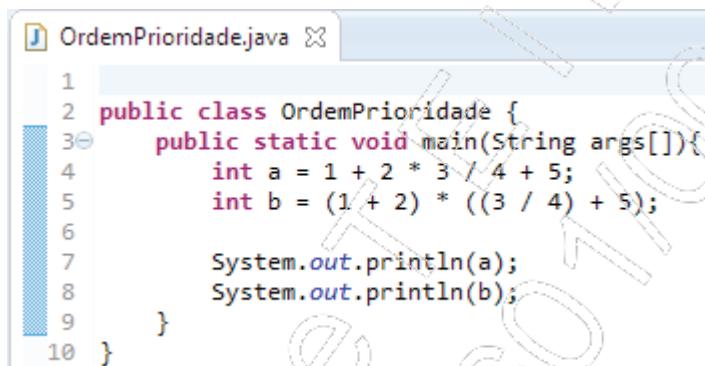
A screenshot of a Java IDE interface. The title bar shows 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Console' (with an 'X'). Below the title bar, it says '<terminated> OperadorTernario [Java Application] C:\Program'. Underneath, there is a single line of text: 'Sim'.

3.8. Precedência dos operadores

Em uma expressão com vários operadores e operandos, a precedência dos operadores determina a ordem em que a expressão será resolvida. A operação de maior precedência é efetuada primeiro. A ordem de prioridade dos operadores pode ser observada na tabela a seguir:

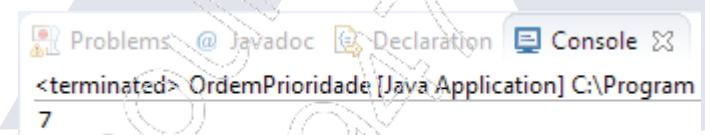
Operadores em ordem de prioridade	
()	[] .
++ -- ~ !	
*	/ %
+	-
>	>= < <=
== !=	
&	
^	
&&	
? :	
= += -= *= /= %=	

O exemplo a seguir ilustra como funciona a precedência de determinados operadores sobre outros:



```
1
2 public class OrdemPrioridade {
3     public static void main(String args[]){
4         int a = 1 + 2 * 3 / 4 + 5;
5         int b = (1 + 2) * ((3 / 4) + 5);
6
7         System.out.println(a);
8         System.out.println(b);
9     }
10 }
```

Após a compilação e execução do código, o resultado é o seguinte:



```
7
15
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

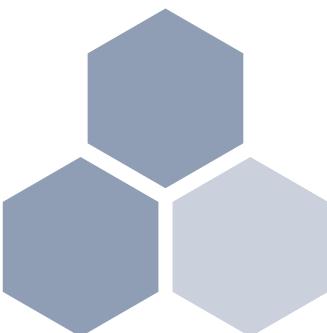
- O operador de atribuição (`=`) define um valor para uma variável ou constante, sendo que a variável ou constante e o valor atribuído devem ser de tipos compatíveis;
- Os operadores aritméticos são utilizados para cálculos matemáticos;
- Os operadores incrementais e decrementais têm a função de aumentar ou diminuir o valor de uma variável em 1;
- Os operadores relacionais comparam dois valores ou expressões numéricas e retornam um resultado booleano;
- Os operadores lógicos servem para avaliar o resultado booleano de operações com expressões relacionais;
- O operador ternário ou operador condicional atribui determinado valor a uma variável ou constante de acordo com o resultado de um teste lógico;
- Os operadores possuem determinada ordem de precedência, que define a ordem em que uma expressão será resolvida.



3

Operadores

Teste seus conhecimentos



1. Qual é o resultado do código a seguir?

```
int b = 5;  
b *= 10;  
System.out.println(b);
```

- a) 10
- b) 5
- c) 25
- d) 50
- e) Nenhuma das alternativas anteriores está correta.

2. Marque a opção em que aparecem somente os operadores aritméticos disponíveis em Java:

- a) +, >, -, =
- b) ++, --, ==
- c) +, -, *, /, %
- d) ++, --, **, //
- e) Nenhuma das alternativas anteriores está correta.

3. Marque a opção em que aparecem somente os operadores relacionais disponíveis em Java:

- a) ||, &&, !, +, -
- b) ==, !=, >, <, >=, <=
- c) ==, ++, --
- d) >, <, =, /, *
- e) Nenhuma das alternativas anteriores está correta.

4. Qual operador lógico retorna verdadeiro apenas quando todas as expressões lógicas são verdadeiras?

- a) &&
- b) ||
- c) ++
- d) !
- e) Nenhuma das alternativas anteriores está correta.

5. Qual é a função dos operadores incrementais?

- a) Diminuir o valor de uma variável em 1.
- b) Dobrar o valor de uma variável.
- c) Aumentar o valor de uma variável em 1.
- d) Dividir o valor de uma variável por 10.
- e) Nenhuma das alternativas anteriores está correta.



3

Operadores



Mãos à obra!

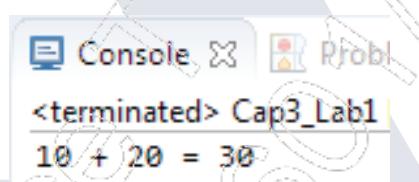


Laboratório 1

A – Somando dois valores e mostrando o resultado

1. Crie uma classe e insira a estrutura básica de um programa Java;
2. Declare três variáveis do tipo **int** com os nomes **valor1**, **valor2** e **resultado**;
3. Atribua valores para as variáveis **valor1** e **valor2**;
4. Atribua o resultado da soma das variáveis **valor1** e **valor2** na variável **resultado**;
5. Imprima o resultado na tela;
6. Compile e execute o programa.

O resultado deve ser como o exibido a seguir:

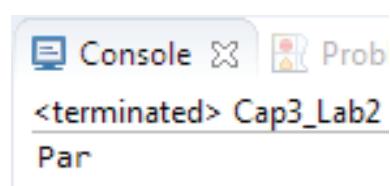


Laboratório 2

A – Verificando se um número é par ou ímpar

1. Crie uma classe e insira a estrutura básica de um programa Java;
2. Declare uma variável do tipo **int** com o nome **valor** e atribua um valor;
3. Usando o operador ternário, verifique se o número é par ou ímpar e imprima na tela;
4. Compile e execute o programa.

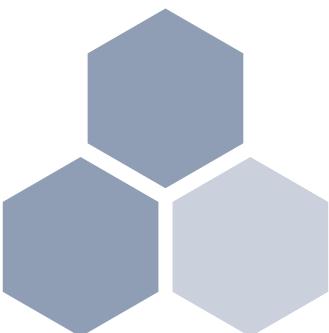
O resultado deve ser como o exibido a seguir:



4

Estruturas de controle

- Estruturas de desvios condicionais;
- Estruturas de repetição;
- Outros comandos.



4.1. Introdução

Neste capítulo, abordaremos dois tipos de estruturas fundamentais para qualquer tipo de programação em Java: as estruturas de desvios condicionais e as estruturas de repetição. Veremos, também, alguns comandos que podem auxiliar na utilização desses recursos.

4.2. Estruturas de desvios condicionais

As estruturas de desvios condicionais permitem desviar o fluxo de execução de um programa de acordo com determinadas condições. Em Java, você utiliza as estruturas **if**, **if / else** e **switch**, caso queira executar trechos diferentes de um programa com base em determinadas condições. Para facilitar a aplicação de cada uma delas, podemos dividi-las em três categorias:

- Estrutura de desvio simples: **if**;
- Estrutura de desvio duplo: **if / else**;
- Estrutura de desvio múltiplo: **switch**.

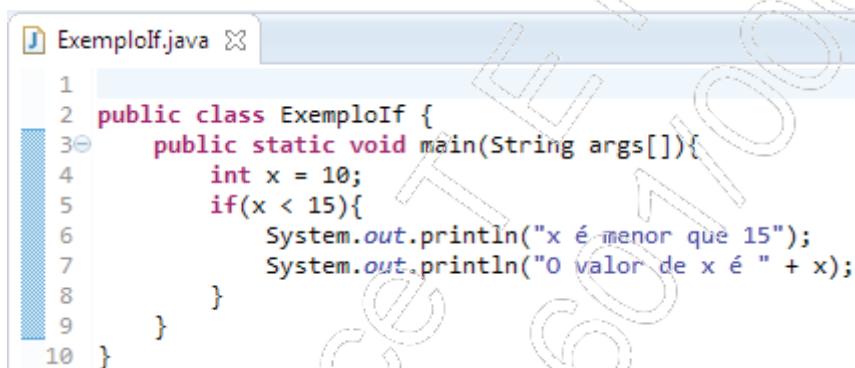
Veremos essas estruturas adiante.

4.2.1. if / else

Você pode usar uma instrução **if** para avaliar expressões com resultados booleanos. Essa construção é responsável por realizar desvios simples no fluxo do código Java. A sintaxe é a seguinte:

```
if (<teste condicional>){  
    Comandos;  
}
```

Caso a expressão booleana seja avaliada como verdadeira, o bloco de instruções entre as chaves (`{ }`) é executado. Caso contrário, o bloco não é executado. Veja um exemplo:



```
ExemploIf.java  
1  
2 public class ExemploIf {  
3     public static void main(String args[]){  
4         int x = 10;  
5         if(x < 15){  
6             System.out.println("x é menor que 15");  
7             System.out.println("O valor de x é " + x);  
8         }  
9     }  
10 }
```

A compilação e a execução desse código resultarão no seguinte:



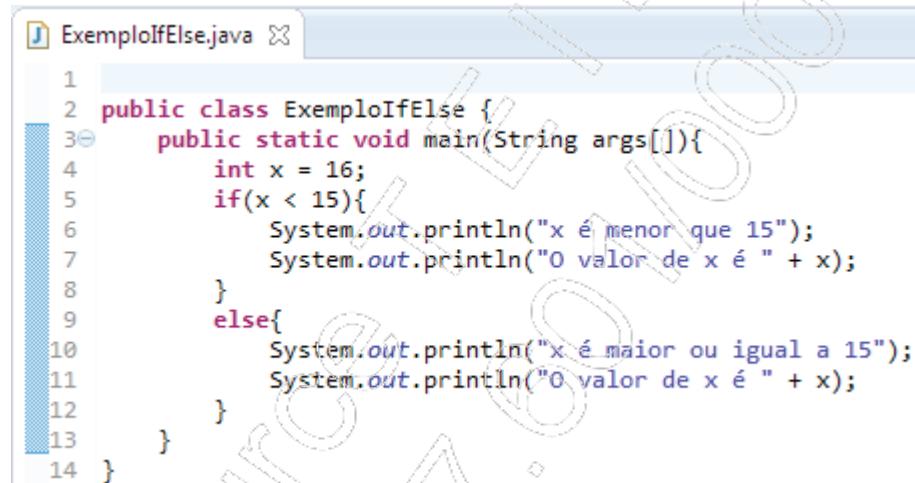
```
Problems @ Javadoc Declaration Console  
<terminated> ExemploIf [Java Application] C:\Program Files\Java  
x é menor que 15  
O valor de x é 10
```

! Quando há apenas uma instrução dentro de um bloco de comandos, as chaves não são obrigatórias. Contudo, sua utilização é considerada uma boa prática e recomendada por facilitar a legibilidade do código.

Além de usar uma instrução simples **if**, você pode usar também estruturas condicionais compostas, em que há uma instrução a ser executada caso a condição seja verdadeira, e também uma outra instrução a ser executada caso a condição seja falsa. É a estrutura **if / else**, responsável por desvios duplos na linguagem Java, cuja sintaxe é a seguinte:

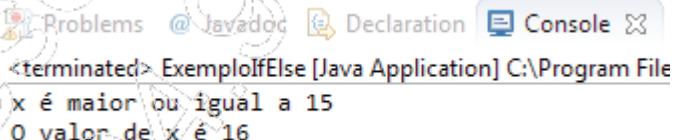
```
if (<teste condicional>) {  
    comandos;  
} else {  
    comandos;  
}
```

Veja, a seguir, um exemplo com a instrução **if / else**:



```
1  
2 public class ExemploIfElse {  
3     public static void main(String args[]){  
4         int x = 16;  
5         if(x < 15){  
6             System.out.println("x é menor que 15");  
7             System.out.println("O valor de x é " + x);  
8         }  
9         else{  
10            System.out.println("x é maior ou igual a 15");  
11            System.out.println("O valor de x é " + x);  
12        }  
13    }  
14 }
```

O resultado da compilação e execução desse código é o seguinte:



```
Problems @ Javadoc Declaration Console  
<terminated> ExemploIfElse [Java Application] C:\Program File  
x é maior ou igual a 15  
O valor de x é 16
```

A estrutura **if / else** pode ser utilizada em cascata, quando existirem diversas condições a serem avaliadas:

```
if (<teste condicional 1>) {
    comandos;
} else if (<teste condicional 2>) {
    comandos;
} else if (<teste condicional 3>) {
    comandos;
} else {
    comandos;
}
```

A seguir, temos um exemplo de utilização do **if / else** em cascata:

```
1  public class ExemploIfElseCascata {
2      public static void main(String[] args) {
3          int hora = 17;
4          if (hora < 12) {
5              System.out.println("Bom dia!");
6          } else if (hora < 18) {
7              System.out.println("Boa tarde!");
8          } else {
9              System.out.println("Boa noite!");
10         }
11     }
12 }
```

4.2.2. switch

Em algumas situações, pode haver uma sequência muito longa de desvios condicionais, o que dificulta a interpretação do programa. Para este tipo de situação, existe a estrutura **switch**, responsável pela criação de desvios múltiplos na linguagem Java. A sua sintaxe é a seguinte:

```
switch (<variável>){
    case <opção 1>: <operação 1>; break;
    case <opção 2>: <operação 2>; break;
    case <opção 3>: <operação 3>; break;
    default: <operação default>;
}
```

Em que:

- <variável> refere-se à variável observada, cujo valor atribuído será controlado;
- <opção> indica o conteúdo literal da variável que será avaliado;
- <operação> indica a execução de um ou vários comandos em cada <case>.

Em **switch**, você pode usar como variável observada os seguintes tipos de dados: **byte**, **char**, **int**, **short**, **enum** e **String** (suporte disponível a partir do Java 7).

A diretiva **case** avalia somente o argumento que possui o mesmo tipo definido em **switch**. Contudo, o argumento de **case** é final, ou seja, ele deve ser resolvido em tempo de compilação. Por isso, utilizamos, em **case**, uma variável final constante com valor literal. Se mais de uma diretiva **case** possuir o mesmo valor, ela não será válida.

! A instrução **switch** apenas verifica igualdades, o que não é o caso de operadores relacionais, tais como menor que ou igual (\leq).

Durante a execução de uma instrução **switch**, o programa verificará, por igualdade, se o valor contido na variável observada é abordado em algum dos **cases** e, em caso positivo, executará o bloco de comandos respectivos até encontrar o comando **break**, saindo imediatamente do **switch** a partir deste ponto. Se, após iniciada a execução do bloco de comandos, não for encontrado nenhum **break**, a execução continuará até que algum **break** esteja à frente ou termine o **switch**.

! Veremos o comando **break** mais detalhadamente em um tópico posterior.

Quando a execução do programa passa de uma diretiva **case** para a seguinte, temos o processamento conhecido como passagem completa, que é a execução de todas as instruções **case** até o final da instrução **switch**.

A estrutura **switch / case** não pode utilizar operadores lógicos (**&&**, **||** ou **!**). Sendo assim, você só pode utilizá-la para comparar valores simples, cujos tipos de dados sejam **int**, **short**, **byte**, **char**, **String** ou **enum**.

Veja, a seguir, um exemplo com a instrução **switch / case**:

```
1  public class ExemploSwitchCase {
2      public static void main(String args[]){
3          int mes = 7;
4          switch(mes){
5              case 1: System.out.println("Janeiro"); break;
6              case 2: System.out.println("Fevereiro"); break;
7              case 3: System.out.println("Março"); break;
8              case 4: System.out.println("Abril"); break;
9              case 5: System.out.println("Maio"); break;
10             case 6: System.out.println("Junho"); break;
11             case 7: System.out.println("Julho"); break;
12             case 8: System.out.println("Agosto"); break;
13             case 9: System.out.println("Setembro"); break;
14             case 10: System.out.println("Outubro"); break;
15             case 11: System.out.println("Novembro"); break;
16             case 12: System.out.println("Dezembro"); break;
17         }
18     }
19 }
20 }
```

A compilação e a execução desse código têm o seguinte resultado:

```
Problems @ Javadoc Declaration Console <terminated> ExemploSwitchCase [Java Application] C:\Program Julho
```

Em uma estrutura **switch / case**, podemos utilizar um comando **default**, que será executado sempre que o valor assumido pela variável não encontrar um valor **case** correspondente. É comum que o comando **default** esteja ao fim do **switch**, após todos os **cases**. Nesse caso, o **break**, nesse bloco, torna-se dispensável.

Veja, no exemplo a seguir, o uso da seção **default** e da cláusula **break** em alguns pontos estratégicos do bloco **switch**:

```
1  public class ExemploSwitchCasePaises {
2      public static void main(String[] args) {
3
4          String pais = "argentina";
5
6
7          switch (pais) {
8              case "brasil":
9                  case "portugal":
10                     System.out.println("Bom dia!");
11                     break;
12                     case "franca":
13                         System.out.println("Bon jour!");
14                         break;
15                     case "mexico":
16                     case "argentina":
17                     case "espanha":
18                         System.out.println("Buenos dias!");
19                         break;
20                     default:
21                         System.out.println("Good morning!");
22
23     }
24 }
25 }
```

A compilação e a execução desse código têm o seguinte resultado:

```
<terminated> ExemploSwitchCasePaises [Java Application] C:\Java\jdk\jdk1.8
Buenos dias!
```

4.3. Estruturas de repetição

Muitas vezes, precisamos repetir a execução de um bloco de códigos do programa até que uma determinada condição seja verdadeira, ou até que o bloco seja executado uma determinada quantidade de vezes. Para que essa repetição seja possível, utilizamos os laços de repetição **while**, **do/while** e **for**. Com essas estruturas, você pode criar contadores e temporizadores, bem como rotinas para obtenção, classificação e recuperação de dados. Elas são abordadas adiante, mas antes vale apresentar as condições indispensáveis a serem observadas em cada espécie de laço. Toda estrutura de repetição deve ser composta por três componentes obrigatórios, que garantem a boa execução do código:

- **Inicialização:** Inicia a variável ou conjunto de variáveis a serem usadas nos testes do laço;
- **Teste de permanência no laço:** Valor booleano ou expressão que avalia para um tipo booleano e que representa a condição de continuidade do laço;
- **Teste de saída do laço:** Comando a ser aplicado em algum ponto interno do laço, de forma a verificar se a condição do teste de permanência continua válida. Sua ausência é a principal causa da criação de laços infinitos.

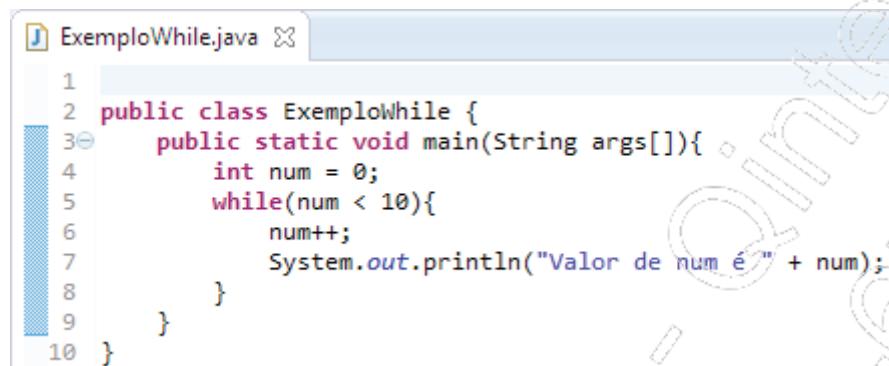
4.3.1. While

Quando não sabemos quantas vezes um determinado bloco de instruções deve ser executado, utilizamos o laço de repetição **while**. Com ele, a execução das instruções vai continuar enquanto uma condição for verdadeira. Veja a sintaxe do laço de repetição **while**:

```
while (<teste condicional>){  
    comandos;  
}
```

As instruções no bloco são executadas somente após a avaliação da expressão. Para isso, a condição avaliada tem que ser verdadeira. Caso seja avaliada como falsa, as instruções não serão executadas. Assim, é possível que as instruções nunca sejam executadas, caso a expressão seja inicialmente avaliada como falsa.

Veja um exemplo:

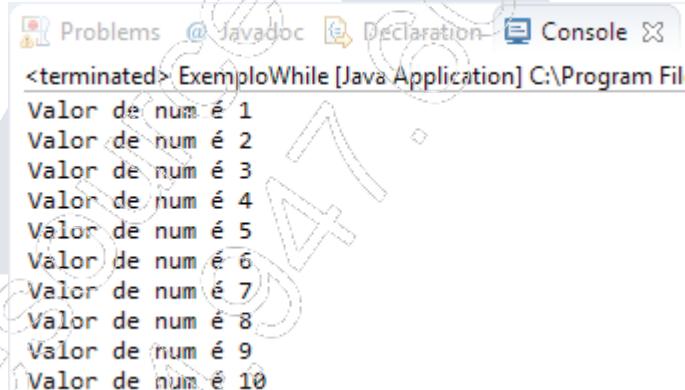


```
J ExemploWhile.java X
1
2 public class ExemploWhile {
3     public static void main(String args[]){
4         int num = 0;
5         while(num < 10){
6             num++;
7             System.out.println("Valor de num é " + num);
8         }
9     }
10 }
```

Em que:

- Inicialização: **int num = 0;;**
- Teste de permanência no laço: **(num < 10);**
- Teste de saída do laço: **num++;**

O resultado é o seguinte:



```
Problems Javadoc Declaration Console X
<terminated> ExemploWhile [Java Application] C:\Program Fil
Valor de num é 1
Valor de num é 2
Valor de num é 3
Valor de num é 4
Valor de num é 5
Valor de num é 6
Valor de num é 7
Valor de num é 8
Valor de num é 9
Valor de num é 10
```

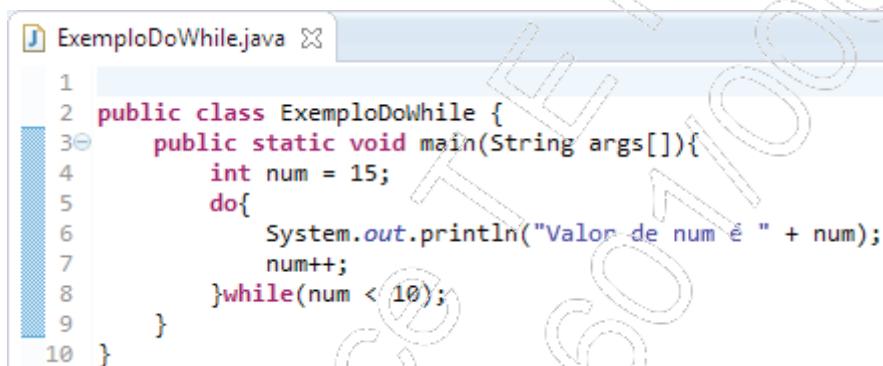
4.3.2. Do / while

O laço **do/while** tem basicamente o mesmo funcionamento de **while**. A diferença é que, aqui, as instruções são executadas antes de a expressão ser avaliada. Assim, garante-se que o bloco de instruções seja executado ao menos uma única vez. Isso acontecerá mesmo se a condição não for verdadeira.

Veja a sintaxe:

```
do {  
    comandos;  
} while(condição);
```

O exemplo a seguir demonstra a aplicação de **do / while**:

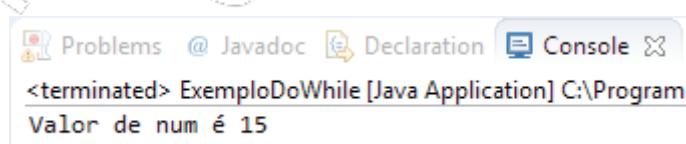


```
ExemploDoWhile.java  
1 public class ExemploDoWhile {  
2     public static void main(String args[]){  
3         int num = 15;  
4         do{  
5             System.out.println("Valor de num é " + num);  
6             num++;  
7         }while(num < 10);  
8     }  
9 }  
10 }
```

Em que:

- Inicialização: **int num = 15;**;
- Teste de permanência no laço: **(num < 10)**;
- Teste de saída do laço: **num++;**

O resultado é o seguinte:



```
Problems @ Javadoc Declaration Console  
<terminated> ExemploDoWhile [Java Application] C:\Program  
Valor de num é 15
```

4.3.3. For

Quando sabemos quantas vezes um bloco de instruções deverá ser executado, utilizamos o laço de repetição **for**. Ele é composto pelo corpo do laço e pelas seguintes partes principais:

- **Declaração e inicialização**

Na primeira parte da instrução **for**, podemos declarar e inicializar uma ou mais variáveis. Elas são colocadas entre parênteses, após a palavra-chave **for** e, se houver mais de uma variável do mesmo tipo, elas serão separadas por vírgulas. Equivale ao momento de inicialização, apontado em cada um dos laços vistos anteriormente. Veja um exemplo:

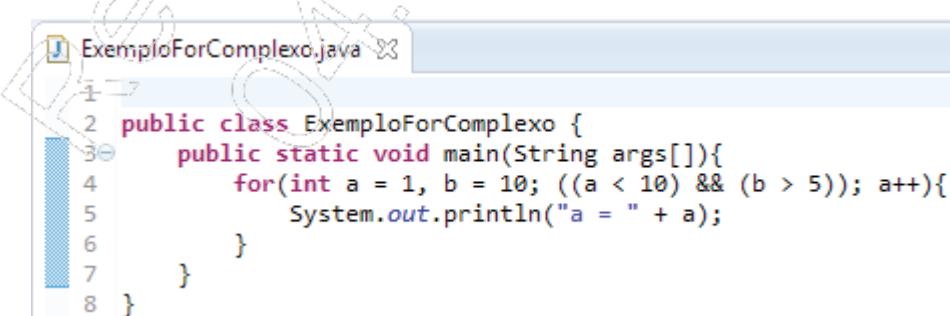
```
for (int a = 1, b = 1; ...) { }
```

A declaração e a inicialização das variáveis na instrução **for** sempre ocorrem antes de outros comandos. Elas acontecem apenas uma vez no laço de repetição, sendo que o teste booleano e a expressão de iteração são executados a cada laço do programa.

- **Expressão condicional**

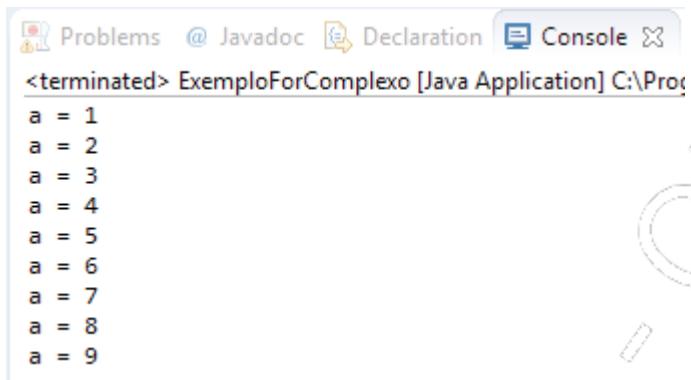
O elemento seguinte é a expressão condicional. Ela se refere a um teste que será executado e deverá retornar um valor booleano. Só podem ser especificadas expressões lógicas, que podem ser complexas. Por isso, é importante ter cuidado com códigos que utilizam diversas expressões lógicas. Equivale ao teste de permanência no laço, apontado em cada um dos laços vistos anteriormente.

Veja um exemplo:



```
ExemploForCompleto.java
1
2 public class ExemploForCompleto {
3     public static void main(String args[]){
4         for(int a = 1, b = 10; ((a < 10) && (b > 5)); a++){
5             System.out.println("a = " + a);
6         }
7     }
8 }
```

E, agora, confira o resultado:



```
a = 1
a = 2
a = 3
a = 4
a = 5
a = 6
a = 7
a = 8
a = 9
```

Neste exemplo, você pode observar que a expressão condicional é válida, mas há casos em que isso não ocorre, como no exemplo a seguir:

```
for (int a = 1, b = 10; (a < 10) , (b > 5); a++)
```

Já neste exemplo, há dois testes booleanos separados por vírgulas, o que não permite que a instrução seja executada e gera um erro. Isso ocorre porque podemos ter apenas uma expressão de teste na sintaxe desse comando.

- **Expressão de iteração**

A expressão de iteração indica o que deverá ocorrer após cada execução do corpo do laço. Ela sempre será processada após a execução do corpo do laço, mas será a última execução da instrução do laço **for**. Equivale ao teste de saída do laço, apontado em cada um dos laços vistos anteriormente, porém, pode haver situações em que esse trecho contenha comandos não relacionados com a saída do laço. Nesse caso, a expressão de saída poderá ser alocada no interior do laço.

Veja, a seguir, a sintaxe completa da instrução **for**, com as partes descritas anteriormente:

```
for (inicialização; condição; iteração){  
    instrução do corpo do laço for;  
}
```

A seguir, um exemplo de uso do laço de repetição **for**:

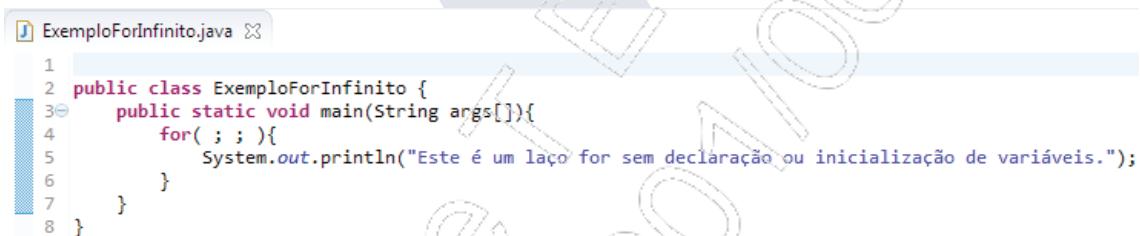
```
for(int a = 1; a < 10; a++){ }
```

Em que:

- **int** refere-se ao tipo de variável que será declarada;
- **a = 1** refere-se à variável que está sendo inicializada;
- **a < 10** refere-se ao teste booleano que será feito na variável **a**;
- **a++** indica a iteração da variável **a**.

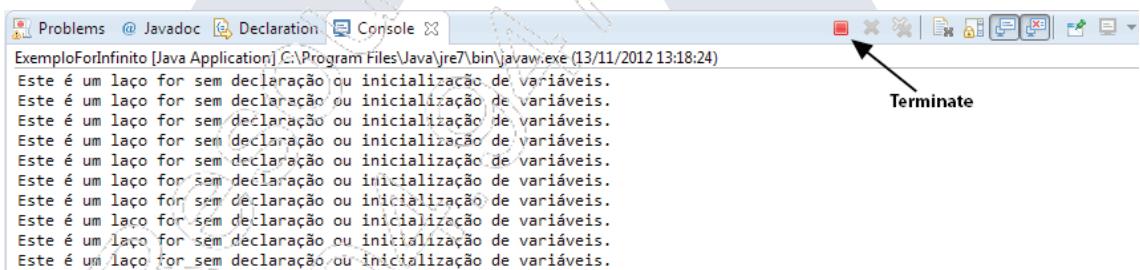
É possível, mas não recomendável, usar um **for** que não possua todas as três partes do laço descritas. Nenhuma das três partes é obrigatória. Neste caso, o laço poderá ser infinito e, não havendo seções de declaração e inicialização, ele agirá como um laço de repetição **while**.

Veja um exemplo:



```
J ExemploForInfinito.java X
1
2 public class ExemploForInfinito {
3     public static void main(String args[]){
4         for( ; ; ){
5             System.out.println("Este é um laço for sem declaração ou inicialização de variáveis.");
6         }
7     }
8 }
```

O **for** infinito executará a instrução até que ela seja interrompida pelo botão **Terminate**, o qual estará ativo nessa ocasião.



```
Problems @ Javadoc Declaration Console X
ExemploForInfinito [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (13/11/2012 13:18:24)
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
Este é um laço for sem declaração ou inicialização de variáveis.
```

Com relação ao escopo das variáveis que foram declaradas no interior de um laço de repetição, qualquer que seja, vale afirmar que ele é finalizado juntamente com o laço. Sendo assim, devemos observar que:

- A variável declarada no interior de um laço pode ser utilizada apenas no próprio laço de repetição;
- A variável declarada fora do laço de repetição, mas inicializada na instrução **for** ou no interior do **while** e **do/while**, pode ser utilizada fora do laço de repetição.

O exemplo a seguir mostra a declaração de uma variável fora da instrução **for**, mas cuja inicialização ocorre dentro da instrução **for**:

```
ExemploDeclaracaoFor.java
1
2 public class ExemploDeclaracaoFor {
3     public static void main(String args[]){
4         int a;
5         for(a = 9; a < 15; a++){
6             System.out.println("a = " + a);
7         }
8     }
9 }
```

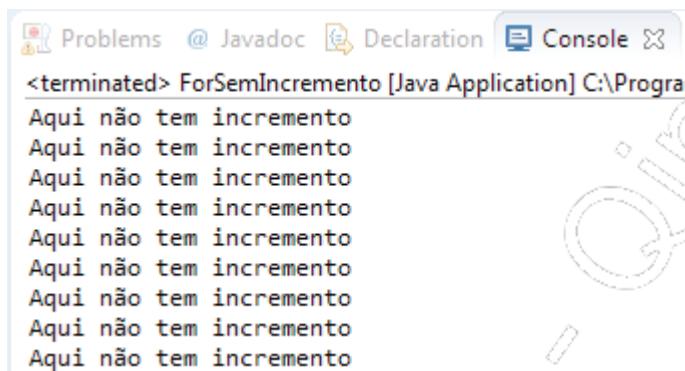
O resultado é o seguinte:

```
Problems @ Javadoc Declaration Console
<terminated> ExemploDeclaracaoFor [Java Application] C:\Pro
a = 9
a = 10
a = 11
a = 12
a = 13
a = 14
```

Vemos, assim, que as seções da instrução **for** não são dependentes e, por isso, não precisam operar sobre as mesmas variáveis. Quanto à expressão de iteração, ela não precisa, necessariamente, configurar ou incrementar algo, como no exemplo a seguir:

```
ForSemIncremento.java
1
2 public class ForSemIncremento {
3     public static void main(String args[]){
4         for(int x = 10, y = 1; x != 1; System.out.println("Aqui não tem incremento")){
5             x = x - y;
6         }
7     }
8 }
```

O resultado da compilação e execução desse código é o seguinte:



```
<terminated> ForSemIncremento [Java Application] C:\Progra
Aqui não tem incremento
```

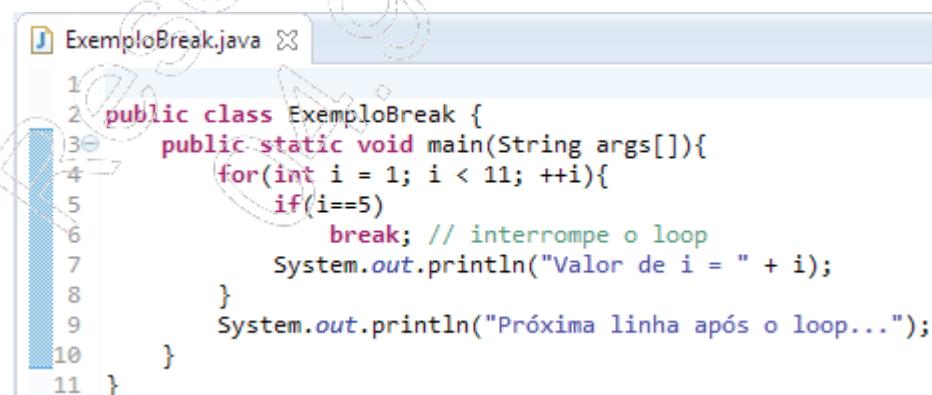
4.4. Outros comandos

Quando trabalhamos com as instruções vistas até aqui, podemos, em alguns casos, utilizar comandos para auxiliar o controle do fluxo de execução. Abordamos, adiante, os comandos **break** e **continue**.

4.4.1. Break

O comando **break** pode ser utilizado com laços de repetição **while**, **do/while**, **for** e estruturas **switch**. Quando utilizado em um laço de repetição, causa sua interrupção imediata, continuando a execução do programa na próxima linha **após o laço**. É comumente utilizado dentro de um bloco de desvio condicional no laço, de forma que a interrupção ocorra caso alguma condição seja atendida.

Veja o exemplo a seguir:



```
1  public class ExemploBreak {
2      public static void main(String args[]){
3          for(int i = 1; i < 11; ++i){
4              if(i==5)
5                  break; // interrompe o loop
6              System.out.println("Valor de i = " + i);
7          }
8          System.out.println("Próxima linha após o loop...");
9      }
10 }
11 }
```

O resultado é o seguinte:



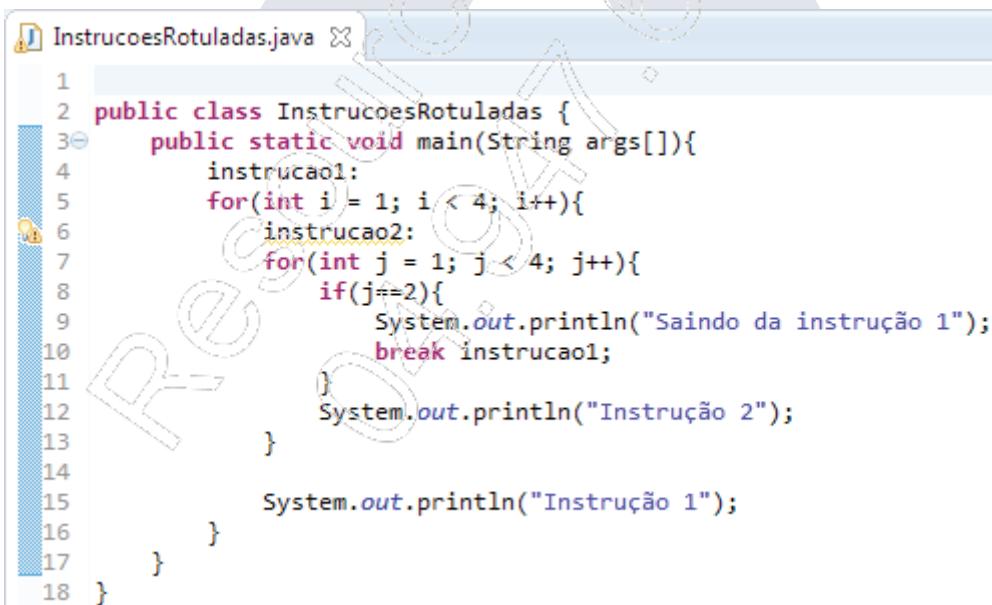
```
Problems @ Javadoc Declaration Console
<terminated> ExemploBreak [Java Application] C:\Program Fil
Valor de i = 1
Valor de i = 2
Valor de i = 3
Valor de i = 4
Próxima linha após o loop...
```

4.4.1.1. Instruções rotuladas

Utilizamos as instruções rotuladas quando temos laços de repetição aninhados e precisamos indicar qual deles deve ser finalizado, ou quando queremos indicar a partir de qual laço a próxima iteração deverá continuar.

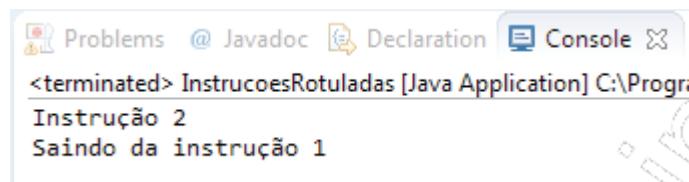
O rótulo atribuído à instrução deve seguir as regras de nomes de variável válidos para a linguagem Java. Além disso, devemos verificar se ele segue as convenções de nomeação para Java.

Uma instrução **break** rotulada indica que o programa deve executar fora do laço de repetição rotulado, mas não do laço de repetição atual. Veja no exemplo a seguir:



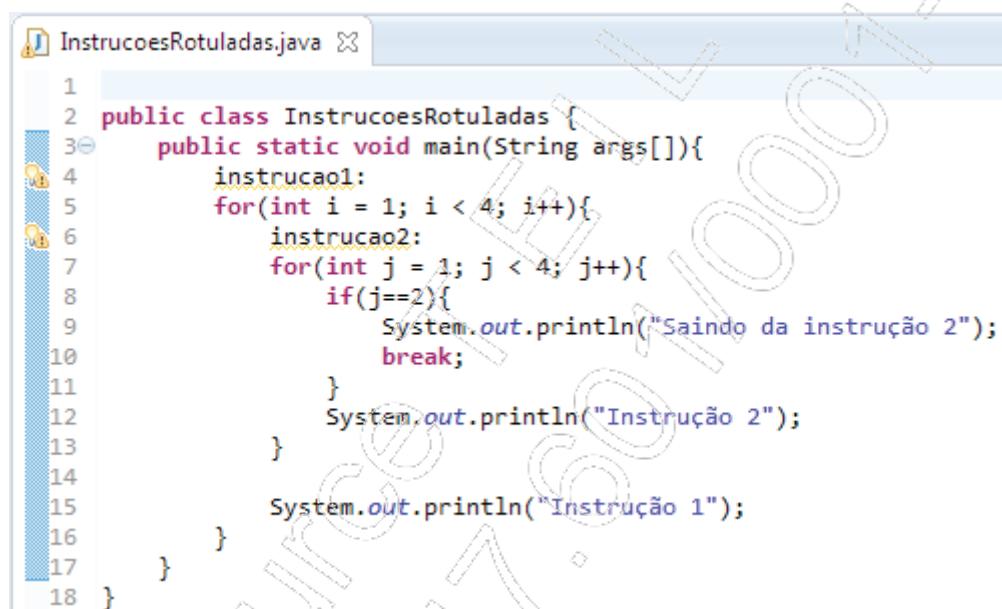
```
InstrucoesRotuladas.java
1
2 public class InstrucoesRotuladas {
3     public static void main(String args[]){
4         instrucao1:
5             for(int i = 1; i < 4; i++){
6                 instrucao2:
7                     for(int j = 1; j < 4; j++){
8                         if(j==2){
9                             System.out.println("Saindo da instrução 1");
10                            break instrucao1;
11                         }
12                         System.out.println("Instrução 2");
13                     }
14                     System.out.println("Instrução 1");
15                 }
16             }
17     }
18 }
```

O resultado será o seguinte:



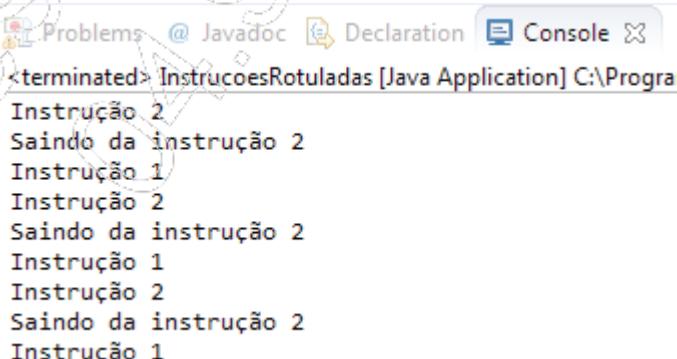
```
Problems @ Javadoc Declaration Console ×
<terminated> InstrucoesRotuladas [Java Application] C:\Program Files\Java\Java SE\1.8.0_101\jre\bin\javaw.exe
Instrução 2
Saindo da instrução 1
```

Quando uma instrução **break** não for rotulada, a estrutura do laço de repetição atual será finalizada e o programa continuará a ser executado a partir da próxima linha de código após o bloco do laço, como no seguinte exemplo:



```
InstrucoesRotuladas.java
1
2 public class InstrucoesRotuladas {
3     public static void main(String args[]){
4         instrucao1:
5             for(int i = 1; i < 4; i++){
6                 instrucao2:
7                     for(int j = 1; j < 4; j++){
8                         if(j==2){
9                             System.out.println("Saindo da instrução 2");
10                            break;
11                         }
12                         System.out.println("Instrução 2");
13                     }
14                     System.out.println("Instrução 1");
15                 }
16             }
17         }
18     }
```

O resultado será o seguinte:



```
Problems @ Javadoc Declaration Console ×
<terminated> InstrucoesRotuladas [Java Application] C:\Program Files\Java\Java SE\1.8.0_101\jre\bin\javaw.exe
Instrução 2
Saindo da instrução 2
Instrução 1
Instrução 2
Saindo da instrução 2
Instrução 1
Instrução 2
Saindo da instrução 2
Instrução 1
```

4.4.2. Continue

O comando **continue** pode ser utilizado somente em laços de repetição. Quando ele é executado, o laço volta imediatamente para o teste de condição. Diferentemente do comando **break**, o **continue** causa a saída da iteração atual do laço, dirigindo-se imediatamente à iteração seguinte.

O exemplo a seguir mostra a utilização do comando **continue** com o laço de repetição **while**:

```
1 public class ExemploContinue {
2     public static void main(String args[]){
3         int num = 0;
4         while(num<10){
5             ++num;
6             if(num==5){
7                 continue;
8             }
9             System.out.println("Valor de num = " + num);
10        }
11    }
12 }
```

O resultado é o seguinte:

```
<terminated> ExemploContinue [Java Application] C:\Program  
Valor de num = 1  
Valor de num = 2  
Valor de num = 3  
Valor de num = 4  
Valor de num = 6  
Valor de num = 7  
Valor de num = 8  
Valor de num = 9  
Valor de num = 10
```

Ao usar o comando `continue`, é preciso considerar os efeitos que ele pode causar sobre a iteração do laço de repetição. O `continue` também pode ser aplicado nas instruções de repetição rotuladas.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

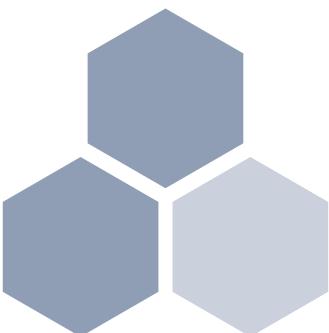
- As estruturas de desvio condicional permitem desviar o fluxo de execução de um programa de acordo com determinadas condições;
- Uma instrução **if** é uma estrutura condicional simples e serve para avaliar expressões com resultados booleanos. A estrutura **if / else** é uma estrutura condicional dupla, em que há uma instrução a ser executada caso a condição seja verdadeira e uma outra instrução a ser executada caso a condição seja falsa;
- A estrutura **switch** é uma estrutura de controle com múltipla escolha e serve para situações em que há uma sequência muito longa de desvios condicionais;
- As estruturas ou laços de repetição servem para situações em que é necessário repetir a execução de um bloco de códigos do programa até que uma determinada condição seja verdadeira, ou até que o bloco seja executado uma determinada quantidade de vezes. Em toda estrutura de repetição, é preciso ficar atento à inclusão dos três requisitos fundamentais: inicialização, teste de permanência no laço e teste de saída do laço;
- A estrutura **while** é usada quando não sabemos quantas vezes um determinado bloco de instruções deve ser executado. Com ela, a execução das instruções continua enquanto uma condição for verdadeira;
- O laço **do/while** tem basicamente o mesmo funcionamento do **while**, com a diferença de que, aqui, as instruções são executadas antes de a expressão ser avaliada, de modo que o bloco de instruções sempre será executado pelo menos uma vez;
- O laço de repetição **for** deve ser usado quando sabemos quantas vezes um bloco de instruções deverá ser executado;
- O comando **break** pode ser utilizado com laços de repetição **while**, **do/while**, **for** e estruturas **switch**. Em um laço de repetição, ele causa sua interrupção imediata, continuando a execução do programa na próxima linha após o laço;
- O comando **continue** só pode ser utilizado em laços de repetição. Quando ele é executado, o laço volta imediatamente para o teste de condição, pulando a iteração atual.



4

Estruturas de controle

Teste seus conhecimentos



1. Quais estruturas de desvios condicionais estão disponíveis em Java?

- a) do/while, for
- b) if/else, switch
- c) if/else, do/while
- d) do/while, switch/case
- e) Nenhuma das alternativas anteriores está correta.

2. Qual estrutura de repetição executa, pelo menos uma vez, o bloco de instruções?

- a) while
- b) for
- c) do/while
- d) if/else
- e) Nenhuma das alternativas anteriores está correta.

3. Quais são as partes principais do laço de repetição for?

- a) Expressão de inicialização, expressão de iteração e expressão de finalização.
- b) Declaração e inicialização de variáveis, expressão condicional e expressão de iteração.
- c) Declaração de variáveis, expressão de finalização e expressão condicional.
- d) Expressão condicional, declaração de variáveis e verificação lógica.
- e) Nenhuma das alternativas anteriores está correta.

4. Quantos números serão exibidos quando o código a seguir for executado?

```
for (int i = 5; i <= 10; i++) {  
    System.out.println(i);  
}
```

- a) 5
- b) 6
- c) 10
- d) 15
- e) 50

5. Qual é a função do comando break?

- a) Interromper o programa.
- b) Interromper uma variável.
- c) Interromper um laço de repetição.
- d) Interromper todas as ações do programa.
- e) Nenhuma das alternativas anteriores está correta.

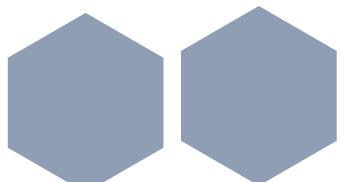


4

Estruturas de controle



Mãos à obra!



Laboratório 1

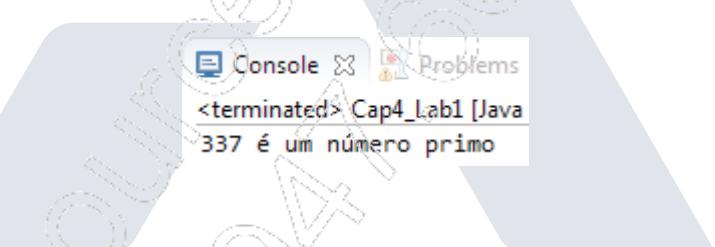
A – Verificando se um número é primo

Vamos criar um programa que verifica se um número é primo. Lembre-se de que um número primo só é divisível por 1 e por ele mesmo.

1. Crie uma classe e insira a estrutura básica de um programa Java;
2. Declare uma variável chamada **numero**, que receberá o valor a ser verificado se é primo ou não;
3. Implemente uma lógica onde será contado o número de divisores do número informado;
4. Ao final da contagem, se o número de divisores encontrados for superior a dois, o número informado não é primo. Caso contrário, é primo;

 Lembre-se que 0 (zero) e 1 (um) não são considerados números primos.

5. Compile e execute o programa:



```
Console X Problems
<terminated> Cap4_Lab1 [Java]
337 é um número primo
```

Laboratório 2

A – Verificando quantos dias tem cada mês

Vamos criar um programa que imprime na tela a quantidade de dias existentes no mês, dada uma variável que contenha o nome do mês em questão.

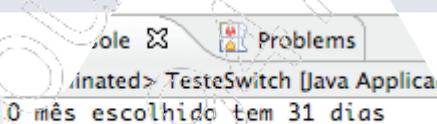
1. Crie uma classe e insira a estrutura básica de um programa Java;
2. Declare uma variável do tipo **String** e armazene o nome de um mês de sua preferência, conforme o exemplo a seguir:

```
String mes = "Novembro";
```

3. Utilize uma estrutura de decisão múltipla **switch**, tendo a variável **mes** como referência, e imprima, para o mês escolhido, a frase: “O mês **escolhido** tem **XX** dias”. Utilize strings iniciadas com maiúsculo ou minúsculo (padronize para evitar problemas!);

4. Acrescente um item **default** à estrutura **switch** com uma mensagem de erro que será impressa no console, caso algum mês inexistente seja atribuído à variável **mes**;

5. Altere os nomes atribuídos à variável **mes**, compile e execute seu programa algumas vezes:



Laboratório 3

A - Exibindo os anos de ocorrência de todas as copas do mundo de futebol

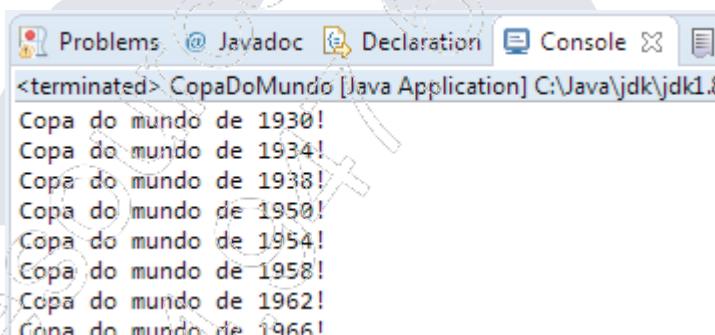
Vamos criar um programa que imprime na tela todos os anos em que houve Copa do Mundo de Futebol.

1. Crie uma classe e insira a estrutura básica de um programa Java;
2. Utilize a estrutura **for**, declarando a variável inteira **ano** e inicializando com o valor **1930** (ano da primeira copa do mundo). Esta variável deverá ser incrementada de 4 em 4 (período entre as copas) até atingir o ano atual;
3. Considere o caso dos anos de **1942** e **1946** que foram anos em que não ocorreram copas devido à segunda guerra mundial;

 Dica: Utilize uma das opções de **break/continue**.

4. Imprima cada um dos anos de copa do mundo.

O resultado deverá ser semelhante a este:



```
Problems Javadoc Declaration Console <terminated> CopaDoMundo [Java Application] C:\Java\jdk\jdk1.8
Copa do mundo de 1930!
Copa do mundo de 1934!
Copa do mundo de 1938!
Copa do mundo de 1950!
Copa do mundo de 1954!
Copa do mundo de 1958!
Copa do mundo de 1962!
Copa do mundo de 1966!
```

5

Introdução à orientação a objetos

- ◆ Classes;
- ◆ Objeto;
- ◆ Atributos;
- ◆ Tipos construídos;
- ◆ Encapsulamento;
- ◆ Pacotes;
- ◆ UML – Diagramas de casos de uso,
classes e pacotes.

5.1. Apresentação

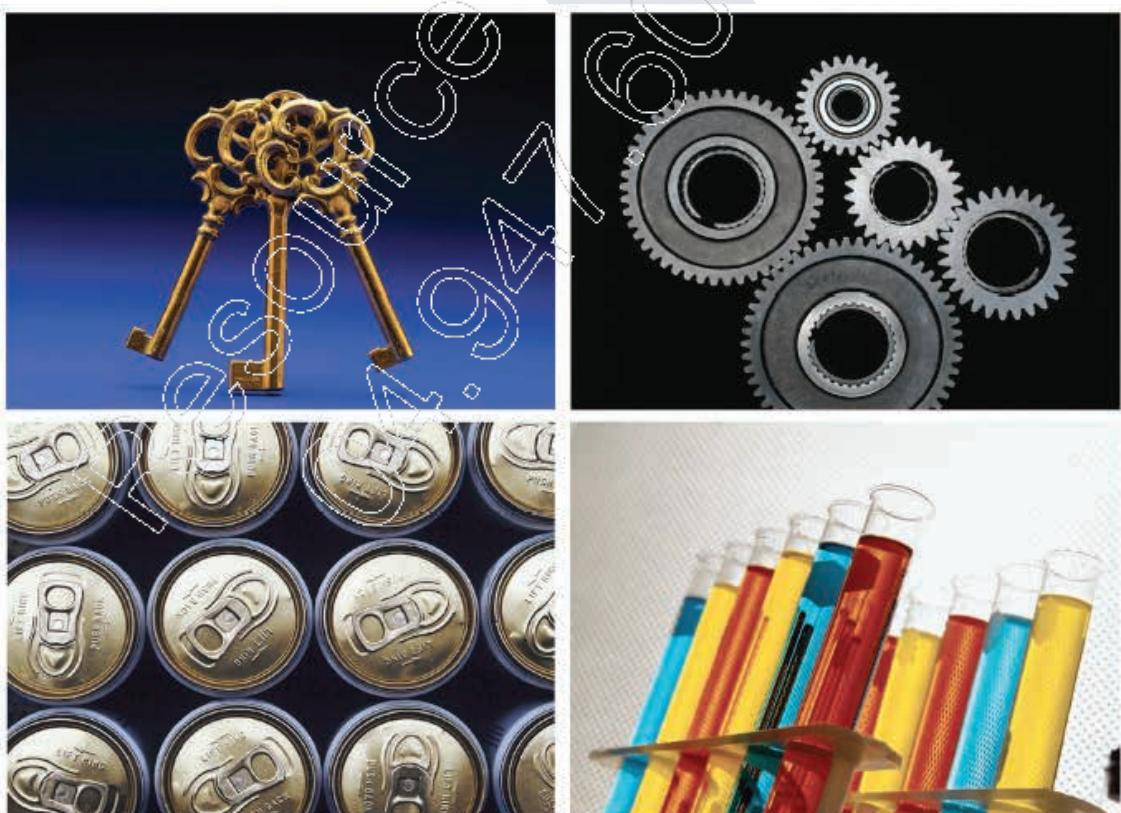
Nos capítulos anteriores, já vimos algumas vezes termos como classe, objeto e método. São conceitos relativos à programação orientada a objetos, que, como você já sabe, é o paradigma da linguagem Java.

Neste capítulo, você conhecerá esses e outros conceitos fundamentais da orientação a objetos, indispensáveis para uma boa compreensão da estrutura e funcionamento da linguagem Java.

5.2. Classes

Classe é um conceito central na programação orientada a objetos. Com uma classe, representamos uma categoria de elementos do mundo real.

Podemos definir uma classe, genericamente, como um modelo que representa um conjunto de elementos (pessoas, objetos, animais etc.) com características comuns. Em programação, mais especificamente, uma classe pode ser entendida como um modelo definido para um dado conjunto de objetos que possuem os mesmos **atributos** e **métodos**.



Uma classe é definida a partir do processo de abstração, que faz a identificação de um objeto de acordo com o que ele é (conjunto de atributos – estado) e com o que ele faz (conjunto de métodos – comportamento), sem levar em conta outras características.

Chamamos de classe concreta aquela que tem objetos instanciados diretamente a partir dela. Mais à frente neste capítulo, apresentaremos o conceito de instanciação.

5.3. Objeto

Um objeto pode ser entendido como uma representação do mundo real ou, mais precisamente, qualquer elemento do mundo ao qual é possível atribuir certas características e comportamentos. Transferindo isso para o campo computacional, podemos dizer que os objetos são elementos que representam uma entidade, abstrata ou concreta, no domínio de interesse do problema analisado.

A programação orientada a objetos tem como característica fundamental usar conceitos do mundo real para processar dados em um programa e, assim, dar uma base prática a uma solução computacional. Os objetos são o principal elemento desse tipo de programação.

Quando um programador formula uma solução computacional para problemas do mundo real, ele está decompondo esses problemas em objetos. Para fazer essa abstração, não existe uma maneira correta ou única, há diversas possibilidades. Tudo depende da natureza de cada problema específico e das escolhas de quem está projetando a solução. Por isso, o trabalho com objetos é bastante amplo.

Todo objeto possui identidade própria, podendo ser distinguido de outros objetos. Um objeto criado é alocado em certo espaço na memória, onde são armazenados seu estado e o conjunto de operações que podem ser aplicadas a ele, conjunto esse também chamado de comportamento. Em outras palavras, o conjunto de atributos é o conjunto de métodos do objeto.

Nas linguagens orientadas a objetos, um objeto pertence a um determinado grupo com características e comportamentos comuns, ou seja, todo objeto é elemento pertencente a uma classe.

5.3.1. Instanciação

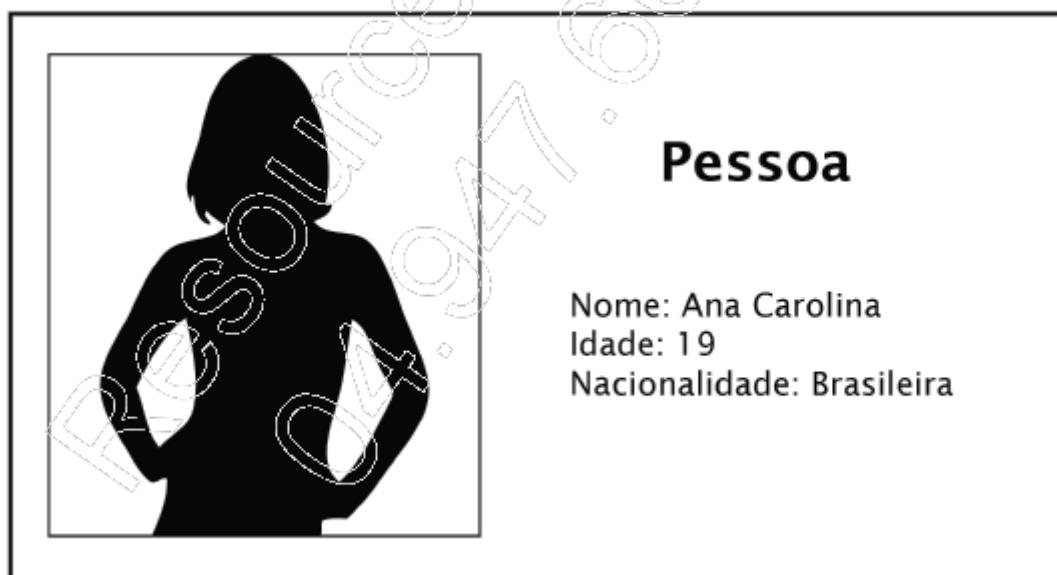
Um objeto é sempre criado a partir de uma classe. Isso quer dizer que um objeto não é criado diretamente, mas sempre a partir da definição, na classe, dos atributos e métodos que o compõem.

A criação de um objeto é chamada instanciação. Na programação orientada a objetos, dizemos que todo objeto é instância de uma classe. Instanciar um objeto representa a alocação de uma área de memória que comporte a estrutura de dados definida na classe, para ser usada no programa.

5.4. Atributos

Os atributos podem ser definidos como características específicas de um objeto. Podemos ampliar um pouco essa definição e compreendê-los também como variáveis ou campos que armazenam valores referentes a diferentes características de um objeto.

No exemplo a seguir, temos um objeto **Pessoa** que possui os atributos **Nome**, **Idade** e **Nacionalidade**. Cada um desses atributos é expresso por determinados valores.



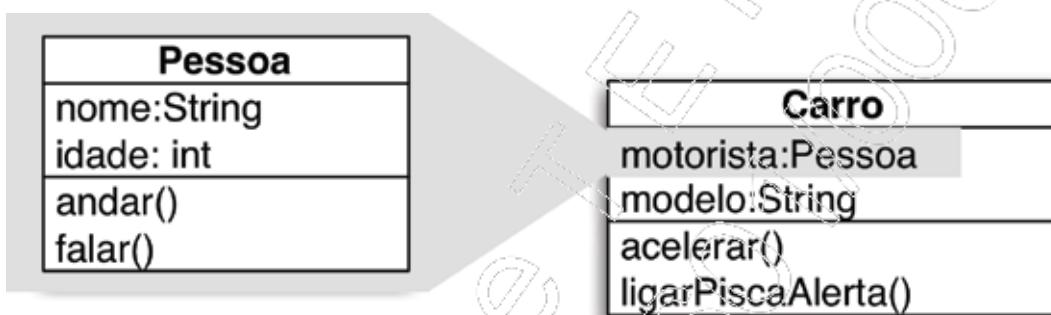
Dentro de uma classe de objetos, os nomes de cada atributo devem ser exclusivos, mas em duas classes diferentes pode haver atributos com o mesmo nome.

Os valores dos atributos são alterados por meio de ações internas ou externas. Para isso, é necessário disparar eventos.

É importante destacar que cada objeto é responsável pelo controle da alteração de seus próprios atributos e não deve interferir nos atributos de outro objeto, salvo em caso de solicitação de serviços. Ou seja, um objeto deve ser projetado para ser independente em relação aos outros.

5.5. Tipos construídos

As variáveis em um programa podem ser criadas a partir de tipos construídos, ou seja, a partir de classes já existentes. Uma variável de tipo construído está diretamente relacionada a um tipo de classe criada. Veja, emblematicamente, uma classe que possui como variável um objeto provindo de um tipo construído:



Na figura anterior, o atributo **motorista** da classe **Carro** é do tipo construído (classe) **Pessoa**, ou seja, a classe **Carro** se compõe, dentre outros atributos, da classe **Pessoa**.

O processo de criação de variáveis por meio de tipos construídos recebe o nome de **instanciação**. Este é um processo diferente da criação de variáveis de tipo primitivo (**byte**, **short**, **char**, **int**, **long**, **float**, **double**, **boolean**).

5.5.1. Atribuição entre objetos de tipos construídos

Quando atribuímos variáveis de tipos construídos, o objeto que recebe essa atribuição tem seu valor ou valores perdidos, passando a referenciar o objeto ao qual foi atribuído. Os dois objetos, então, passam a fazer referência ao mesmo objeto que existe na memória. Quando um objeto não é referenciado por nenhuma variável, ele está pronto para ser coletado pelo Garbage Collector.

Os exemplos descritos a seguir mostram o processo de atribuição entre tipos construídos:

- **Exemplo 1**

```
Pessoa.java
1
2 public class Exemplo1Atribuicao {
3     public static void main(String args[]){
4
5         Pessoa joao = new Pessoa();
6         Pessoa maria = new Pessoa();
7
8         joao.sexo = 'M';
9         joao.idade = 45;
10
11        maria.sexo = 'F';
12        maria.idade = 17;
13
14        joao = maria; // atribuição realizada
15        // as variáveis joao e maria fazem referência ao mesmo objeto
16        System.out.println("João sexo = " + joao.sexo);
17        System.out.println("João idade = " + joao.idade);
18    }
19 }
```

Depois de compilado e executado o código, o resultado será o seguinte:

```
Problems @ Javadoc Declaration Console
<terminated> Exemplo1Atribuicao [Java Application] C:\Program
João sexo = F
João idade = 17
```

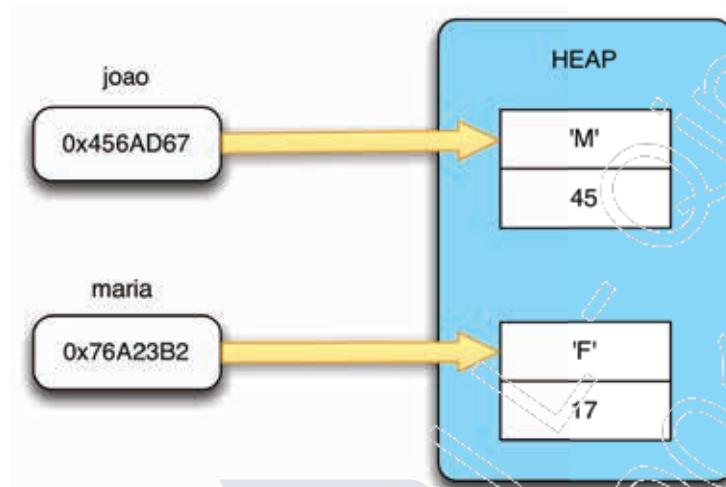
- Exemplo 2

```
1  public class Pessoa {
2      public static void main(String args[]){
3          Pessoa joao = new Pessoa();
4          Pessoa maria = new Pessoa();
5
6          joao.sexo = 'M';
7          joao.idade = 45;
8
9          maria.sexo = 'F';
10         maria.idade = 17;
11
12         joao = maria; // atribuição realizada
13         // as variáveis joao e maria fazem referência ao mesmo objeto
14         System.out.println("João sexo = " + joao.sexo);
15         System.out.println("João idade = " + joao.idade);
16         // qualquer alteração efetuada no objeto refletirá nas duas variáveis
17         joao.idade = 50;
18         System.out.println("Maria idade = " + maria.idade);
19
20         joao.sexo = 'M';
21         System.out.println("Maria sexo = " + maria.sexo);
22     }
23 }
24 }
```

Após a compilação e execução do código, o resultado será o seguinte:

```
<terminated> Exemplo2Atribuicao [Java Application] C:\Program
João sexo = F
João idade = 17
Maria idade = 50
Maria sexo = M
```

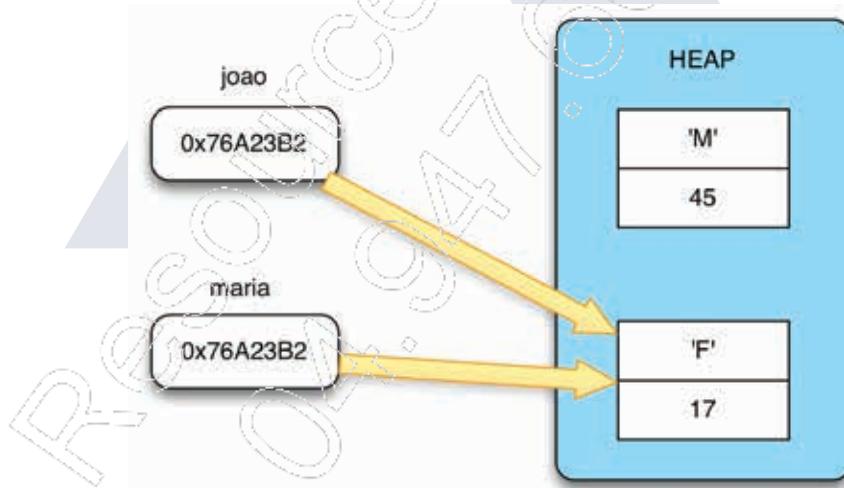
Esquematicamente, veja o que ocorre com a declaração e instanciação dos objetos **joao** e **maria**, ambos do tipo **Pessoa**:



Após a atribuição realizada, temos:

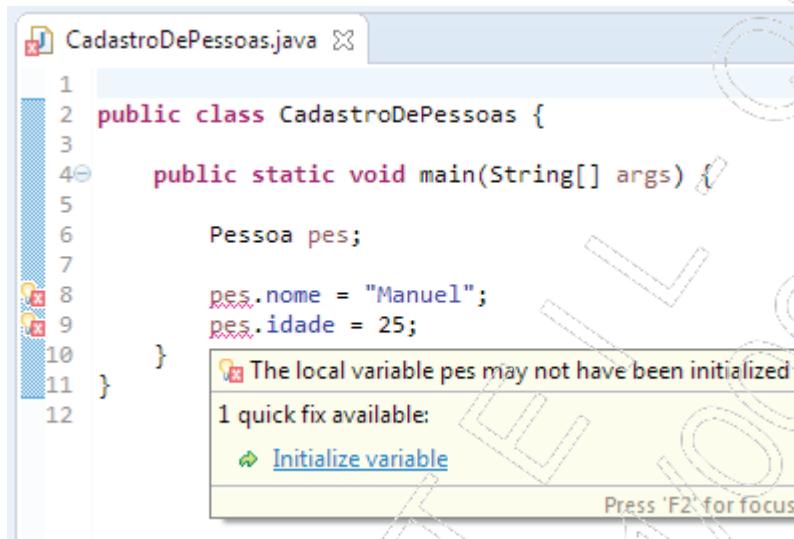
```
joao = maria;
```

Veja, agora, o que ocorre com os objetos na memória:



5.5.2. Variáveis não inicializadas

Variáveis locais (criadas dentro de funções ou métodos) são facilmente detectadas pelo Java quando não são inicializadas (instanciadas) antes de sua utilização. Veja:



A screenshot of an IDE showing a Java code editor. The file is named 'CadastroDePessoas.java'. The code contains a class definition with a main method. In the main method, there is a local variable 'Pessoa pes' declared on line 6. On line 8, 'pes.nome' is assigned the value 'Manuel', and on line 9, 'pes.idade' is assigned the value 25. A tooltip window appears over the assignment on line 8, indicating a warning: 'The local variable pes may not have been initialized' with a quick fix option 'Initialize variable'. The tooltip also says 'Press 'F2' for focus'.

```
1 public class CadastroDePessoas {
2     public static void main(String[] args) {
3         Pessoa pes;
4         pes.nome = "Manuel";
5         pes.idade = 25;
6     }
7 }
```

As variáveis de classe (também chamadas de **atributos**) são aquelas declaradas no corpo da classe, ou seja, são declaradas fora de qualquer método.

Quando declaramos um atributo em uma classe e não o inicializamos explicitamente, em tempo de execução, a máquina virtual o inicializa com valores padrão para cada tipo:

- Numéricos e caracteres (**byte, short, char, int, long, float, double**): **0 (Zero)**;
- Booleanos: **false**;
- Tipos construídos: **null**.

5.5.3. O uso do **this**

Quando for preciso fazer referência a um objeto criado dentro da própria classe, ou ainda, quando for preciso referenciar o próprio objeto onde se encontra o método, você deve utilizar a palavra **this**. Veja as duas situações que, a princípio, exigem a utilização desta palavra:

- Quando uma referência do próprio objeto onde o método se encontra tem que ser passada a um método qualquer;
- Quando uma classe possui uma variável de instância e uma ou mais variáveis locais com o mesmo nome e faz-se necessário referenciá-las dentro do método que declarou aquela variável local. Essa ocorrência é denominada de sombreamento de campo, dado que o compilador sempre procurará referenciar o objeto mais próximo do escopo onde é chamado.

Nesta última situação, você deve utilizar a palavra **this** apenas se o método ao qual a variável pertence precisar referenciar um atributo da classe.

A partir destas informações, veja o exemplo a seguir:

```
1 public class Empregado {  
2     private String endereco; Variável  
3     private int idade;  
4  
5     void setEndereco(String endereco) {  
6         this.endereco = endereco; Argumento  
7     }  
8  
9     void setIdade(int idade) {  
10        this.idade = idade;  
11    }  
12  
13    String mostrar(){  
14        return (endereco + idade);  
15    }  
16}  
17
```

```
1  
2 public class TesteEmpregado {  
3     public static void main(String args[]){  
4         Empregado emp = new Empregado();  
5         emp.setEndereco("Centro");  
6         emp.setIdade(40);  
7  
8         System.out.println(emp.mostrar());  
9     }  
10 }
```

Nesse exemplo, a palavra **this** refere-se aos atributos definidos na classe e não aos parâmetros dos métodos. Observando o método **mostrar()**, verifica-se que não foi necessário utilizar **this**. Isso porque, quando o atributo está claro e os parâmetros não possuem nomes iguais, implicitamente, qualquer referência a nome ou endereço assume o atributo da classe.

5.6. Encapsulamento

O encapsulamento é um mecanismo que permite ocultar os detalhes da implementação interna de uma classe, restringindo o acesso a suas variáveis e métodos. O encapsulamento permite que você utilize uma classe sem conhecer seu mecanismo interno de implementação. Isso otimiza a legibilidade do código, ajudando a minimizar os erros de programação, além de facilitar a ampliação do código com novas atualizações, já que o proprietário de uma classe pode modificá-la internamente sem que o usuário precise saber disso.

Para que o mecanismo de encapsulamento seja possível, é fundamental que o desenvolvedor Java tenha em mãos a possibilidade de restringir ou garantir o acesso a determinados objetos e seus membros (atributos e métodos) para outros objetos. Isso é possível por meio da utilização de modificadores ou qualificadores de acesso.

O nível de acesso aos elementos de uma classe é definido pelos seguintes qualificadores:

- **public**: Não possui restrições. Desse modo, uma variável pública não é considerada encapsulada;
- **protected**: As variáveis e métodos podem ser acessados pela sua própria classe e subclasses (não se preocupe, em breve você entenderá o que são subclasses);
- **default**: Uma classe só pode ser acessada por classes do mesmo pacote (em breve você também saberá o que são pacotes);
- **private**: Somente a própria classe pode acessar variáveis e métodos. É o nível mais restrito de encapsulamento.

5.7. Pacotes

Um pacote (ou package) consiste, no sistema de arquivos do computador, em um diretório que armazena um conjunto de classes que, geralmente, possuem características ou uma finalidade em comum. Toda classe pertence a algum pacote. Quando não se especifica um pacote para uma classe, ela se torna parte do pacote default, ou seja, o diretório raiz para o código-fonte do seu programa.

Na tabela a seguir, você encontra alguns dos principais pacotes de Java:

Pacote	Classes do pacote
java.applet	Contém as classes para uso de programa em applets.
java.awt	Classes para criação de interface gráfica.
java.nio	Este pacote contém classes voltadas à atividade de entrada e saída de dados para acesso a recursos externos ao programa, como arquivos, rede etc.
java.lang	Classes fundamentais para desenvolvimento de programas em Java.
java.security	Classes e interfaces de segurança.
java.sql	Contém as APIs para acessar e processar ações de armazenamento de banco de dados controlados por meio de Java.
java.text	Classes e interfaces para tratamento de textos, datas e números.

Para definir um pacote, você deve inserir, na primeira linha de uma classe (a linha de definição da classe), o comando **package**, conforme a seguinte sintaxe:

```
package <nome do pacote>;
```

Um pacote deve ser nomeado com letras minúsculas e pode ser separado das outras partes do endereço por ponto final (.). Recomenda-se que os pacotes tenham o nome reverso ao do site onde estão armazenados. Veja no exemplo:

```
ExemploPackage.java
1 package com.exemplo;
```

Se a classe sendo criada estiver em um pacote, torna-se obrigatória a declaração do **package** no código-fonte da classe. Além disso, essa declaração deve ser a primeira linha de comando do arquivo que contém a classe.

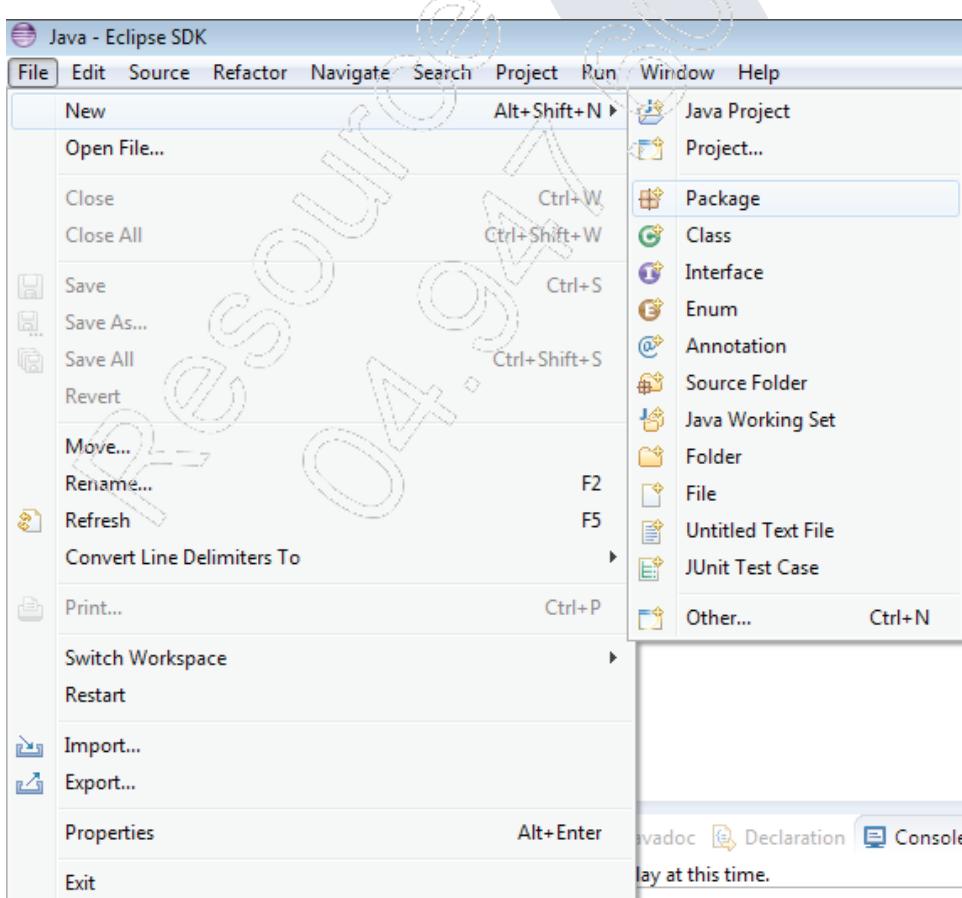
Pacotes representam namespaces em Java. Um namespace é um contexto de nomeação que pode ser usado para prover segurança e encapsulamento às suas classes. Não é possível ter duas classes com o nome igual no mesmo namespace. Logo, em pacotes diferentes, é possível criar classes de mesmo nome. Dessa forma, cada classe passa a ser identificada pelo seu nome completamente qualificado, composto do endereço completo da classe, formado pelo encadeamento de seus pacotes separados por ponto. No exemplo anterior, a classe declarada tem o seguinte nome qualificado:

`com.exemplo.ExemploPackage.`

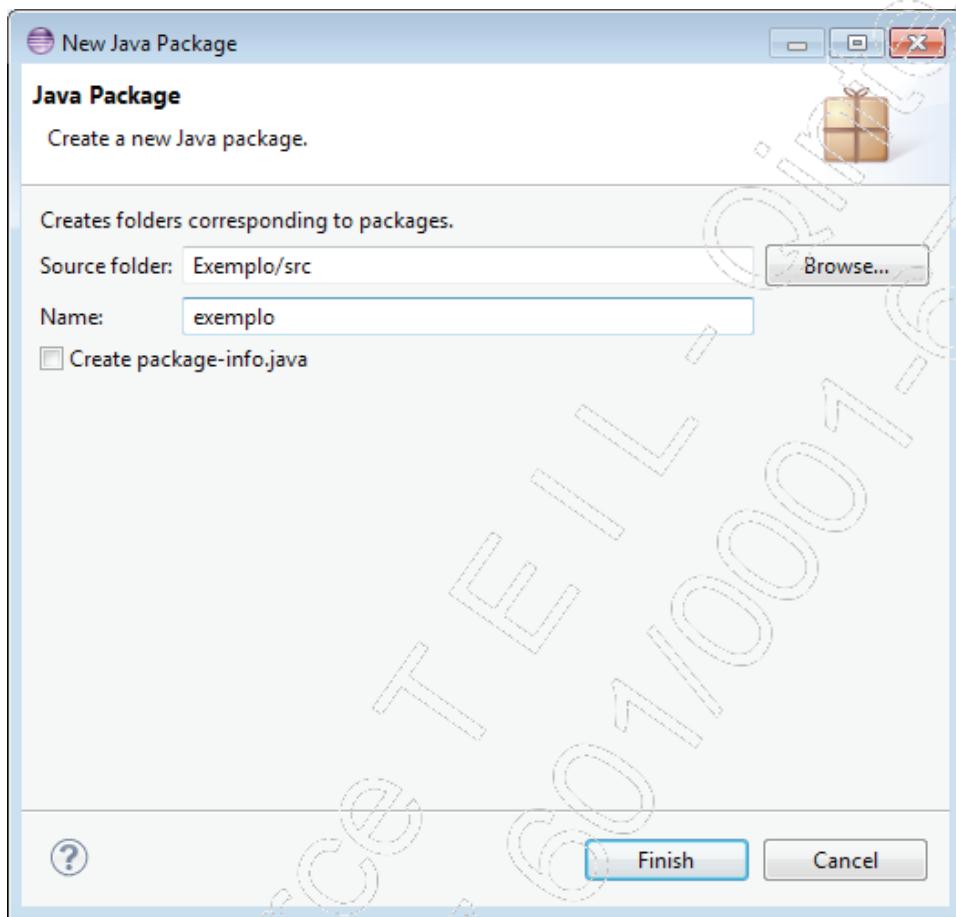
5.7.1. Criando um pacote

Veja o procedimento que você deve executar para criar um pacote no Eclipse:

1. Acesse **File / New / Package** para que a janela **New Java Package** seja exibida:



2. Defina a pasta do projeto no qual será criado o pacote e um nome:



3. Clique em **Finish**. O pacote será criado no subdiretório **src** do projeto.

5.7.2. Acessando uma classe em outro pacote

Com a definição de pacotes, as classes passam a se localizar em namespaces diferentes (pacotes diferentes) e é preciso analisar o procedimento necessário para importá-las para a classe em que se deseja ter acesso.

Quando for necessário utilizar uma classe que está em outro pacote, deve-se importar esse pacote no início da definição da classe. Para isso, basta usar o comando **import** e o nome do pacote, ou melhor, o nome completamente qualificado da classe. Observe, no exemplo a seguir, como utilizar esse comando:

```
Circulo.java
1 package grafico;
2
3 public class Circulo {
4     public void mostrar(){
5         System.out.println("Método executado da classe Circulo do pacote gráfico");
6     }
7 }
```

```
ExemploPackage.java
1 import grafico.Circulo;
2
3 public class ExemploPackage {
4     public static void main(String args[]){
5         Circulo c = new Circulo();
6         c.mostrar();
7     }
8 }
```

Observe que as duas classes estão em pacotes diferentes. No exemplo, utilizamos **public** para a classe **Circulo**. Dessa forma, determinamos que esse membro poderá ser referenciado por outras classes que não estejam no mesmo pacote.

5.8. UML – Diagramas de casos de uso, classes e pacotes

Como a programação orientada a objetos lida com representações abstratas do mundo real, são utilizados modelos de representação (ou notação) gráfica dos objetos. São ferramentas que auxiliam em todo o ciclo de desenvolvimento, tornando mais objetivas as implementações em linguagens orientadas a objetos.

Você verá aqui as notações gráficas da linguagem UML (Unified Modeling Language), um padrão internacional para desenho e modelagem de software orientado a objetos (OO). A UML é largamente utilizada em empresas e instituições como linguagem técnica de representação para software OO, além de ser a forma universal de representar ideias, sugestões, padrões e frameworks criados sob o paradigma da Orientação a Objetos.

A UML apresenta diversos tipos de diagramas contendo as diferentes visões necessárias ao projeto de aplicações OO. Alguns desses diagramas são essenciais para o entendimento e implementação do software desejado.

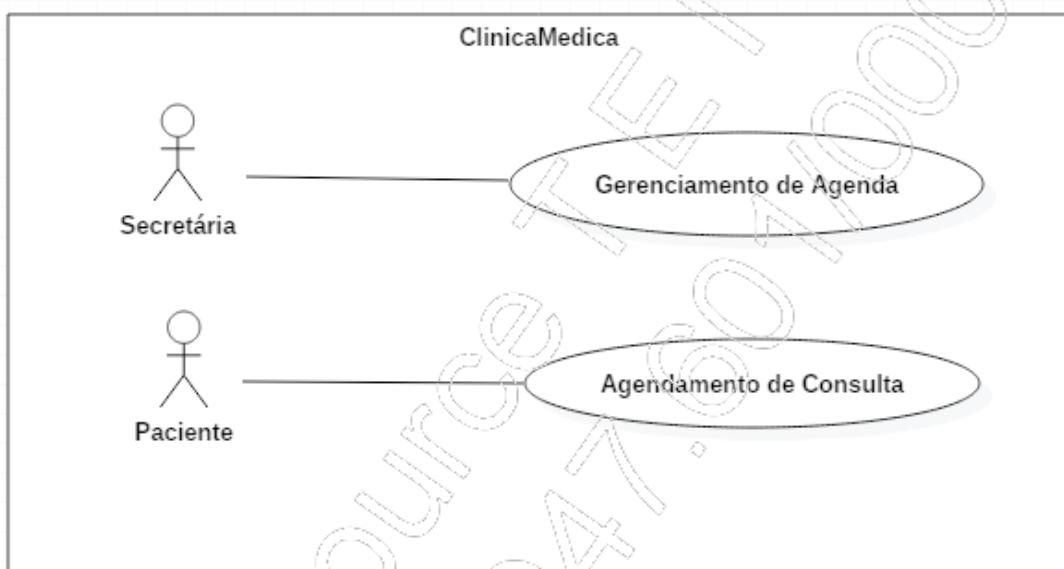
A seguir, uma visão geral dos diagramas básicos numa fase inicial do desenvolvimento de um software: de casos de uso, classes e pacotes.

5.8.1. Diagrama de casos de uso

O diagrama de casos de uso é utilizado nas fases iniciais do desenvolvimento de sistemas. Descreve um conjunto de ações (casos de uso) que um sistema deve executar em colaboração com um ou mais usuários externos ao sistema (atores), que resultem em um valor real para os envolvidos com o sistema.

Ele apresenta uma visão geral das fronteiras e grandes “temas” do sistema atrelada a um ou mais documentos contendo o passo a passo de cada funcionalidade (descrição de cenários).

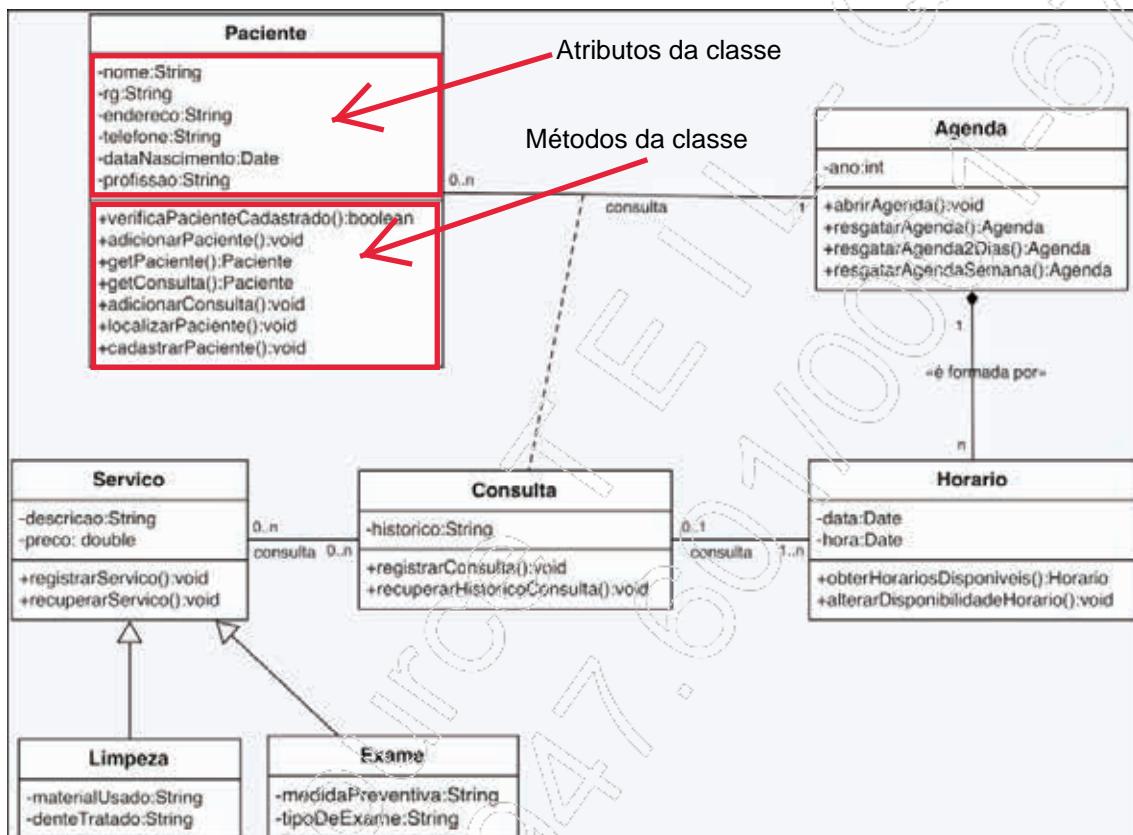
A seguir, um exemplo de como se apresenta um diagrama de caso de uso:



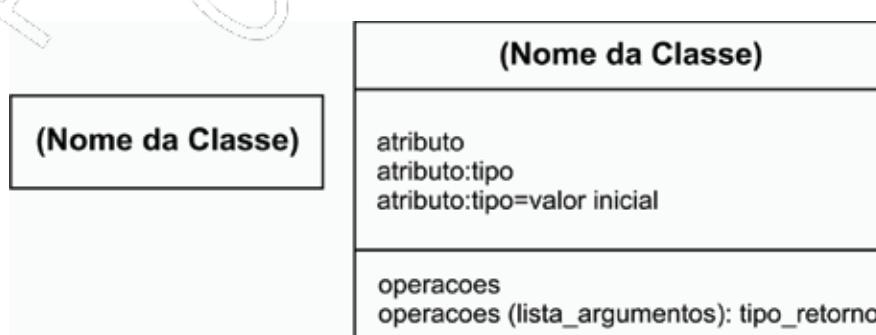
5.8.2. Diagrama de classes

O diagrama de classes descreve a estrutura estática das classes de um sistema, o que inclui o nome da classe, seus atributos e operações e seus relacionamentos com outras classes.

A seguir, veja um exemplo de diagrama de classes em um modelo fictício:

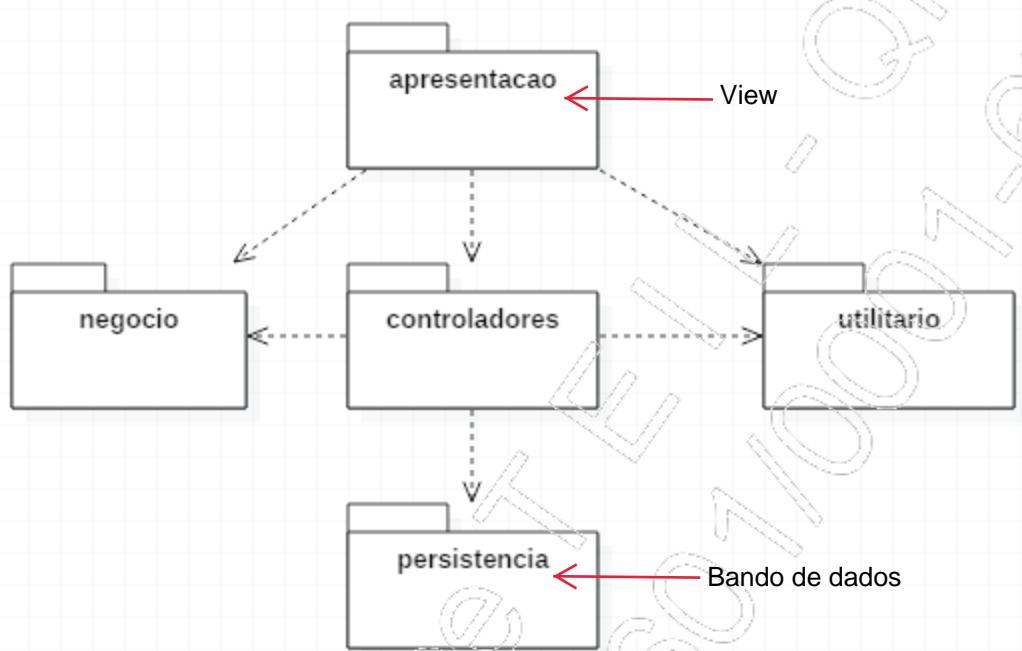


A notação UML para classes consiste em um retângulo contendo o nome da classe em negrito e seções opcionais para os atributos e operações (estas seções devem ser separadas por linhas horizontais):



5.8.3. Diagrama de pacotes

O diagrama de pacotes representa uma visão de “camadas” do sistema, ou seja, exibe a organização e dependências entre grupamentos de classes que compõem o sistema como um todo. Adiante, um exemplo de sua disposição:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

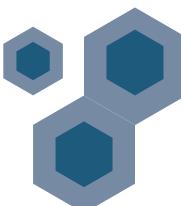
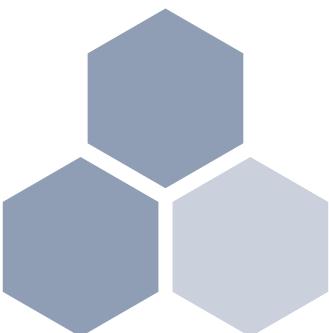
- Uma classe é um modelo para um conjunto de objetos que possuem atributos e métodos comuns entre si. Os objetos são considerados instâncias das classes;
- No campo computacional, objetos são elementos que representam uma entidade no domínio de interesse do problema analisado;
- Os objetos possuem atributos, que são as suas características particulares, e métodos, que definem as ações dos objetos;
- O encapsulamento permite ocultar os detalhes da implementação interna de uma classe, restringindo o acesso a suas variáveis e métodos. Isso facilita a legibilidade e ampliação do código;
- Pacotes representam o conceito de namespace e agrupam classes com comportamentos comuns dentro de um sistema;
- UML é uma notação padronizada para representação de diferentes visões de um sistema computacional. Dentre os principais estão os diagramas de casos de uso, classes, pacotes e sequência.



5

Introdução à orientação a objetos

Teste seus conhecimentos



1. No campo computacional, o que é um objeto?

- a) É a implementação de uma operação para uma classe específica.
- b) É um elemento que representa uma entidade, abstrata ou concreta, no domínio de interesse do problema analisado.
- c) É uma característica específica de uma classe.
- d) É um elemento que representa um método, abstrato ou concreto, no domínio de interesse do problema analisado.
- e) Nenhuma das alternativas anteriores está correta.

2. O que é um atributo?

- a) É um elemento que representa um método, abstrato ou concreto, no domínio de interesse do problema analisado.
- b) É um elemento que representa uma entidade, abstrata ou concreta, no domínio de interesse do problema analisado.
- c) É uma característica específica de um método.
- d) É uma característica específica de um objeto.
- e) Nenhuma das alternativas anteriores está correta.

3. Dentro do paradigma da POO, o que é uma classe?

- a) É um elemento que representa uma entidade, abstrata ou concreta, no domínio de interesse do problema analisado.
- b) É a implementação de uma operação para uma classe específica.
- c) É um conjunto de elementos com características comuns.
- d) É uma característica específica de um objeto.
- e) Nenhuma das alternativas anteriores está correta.

4. O que é um pacote?

- a) É um objeto que armazena um conjunto de atributos com características ou uma finalidade em comum.
- b) É uma classe que armazena um conjunto de métodos com características ou uma finalidade em comum.
- c) É um diretório que armazena um conjunto de classes com características ou uma finalidade em comum.
- d) É um programa que armazena um conjunto de comandos com características ou uma finalidade em comum.
- e) Nenhuma das alternativas anteriores está correta.

5. Qual das afirmações sobre encapsulamento está correta?

- a) Permite ocultar os seus atributos, de modo que eles só possam ser lidos ou alterados pelos métodos da própria classe.
- b) Permite mostrar os seus atributos, de modo que eles possam ser lidos ou alterados pelos métodos de qualquer classe.
- c) Permite ocultar os seus atributos, de modo que eles não possam ser lidos ou alterados.
- d) Permite mostrar os seus atributos, de modo que eles possam ser lidos ou alterados pelos métodos da própria classe.
- e) Nenhuma das alternativas anteriores está correta.

6. O que é instanciação?

- a) É um processo por meio do qual se realiza a implementação de um método existente.
- b) É um processo por meio do qual se realiza a implementação de uma classe existente.
- c) É um processo por meio do qual se realiza a cópia de um método existente.
- d) É um processo por meio do qual se cria um objeto a partir de uma classe concreta existente.
- e) Nenhuma das alternativas anteriores está correta.

7. Qual o tipo de acesso que uma classe deve ter para acessar outra classe de um pacote diferente?

- a) Padrão
- b) Privado
- c) Público
- d) Abstrato
- e) Nenhuma das alternativas anteriores está correta.



5

Introdução à orientação a objetos

Mãos à obra!

Resolução
de
OAB
-
Enunciados
-
Questões
-
Soluções
-
Quntas



Laboratório 1

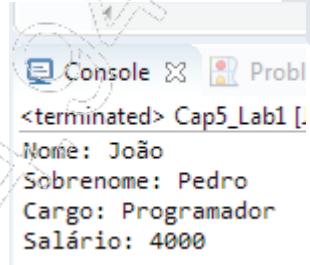
A – Criando uma classe Funcionario

1. Crie um pacote chamado **cap5lab1**;
2. Dentro do pacote, crie uma classe chamada **Funcionario**;
3. Defina os atributos **nome**, **sobrenome**, **cargo** e **salario** na classe;
4. Salve a classe.

B – Criando uma classe para testar a classe Funcionario

1. No pacote **cap5lab1**, crie uma classe chamada **Cap5_Lab1** e insira a estrutura básica de um programa Java;
2. Declare um objeto do tipo **Funcionario** e o instancie;
3. Determine valores para os atributos **nome**, **sobrenome**, **cargo** e **salario** do objeto criado;
4. Imprima os atributos do objeto na tela;
5. Compile e execute o programa.

O resultado deve ser como o exibido a seguir:



```
Console X Problemas
<terminated> Cap5_Lab1 []
Nome: João
Sobrenome: Pedro
Cargo: Programador
Salário: 4000
```

Laboratório 2

A - Criando uma classe Carro

1. Crie uma classe chamada **Carro** dentro de um pacote denominado **carro**;
2. Defina como atributos dessa classe: **modelo**, **potência do motor** e **cor**;
3. Salve a classe.

B - Criando uma classe Garagem que contém carros como atributos

1. No mesmo pacote da classe **Carro**, crie uma classe denominada **Garagem**;
2. Crie dois atributos do tipo **Carro** nessa classe e nomeie-os de **carroPasseio** e **carroUtilitario**;
3. Crie a mesma estrutura básica de um programa Java nessa classe (método **main(String [] args)**) e, dentro desse método, instancie um objeto do próprio tipo **Garagem** com o nome de “**g**”;
4. Dentro do método **main**, e utilizando o objeto **g**, do tipo **Garagem**, configure valores adequados à sua escolha para os atributos de cada um dos objetos do tipo **Carro** criados na classe **Garagem**;
5. **Imprima na tela os atributos de cada um dos carros criados;**
6. Compile e execute o programa.

O resultado será o seguinte:

```
Console X Problems
<terminated> Garagem [Java Application] /Library/Java/...
Carro de Passeio:
Cor: vermelho
Modelo: VW Jetta
Potência: 2.3
-----
Carro Utilitário:
Cor: branco
Modelo: Renault Boxer
Potência: 3.8
```




5

Introdução à orientação a objetos

Projeto Prático – Fase 1



Apresentação

Vamos iniciar nosso projeto prático, que consiste no desenvolvimento de uma aplicação Java completa, desde a análise de requisitos até a implantação do sistema e utilização pelo usuário final. Este sistema será um software utilitário nos moldes do IMDb, site de crítica popular de filmes e séries (para mais detalhes, acesse <http://www.imdb.com/>).

Esse sistema utilitário conterá as seguintes funcionalidades:

- Cadastro de dados de filmes e suas notas médias;
- Importação de banco de dados de filmes extraído do site para a aplicação;
- Consulta de filmes cadastrados;
- Sorteio de um filme para assistir baseado em critérios do usuário.

Atividades

Os requisitos do sistema serão apresentados através de diagramas UML e artefatos relacionados.

Como dados de entrada, serão apresentados:

- Diagrama de casos de uso com descrição dos cenários;
- Diagrama de classes;
- Diagrama de pacotes.

A seguir, as atividades de projeto que serão desenvolvidas nesta fase:

- Criação do projeto na IDE;
- Codificação das classes conforme diagrama apresentado.

6

Métodos

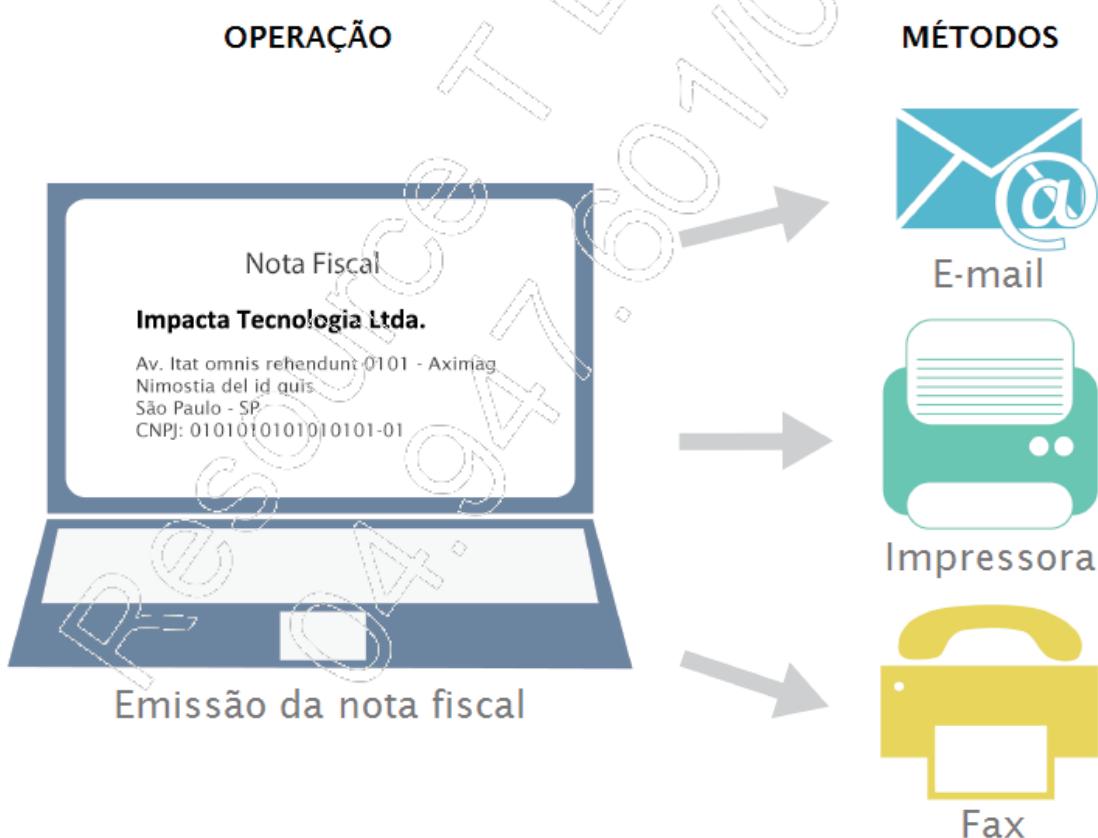
- Estrutura de um método;
- Comando return;
- Chamando um método (mensagens);
- Passagem de parâmetros;
- Varargs;
- Métodos assessores;
- Modificadores de métodos;
- Modificador static;
- Método main();
- Sobrecarga de métodos;
- UML – Diagrama de sequência.

6.1. Introdução

Os métodos definem as ações dos objetos. Por meio deles, os objetos podem ter seus atributos modificados, manifestarem-se e interagir com outros objetos.

Podemos dizer também que um método funciona como a implementação de uma operação para uma classe específica. Uma operação é uma transformação que pode ser aplicada a objetos. Os objetos de uma mesma classe compartilham as mesmas operações. Toda operação possui um objeto-alvo e é a classe desse objeto que determina o comportamento da operação.

O conceito de método pode ser ilustrado com a seguinte situação: imagine que você deve emitir uma nota fiscal. A emissão da nota é a operação. Para fazer isso, você tem diversas possibilidades, dentre as quais o envio por e-mail e a impressão em papel. Esses são os métodos, ou possibilidades de implementação da operação.



6.2. Estrutura de um método

A sintaxe de um método é apresentada e explicada a seguir:

```
<modificadores> <tipoRetorno> <nomeDoMétodo>(<lista de  
parâmetros>){  
    // Corpo do método com instruções  
}
```

Em que:

- **modificador(es)**: Refere-se a um ou mais modificadores existentes, os quais definirão as regras de visibilidade, sobrescrita e outros aspectos comportamentais do método;
- **tipoRetorno**: Refere-se ao tipo de dado válido retornado pelo método. Pode ser qualquer tipo de dados, incluindo aqueles de classes desenvolvidas pelo programador. Se o método não retornar um valor, o tipo obrigatório de retorno a ser usado será **void**;
- **nomeDoMétodo**: Refere-se ao nome do método, que pode ser um identificador legal não utilizado por outros itens do escopo atual;
- **lista de parâmetros**: Refere-se à sequência de pares compostos por um tipo e um identificador, os quais estão separados por vírgulas. Os parâmetros são variáveis, ou seja, eles recebem o valor dos argumentos que são passados para o método quando ele é chamado. Se não houver parâmetros para o método, você não deve inserir conteúdo na lista de parâmetros, porém, o par de parênteses é um conjunto obrigatório a ser escrito em todo método, ainda que vazio.

6.3. Comando return

Como você viu, quando um método não retorna um valor, ele deve ser declarado como **void**. Contudo, se ele retornar um valor, este será devolvido para a rotina que fez a chamada do método. Isso é feito pelo comando **return**, o qual, basicamente, encerra o método e retorna a execução do programa para o chamador do método. Veja o exemplo:

```
public int devolverValor(){  
    int valor = 10;  
    return valor;  
}
```

O comando **return** pode ser utilizado mesmo sem nenhum retorno declarado no método. Nessa situação, ele é opcional e seu uso encerra imediatamente a execução do método e retorna à rotina que o invocou.

6.4. Chamando um método (mensagens)

As mensagens são o meio de comunicação entre os objetos. Com elas, um objeto pode usar um método que foi definido em outro objeto. Ou seja, as mensagens são chamadas que um objeto faz a outro com o objetivo de invocar um de seus métodos. A mensagem já inclui os argumentos necessários para a execução da tarefa solicitada.

Veja um exemplo:

```
pessoa.falar("Bom dia");
```

O objeto **pessoa** está enviando a mensagem **falar** e passando o parâmetro **"Bom dia"**. Este valor será usado para executar a ação corretamente (falar "Bom dia").

Veja, agora, um exemplo de método a partir da estrutura apresentada. O método a ser criado deve somar dois números recebidos como parâmetros:

```
public int somar(int valor1, int valor2){  
    int resultado;  
    resultado = valor1 + valor2;  
    return resultado;  
}
```

Veja as explicações a seguir:

- **public** é um modificador de acesso opcional, declarado para esse método com o fim de torná-lo acessível a partir de qualquer local e por qualquer objeto do programa que também possua acesso à classe onde ele está inserido;
- **int** é o tipo de retorno do método, ou seja, define o tipo de dado **int** (valores numéricos e inteiros);
- **somar** é o nome que foi definido para o método;
- **int valor1** e **int valor2** indicam que o método (**somar**) receberá dois parâmetros do tipo inteiro que possuem os nomes **valor1** e **valor2**;
- **{** é um caractere obrigatório e indica o início do corpo do método;
- **int resultado;** significa uma variável do tipo inteiro que está sendo criada. Nesta variável, será armazenada a soma dos valores recebidos com parâmetros (**valor1** e **valor2**);
- **resultado = valor1 + valor2** refere-se à variável **resultado** que receberá o valor da soma dos parâmetros **valor1** e **valor2**;
- **return resultado;** indica que qualquer chamada ao método **somar** receberá como valor de retorno o resultado da variável **resultado**;
- **}** também é obrigatório e indica o fechamento do corpo do método.

Após definir um método, você precisa referenciá-lo, ou seja, fazer uma chamada ao método. É comum definirmos, nessa situação, dois papéis bem claros para a compreensão de métodos: o método **chamador (caller)** e o método **executor (worker)**. Em toda lógica aplicada a programas Java há, fundamentalmente, um método passando mensagens para outro que as processa e retorna com conteúdo elaborado. Nesse cenário, temos um método chamador invocando outro método, o executor. É importante frisar que os métodos Java trabalham intercambiando esses dois papéis: ora um método é chamador, ora é executor.

Veja um exemplo de como chamar um método:

No arquivo **Calculadora.java**, temos um método executor em nosso programa, “**somar**”:

```
1 public class Calculadora {  
2     public int somar(int valor1, int valor2){  
3         int resultado;  
4         resultado = valor1 + valor2;  
5         return resultado;  
6     }  
7 }  
8 }
```

No arquivo **UsaCalculadora.java**, temos o método chamador, “**main**”:

```
1 public class UsaCalculadora{  
2     public static void main(String args[]){  
3         int s; // definição simples de variável  
4         // criação de uma instância de Calculadora  
5         Calculadora calc = new Calculadora();  
6         // atribuindo a 's' a chamada do método somar com dois valores  
7         s = calc.somar(15, 20);  
8         // mostra o valor de s na tela  
9         System.out.println("s = " + s);  
10    }  
11 }  
12 }
```

O resultado será o seguinte:

```
Console × Problems  
<terminated> UsaCalculadora  
s = 35
```

6.5. Passagem de parâmetros

Na linguagem Java, a passagem de argumentos, independentemente de tratar-se de tipo primitivo ou objeto, é feita apenas por valor, ou seja, o valor do argumento não pode ser alterado dentro do método.

Quando você passa um objeto como argumento de um método, na realidade, está passando o valor da referência do objeto, e não o objeto em si. Isto quer dizer que, quando o valor da referência do objeto é passado como argumento, o seu conteúdo (valores dos atributos) pode ser alterado, mas a referência não pode ser alterada dentro do método.

- Parâmetro refere-se a uma variável definida por um método. Quando este método é chamado, a variável recebe um valor.
- Argumento refere-se a um valor atribuído a um método, quando este é invocado.

O exemplo a seguir ilustra o conceito de passagem por valor em Java:

```
J Pessoa.java X
1
2 public class Pessoa {
3     String nome;
4     int idade;
5     char sexo;
6
7     public void exibirDados(){
8         System.out.println("Nome da pessoa: " + nome);
9         System.out.println("Idade da pessoa: " + idade);
10        System.out.println("Sexo da pessoa: " + sexo);
11    }
12 }

public class PassagemPorValor {
    public static void main(String args[]){
        int valor = 1500;
        // tentativa de alterar o valor
        System.out.println("Antes da chamada do método alterarValor o valor é " + valor);
        alterarValor(valor);
        System.out.println("Depois da chamada do método alterarValor o valor é " + valor);
        System.out.println();

        Pessoa maria = new Pessoa();
        maria.sexo = 'f';
        maria.nome = "Maria";
        maria.idade = 50;

        // tentativa de alterar a referência do objeto
        System.out.println("Valores antes da chamada do método alterarReferenciaObjeto:");
        maria.exibirDados();
        alterarReferenciaObjeto(maria);
        System.out.println("Valores depois da chamada do método alterarReferenciaObjeto:");
        maria.exibirDados();
        System.out.println();

        // alterando o CONTEUDO do objeto
        System.out.println("Valores antes da chamada do método alterarConteudoObjeto:");
        maria.exibirDados();
        alterarConteudoObjeto(maria);
        System.out.println("Valores depois da chamada do método alterarConteudoObjeto");
        maria.exibirDados();
    }

    static void alterarValor(int valor){
        //alterando o valor
        valor = 137;
    }

    static void alterarReferenciaObjeto(Pessoa p){
        Pessoa ana = new Pessoa();
        ana.sexo = 'f';
        ana.nome = "Ana";
        ana.idade = 18;
        p = ana; // alterando o valor de p
    }

    static void alterarConteudoObjeto(Pessoa p){
        // alterando o conteúdo do objeto referenciado por p
        p.sexo = 'm';
        p.nome = "João";
    }
}
```

A imagem a seguir ilustra a saída:

The screenshot shows a Java application named "PassagemPorValor" running in an IDE. The console output is as follows:

```
Console Problems @ Javadoc Declaration Debug
<terminated> PassagemPorValor [Java Application] C:\Program Files\Java\jre7\bi
Antes da chamada do método alterarValor o valor é 1500
Depois da chamada do método alterarValor o valor é 1500

Valores antes da chamada do método alterarReferenciaObjeto:
Nome da pessoa: Maria
Idade da pessoa: 50
Sexo da pessoa: f
Valores depois da chamada do método alterarReferenciaObjeto:
Nome da pessoa: Maria
Idade da pessoa: 50
Sexo da pessoa: f

Valores antes da chamada do método alterarConteudoObjeto:
Nome da pessoa: Maria
Idade da pessoa: 50
Sexo da pessoa: f
Valores depois da chamada do método alterarConteudoObjeto:
Nome da pessoa: João
Idade da pessoa: 33
Sexo da pessoa: m
```

Pelo resultado final apresentado na tela, você pode observar que não se consegue alterar o valor do argumento dentro dos métodos `alterarValor(int valor)` e `alterarReferenciaObjeto(Pessoa p)`, apesar de ser possível alterar o conteúdo do objeto dentro do método `alterarConteudoObjeto(Pessoa p)`. A alteração do valor do argumento, seja ele um tipo primitivo ou referência a um tipo construído, não ocorre porque a passagem de argumentos em Java é feita por valor.

6.6. Varargs

Em alguns casos é necessário escrever métodos que aceitam parâmetros que contenham múltiplos valores de um determinado tipo. Nesses casos, um array do tipo pode ser utilizado (esse assunto será tratado mais adiante na apostila).

```
public int calcular(int[] lista) {
    int soma = 0;
    for (int i=0; i < lista.length; i++) {
        soma += lista[i];
    }
    return soma;
}
```

O código anterior recebe como parâmetro um conjunto de valores inteiros na forma de array e retorna a soma destes valores.

Em vez de se utilizar um array, outra opção seria criar diversos métodos contendo os diferentes números de argumentos necessários.

```
public int calcular(int a, int b) {  
    return a + b;  
}  
  
public int calcular(int a, int b, int c) {  
    return a + b + c;  
}  
  
public int calcular(int a, int b, int c, int d) {  
    return a + b + c + d;  
}  
  
public int calcular(int a, int b, int c, int d, int e) {  
    return a + b + c + d + e;  
}
```

Esta pode ser uma maneira mais fácil, porém menos flexível que usar array. Imagine o tamanho da classe, caso se tenha a possibilidade de informar até 100 ou mais parâmetros.

Com varargs, torna-se possível a criação de métodos que permitem um número variável de parâmetros de um mesmo tipo, deixando para o compilador o trabalho de empacotar os parâmetros informados num array dinamicamente (no momento da chamada do método), ou seja, utiliza-se array na execução do método, porém isso fica “escondido” do programador.

O código adiante mostra como o método ficaria implementado com o uso de varargs:

```
public int calcular(int...lista) {  
  
    int soma = 0;  
    for (int item : lista) {  
        soma += item;  
    }  
    return soma;  
}
```

Os três pontos presentes na assinatura do método indica que pode ser passado qualquer número de valores do tipo int para o método. Além disso, pode ser passando também um array de int.

```
calcular(-3, 25, 13, 0, 76, 89, 56);  
calcular(1, -1, 2);  
calcular(7, 17, 27, 37, 47, 57, 67, 77, 87, 97);
```

Um ponto importante na utilização de varargs é que se ele não for o único parâmetro do método, ele tem que aparecer sempre como último elemento do conjunto de parâmetros.

6.7. Métodos assessores

Conforme os ditames do encapsulamento, é sempre uma boa prática proteger os atributos de uma classe de acessos exteriores indevidos. O estado interno de um objeto é algo de extrema importância para qualquer programa. Dessa forma, utilizamos métodos assessores que permitem obter um maior grau de gerenciamento sobre o acesso aos atributos.

Os métodos designados para essa função e delimitados pela convenção JavaBeans são denominados **getters** e **setters**. Esses métodos possibilitam o acesso aos atributos de um objeto e sua modificação.

A convenção JavaBeans está disponível para download em <http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html>.

Você pode determinar que tipo de atributo sofrerá alterações, assim como definir em que momento estas alterações poderão ocorrer. A validação dos valores a serem especificados aos atributos é outra capacidade viabilizada pelos métodos.

Com os métodos, não só o controle de um atributo é facilitado, como também as modificações posteriores se tornam bem mais práticas. Da próxima vez que precisar alterar um atributo, certamente você não terá que se preocupar em alterar a assinatura do método usado para acessar o atributo.

Depois que um atributo é alterado com o uso de um método, é possível gerar um log com o registro das alterações realizadas.

A seguir, apresentamos os métodos padronizados **getter** e **setter**.

6.7.1. Método getter

O uso do método **getter** é a forma adequada de se encapsular um campo em uma classe e fornecer acesso de leitura ao seu valor.

A seguir, temos a sintaxe de uso do método **getter**:

```
Tipo do atributo get + Nome_do_atributo ();
```

A sintaxe e o uso de **get** são exemplificados a seguir:

```
1  public class AcessoresGet {  
2      private String nome;  
3  
4      String getName(){  
5          return this.nome;  
6      }  
7  }
```

6.7.2. Método setter

Para configurar o valor apresentado pelos atributos em uma classe, você dispõe do método **setter**, cuja sintaxe é a seguinte:

```
void set + Nome do atributo (<tipo da variável> <nome da variável>)
```

Para alteração de valores, o método **setter** tem a vantagem de proteger os atributos contra leituras e modificações não autorizadas, ou seja, por meio deste método, o acesso aos atributos não é público.

 Os métodos **setters** podem conter algumas regras ou formas de gerenciamento.

O uso deste método é exemplificado a seguir:

```
1  public class AcessoresSet {  
2      private String nome;  
3  
4      void setName(String nome){  
5          this.nome = nome;  
6      }  
7  }
```

6.8. Modificadores de métodos

Um método possui diversas capacidades e propriedades que podem ser configuradas de modo a customizar seu comportamento na classe onde é declarado. As funções dessas propriedades vão desde a possibilidade de ser sobreescrito em uma classe derivada até garantir acesso singular a uma única linha de execução em um ambiente multithreaded (não se preocupe, threads serão assunto de um capítulo futuro).

Veja, a seguir, quais são os modificadores de método disponíveis na linguagem Java:

- **abstract**: O método que possui o modificador **abstract** é abstrato, ou seja, não tem o seu corpo especificado. Pode-se dizer que este modificador exige a criação de código específico a ser feito por uma classe derivada;
- **final**: Com este modificador, o método não pode ser alterado nem redefinido por nenhuma classe derivada. Você deve saber que é obrigatório que o método declarado como final possua um corpo;

Os modificadores **abstract** e **final** serão explicados com maior clareza e detalhes no capítulo que trata sobre herança.

- **native**: Ao utilizar este modificador, apenas o cabeçalho é declarado, ou seja, não há corpo. Adicionalmente, o **native** designa e representa um método implementado em outra linguagem;
- **synchronized**: Este modificador de método impede que os dados de uma classe sejam acessados ao mesmo tempo por duas threads (linhas de execução) concorrentemente. Sendo assim, os outros métodos precisam esperar para acessar os dados que estiverem sendo acessados por outro método. O **synchronized** é utilizado para o desenvolvimento de um programa de processamento concorrente;
- **strictfp**: Este modificador, aplicável a métodos e classes, garante conformidade no tratamento de números de ponto flutuante em seu programa, garantindo que as instruções no código gerado pelo compilador sejam tratadas da mesma forma em todas as plataformas em que o programa for executado. Utiliza como padrão as instruções IEEE 754.

Há, ainda, outro modificador de métodos, o **static**, que será abordado com maior profundidade no tópico a seguir.

6.9. Modificador static

Para especificar que um membro (atributo ou método) de uma classe é estático, você deve prefixar à sua declaração o modificador **static**.

Membros estáticos podem ser referenciados pelo próprio nome da classe em que foram definidos. Não é necessário criar objetos desta classe para acessar os membros estáticos fora dela. Por este motivo, você pode se referenciar a eles como membros de classe.

6.9.1. Atributos estáticos

Ao colocar o modificador **static** juntamente com a definição de um atributo, este se torna estático. Isso significa que o atributo passa a pertencer a um contexto global em sua aplicação (em toda a JVM) e uma única instância desse atributo existirá durante o programa. Apesar de referenciado em todas as instâncias dessa classe, quando acessado de qualquer uma delas, os mesmos valores serão compartilhados.

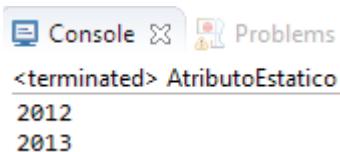
Caso existam 100 instâncias provenientes da mesma classe, quando você alterar este atributo em uma dessas instâncias, automaticamente, a alteração refletirá nas demais instâncias da classe, justamente porque o acesso está ocorrendo no mesmo atributo, alocado em um contexto estático.

Veja o exemplo a seguir:

```
Classe.java
1 public class Classe {
2     static int ano;
3 }
```

```
AtributoEstatico.java
1 public class AtributoEstatico {
2     public static void main(String args[]){
3         Classe a = new Classe();
4         Classe b = new Classe();
5         Classe c = new Classe();
6
7         a.ano = 2012;
8         System.out.println(c.ano);
9         b.ano = 2013;
10        System.out.println(c.ano);
11    }
12 }
13 }
```

O resultado será o seguinte:



A screenshot of a Java IDE showing the output of a static method. The console tab shows the output of the `mostraNome` method from the `AtributoEstatico` class, which prints the years 2012 and 2013.

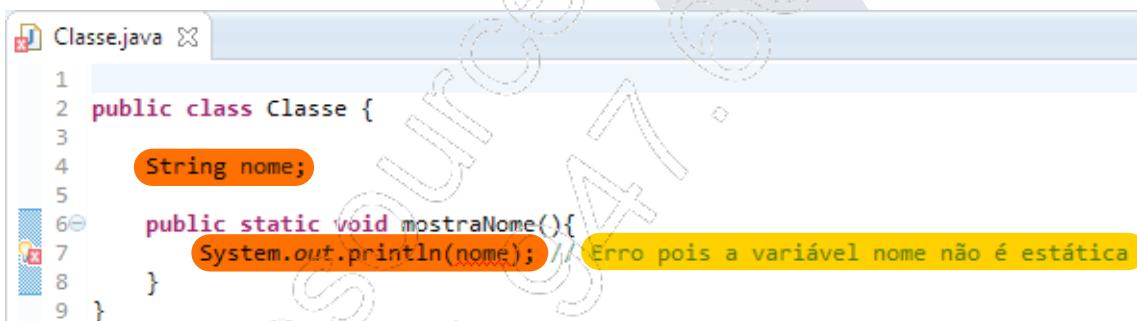
```
Console Problems
<terminated> AtributoEstatico
2012
2013
```

6.9.2. Métodos estáticos

Ao colocar o modificador **static** juntamente com a definição de um método, este se torna estático. Isto não significa que o seu valor não possa mudar, mas que somente haverá uma referência para esse método disponível em toda a aplicação.

O **static** é um modificador da linguagem Java que, quando colocado junto ao método, indica que dentro dele somente é possível acessar outros métodos ou atributos que também sejam declarados como **static**. Você deve sempre se lembrar de que esta é a principal consequência de se utilizar um método **static**.

Veja um exemplo:



A screenshot of a Java code editor showing a class named `Classe.java`. The code defines a class with a non-static attribute `nome` and a static method `mostraNome` that prints the value of `nome`. A tooltip indicates a compile-time error: "Erro pois a variável nome não é estática".

```
1 public class Classe {
2     String nome;
3
4     public static void mostraNome(){
5         System.out.println(nome);
6     }
7 }
8 }
9 }
```

Para acessar um membro ou um método que não seja estático, você deve criar uma instância da classe correspondente para chamar o item desejado. Porém, quando os métodos não são estáticos, é possível acessar ambas as propriedades, estáticas e não estáticas.

Isto tudo ocorre porque não há, dentro do método estático, uma referência para o ponteiro **this**, utilizado quando é necessário referenciar as propriedades da classe com a qual você está trabalhando.

Veja o exemplo a seguir:

The screenshot shows an IDE interface with two tabs: 'Classe2.java' and 'MetodoEstatico.java'. The 'Classe2.java' tab is active, displaying the following code:

```
1 public class Classe2 {  
2     public static void metodoEstatico(){  
3         System.out.println("Método Estático");  
4     }  
5     public void metodoNãoEstatico(){  
6         System.out.println("Método Não Estático");  
7     }  
8 }  
9  
10 }
```

The 'MetodoEstatico.java' tab is also visible, showing the following code:

```
1 public class MetodoEstatico {  
2     public static void main(String args[]){  
3         Classe2.metodoEstatico(); // Não precisa ser instanciada  
4         new Classe2().metodoNãoEstatico(); // Está sendo instaciada  
5     }  
6 }  
7 }
```

O resultado será o seguinte:

The screenshot shows the 'Console' tab of the IDE, which displays the following output:

```
<terminated> MetodoEstatico  
Método Estático  
Método Não Estático
```

6.9.3. Exemplos práticos de membros estáticos

Membros estáticos são tipicamente utilizados para a criação de constantes e de rotinas (métodos) de utilidade geral. Veja os exemplos de utilização a seguir:

- **Criação de constantes**

Em geral, constantes são criadas como atributos estáticos, pois são valores fixos e independentes de quaisquer instâncias daquela classe:

The screenshot shows the 'Trigonometria.java' file with the following code:

```
1 public class Trigonometria {  
2     public static final double PI = 3.1415926;  
3 }  
4  
5 }
```

Por ser estático, podemos acessar este valor sem precisar instanciar a classe:

```
double perimetro = 2 * raio * Trigonometria.PI;
```

- **Criação de rotinas (métodos) de utilidade geral**

Outra situação típica em que podemos utilizar membros estáticos é quando criamos métodos utilitários.

Os métodos utilitários são pequenas rotinas utilizadas no cálculo ou no processamento geral e que podem ser chamados por diversos pontos de sua aplicação:

```
1  public class String_Utils {  
2  
3      public static String inverterString(String valor) {  
4          char[] letras = valor.toCharArray();  
5          char[] letrasInversas = new char[letras.length];  
6  
7          for (int i = 0; i < letras.length; i++) {  
8              letrasInversas[letras.length - i - 1] = letras[i];  
9          }  
10         return new String(letrasInversas);  
11     }  
12 }  
13  
14 }
```

Assim sendo, podemos utilizar este método sem precisar instanciar a classe:

```
String inverso = String_Utils.inverterString("roma");
```

6.10. Método main()

O **main()** é o método **static** mais famoso na linguagem Java. É por meio dele que a aplicação passa a ser executada. Portanto, quando você deseja utilizar algum método ou realizar alguma operação no programa, faça isso dentro do método **main()**.

É importante observar que, para que possa ser executada, toda aplicação ou programa para desktop escrito em linguagem Java deve ter, obrigatoriamente, um único método **main()**, designado como principal. Sendo assim, veja as regras para utilizar este método:

- O acesso deve ser **public**;
- Deve ser **static**;
- Deve ser **void**;
- Deve receber como argumento um array de tipo **String (String args[])**.

A seguir, temos um exemplo que ilustra o fluxo de execução de um programa em Java quando existem chamadas de métodos:

```
1  public class ExemploFluxoExecucaoComandos {  
2      /** definição do método executarTarefa **/  
3      static void executarTarefa(){  
4          System.out.println("Executando uma tarefa");  
5      }  
6      /** término da definição **/  
7  
8      public static void main(String args[]){  
9          System.out.println("Executando o primeiro comando");  
10         System.out.println("Executando o segundo comando");  
11         System.out.println(" . ");  
12         System.out.println(" . ");  
13         System.out.println(" . ");  
14  
15         executarTarefa();  
16  
17  
18         System.out.println("Sou o próximo comando que segue a chamada do método");  
19         System.out.println(" . ");  
20         System.out.println(" . ");  
21         System.out.println(" . ");  
22         System.out.println("Executando o último comando");  
23     }  
24 }
```

Depois de compilado e executado o código anterior, o resultado será como mostra a figura adiante:

The screenshot shows a Java application named "ExemploFluxoExecucaoComandos" running in a terminal window. The output is as follows:

```
<terminated> ExemploFluxoExecucaoComandos [Java Application]
Executando o primeiro comando
Executando o segundo comando
.
.
.
Executando uma tarefa
Sou o próximo comando que segue a chamada do método
.
.
.
Executando o último comando
```

No exemplo a seguir, outros métodos podem ser chamados de forma encadeada:

The screenshot shows a Java file named "ExemploChamadaDeMetodo.java" in an IDE. The code defines a class with three methods: "executarTarefaAA", "executarTarefaBB", and "main". The "main" method calls "executarTarefaBB", which in turn calls "executarTarefaAA".

```
1  public class ExemploChamadaDeMetodo {
2      static void executarTarefaAA(){
3          System.out.println("Executando tarefa AA");
4      }
5
6      static void executarTarefaBB(){
7          System.out.println("Executando tarefa BB");
8          executarTarefaAA();
9      }
10
11     public static void main(String args[]){
12         System.out.println("\\"main\\", é o ponto de partida");
13         executarTarefaBB();
14     }
15 }
16 }
```

Depois de compilado e executado o código anterior, o resultado será como o da figura adiante:

The screenshot shows the execution of the "ExemploChamadaDeMetodo" application in a terminal window. The output is as follows:

```
<terminated> ExemploChamadaDeMetodo
"main", é o ponto de partida
Executando tarefa BB
Executando tarefa AA
```

6.11. Sobrecarga de métodos

Na linguagem Java, é possível definir, dentro de uma mesma classe, dois ou mais métodos de mesmo nome, porém de assinaturas diferentes. O conceito de assinatura, na linguagem Java, remete ao conjunto de componentes do método que o tornam único, sob os olhos do compilador. Esse conjunto é composto pelo nome do método e seus parâmetros. O conjunto de parâmetros deve variar em tipo, número e/ou ordem para que se tenha uma assinatura de métodos diferentes. Este procedimento é conhecido como **sobrecarga de métodos**. É importante lembrar que o tipo de retorno do método não faz parte de sua assinatura, ou seja, mudando-se o tipo de retorno, a assinatura do método permanece a mesma.

Ao chamar um método sobrecarregado, você utiliza um guia para definir qual versão desse método deve ser executada. Este guia corresponde ao tipo, ao número dos argumentos e à sua ordem e, por esse motivo, deve haver uma configuração diferente para cada um dos métodos sobrecarregados.

Mas você deve estar atento para a questão de que o tipo retornado dos métodos sobrecarregados não é suficiente para que as suas versões do método sejam distintas. Isso ocorre porque a versão do método que possui parâmetros correspondentes com os argumentos da chamada é executada quando for encontrada uma chamada a um método sobrecarregado.

Observe um exemplo de sobrecarga de métodos:

```
1  public class SobreCarga {  
2      void mostrar(int valor){  
3          System.out.println("O valor informado foi: " + valor);  
4      }  
5  
6      void mostrar(String nome){  
7          System.out.println("Foi informado o nome: " + nome);  
8      }  
9  
10     void mostrar(){  
11         System.out.println("Nada foi informado!");  
12     }  
13 }  
14 }
```

```
1  public class ExemploSobreCarga {  
2      public static void main(String args[]){  
3          SobreCarga sob = new SobreCarga();  
4          sob.mostrar("Ernesto");  
5          sob.mostrar();  
6          sob.mostrar(38);  
7      }  
8  }  
9 }
```

Depois de compilado e executado o código anterior, o resultado será como o da figura a seguir:

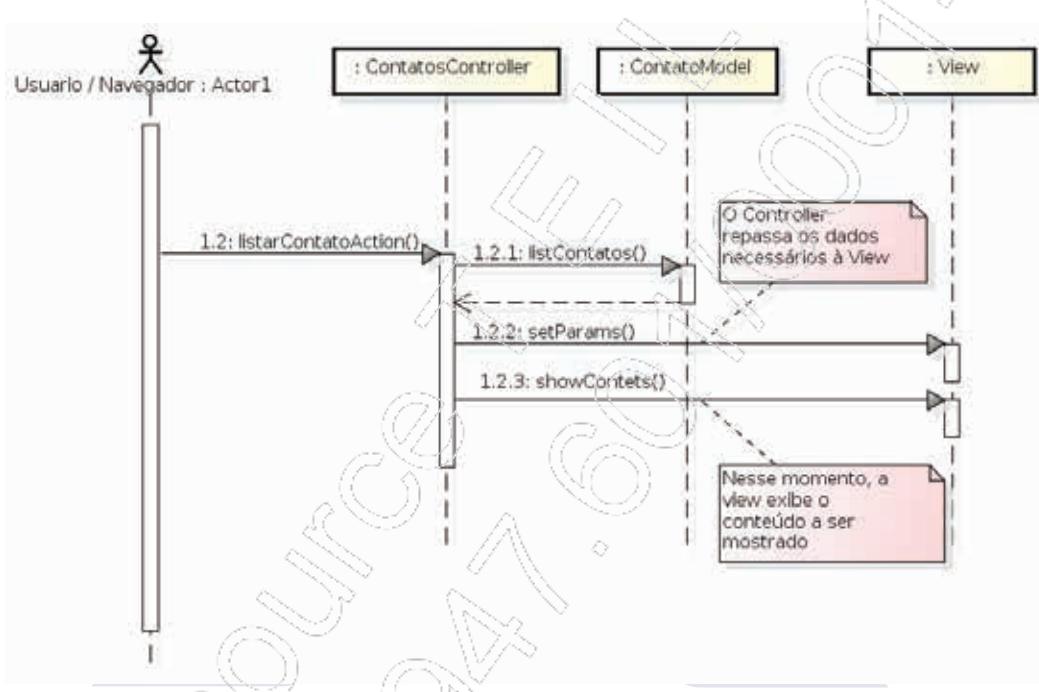
```
Console Problems @ Java  
<terminated> ExemploSobreCarga [Java]  
Foi informado o nome: Ernesto  
Nada foi informado!  
O valor informado foi: 38
```

6.12.UML – Diagrama de sequência

O diagrama de sequência é usado para mostrar as interações entre os objetos em ordem sequencial de ocorrência. Cada cenário descrito a partir dos casos de uso representa um diagrama de sequência.

Na prática, um caso de uso tem seu cenário principal representado por um diagrama de sequência que orienta o desenvolvimento dos métodos em uma aplicação.

A seguir, um exemplo de como é definido um diagrama de sequência:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

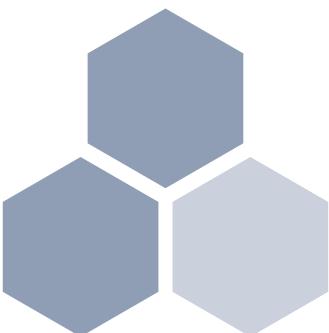
- Ao utilizar os métodos, que são a implementação de uma operação para uma classe, você determina o comportamento dos objetos de uma classe. Os métodos podem ser aplicados a várias classes diferentes, mas todos os objetos que são instâncias de uma mesma classe compartilham os mesmos métodos. Cada método possui uma função específica e deve ser definido antes de ser utilizado;
- A sintaxe de um método é composta pelos seguintes itens: **modificador(es); tipoRetorno; nomeDoMetodo; lista de parâmetros;**
- Após definir um método, você precisa referenciá-lo, ou seja, fazer uma chamada ao método. Os métodos assumem sempre um de dois papéis, chamador ou executor, podendo servir tanto como executor em um momento do programa como chamador em outro determinado instante;
- Na linguagem Java, a passagem de argumentos é feita apenas por valor, ou seja, o valor do argumento não pode ser alterado dentro do método, independentemente da natureza do argumento: se tipo primitivo ou objetos;
- Quando você precisar restringir o acesso a um método, utilize modificadores de acesso, os quais definem o nível de visibilidade do método em relação às outras classes. Porém, você deve saber que um método sempre é acessível pelos métodos da própria classe, independentemente do modificador que for escolhido para a restrição. Isto significa que você pode utilizar outro método qualquer que está na mesma classe para chamar o método cujo acesso está restrito;
- Um método possui propriedades e capacidades adicionais à disposição do desenvolvedor para se almejar comportamento específico. Para isso, utilize os modificadores de método;
- O **main()** é o método **static** mais famoso na linguagem Java. É por meio dele que a aplicação passa a ser executada. Portanto, quando você deseja utilizar algum método ou realizar alguma operação no programa, faça isso dentro do método **main()**;
- Na linguagem Java, é possível definir dois ou mais métodos de mesmo nome e lista de parâmetros diferentes dentro de uma mesma classe. Este procedimento é conhecido como sobrecarga de métodos. Neste procedimento, as declarações da lista de parâmetros devem ser diferentes em tipo, número e/ou ordem, porém, o nome utilizado é o mesmo;
- O diagrama de sequência é usado para mostrar as interações entre os objetos em ordem sequencial de ocorrência e auxilia a criação de métodos em um sistema.



6

Métodos

Teste seus conhecimentos



1. O que é um método?

- a) É a implementação de uma operação para uma classe específica.
- b) É uma característica específica de um objeto.
- c) É um elemento que representa uma entidade, abstrata ou concreta, no domínio de interesse do problema analisado.
- d) É um conceito central na programação orientada a objetos.
- e) Nenhuma das alternativas anteriores está correta.

2. Qual é a sintaxe de um método?

- a) <nomeDoMétodo> <modificador> <tipoRetorno> (<listaDeParâmetros>){ }
- b) <tipoRetorno> <modificador> <nomeDoMétodo> (<listaDeParâmetros>){ }
- c) <modificador> <tipoRetorno> <nomeDoMétodo> (<listaDeParâmetros>){ }
- d) <nomeDoMétodo> <tipoRetorno> <modificador> (<listaDeParâmetros>){ }
- e) Nenhuma das alternativas anteriores está correta.

3. Qual o valor que o método a seguir deve retornar?

```
public int somar(){  
    int a = 15, b = 3;  
    return a+b;  
}
```

- a) 15
- b) 18
- c) 3
- d) 12
- e) 45

4. Quais são os modificadores de acesso a métodos disponíveis em Java?

- a) public, protected, private e void.
- b) public, abstract, private e protected.
- c) public, protected, private e friendly.
- d) public, abstract, void e friendly.
- e) Nenhuma das alternativas anteriores está correta.

5. Quais são os modificadores de método disponíveis em Java?

- a) abstract, static, void, native e final.
- b) abstract, native, synchronized, protected e public.
- c) public, final, private, static e void.
- d) abstract, final, native, synchronized e static.
- e) Nenhuma das alternativas anteriores está correta.

6. Em que situação ocorre uma sobrecarga de métodos?

- a) Quando dois ou mais métodos tem o mesmo nome e lista de parâmetros diferentes dentro de uma classe.
- b) Quando dois ou mais métodos tem o mesmo nome e a mesma lista de parâmetros dentro de uma classe.
- c) Quando dois ou mais métodos tem a mesma lista de parâmetros e nomes diferentes dentro de uma classe.
- d) Quando dois ou mais métodos tem lista de parâmetros e nomes diferentes dentro de uma classe.
- e) Nenhuma das alternativas anteriores está correta.



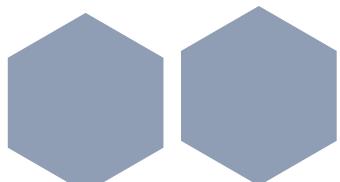
6

Métodos



Mãos à obra!

Quintessential
TEIL - 10007-67
Resource
OA.



Laboratório 1

A – Implementando a classe Calculadora

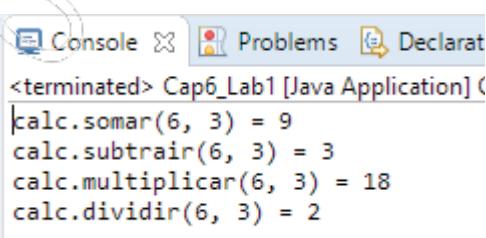
A partir da classe **Calculadora**, usada como exemplo no capítulo 6, vamos implementar os métodos **subtrair**, **multiplicar** e **dividir**.

1. Abra a classe **Calculadora**;
2. Crie um método público que retorne o tipo **int**, com o nome **subtrair** e que receba dois parâmetros do tipo **int**;
3. Dentro do método, retorne a diferença entre o primeiro parâmetro e o segundo parâmetro;
4. Crie os métodos públicos **multiplicar** e **dividir**, retornando, respectivamente, a multiplicação e divisão dos parâmetros;
5. Salve e compile o programa.

B – Testando a classe Calculadora

1. Crie uma classe chamada **Cap6_Lab1** e insira a estrutura básica de um programa Java;
2. Instancie um objeto da classe **Calculadora**;
3. Use os métodos **somar**, **subtrair**, **multiplicar** e **dividir**. Imprima os resultados na tela;
4. Compile e execute o programa.

O resultado será o seguinte:



```
Console Problems Declarat
<terminated> Cap6_Lab1 [Java Application]
calc.somar(6, 3) = 9
calc.subtrair(6, 3) = 3
calc.multiplicar(6, 3) = 18
calc.dividir(6, 3) = 2
```

Laboratório 2

A – Criando métodos sobrecarregados na classe Calculadora

1. Utilizando a classe **Calculadora** do laboratório anterior, crie três sobrecargas do método **subtrair**, uma recebendo como parâmetros dois tipos **double**, outra recebendo um **int** e um **double** e uma terceira recebendo um **double** e um **int**. Não se esqueça de seguir essa ordem;

2. Salve e compile a classe.

B – Testando a nova versão da classe Calculadora

1. Crie uma classe chamada **Cap6_Lab2** e, nessa classe, crie o método **main**;

2. Dentro dessa classe, crie uma instância de **Calculadora** e faça chamadas às versões sobrecarregadas do método **subtrair**, criadas neste laboratório;

3. Execute o programa.

A saída será a seguinte:

```
Console X Problems Declaration @-Javadoc
<terminated> Cap6_Lab2 [Java Application] C:\Program Files\Java\jre-9\bin\
Métodos subtrair sobrecarregados:
Chamando a versão de subtrair com 2 double: 3.1
Chamando a versão de subtrair com 1 double e um int: 4.2
Chamando a versão de subtrair com 1 int e um double: 2.8
```

Laboratório 3

A - Criando a classe Aluno

1. Crie uma classe denominada **Aluno**;
2. Nessa classe, crie um campo de acesso **private** da string chamada **nome**;
3. Crie os métodos **getter** e **setter** para acesso ao campo **nome**;
4. Crie um novo campo público, estático, do tipo **int** e de nome **contadorDeAlunos**;
5. Salve e compile sua classe.

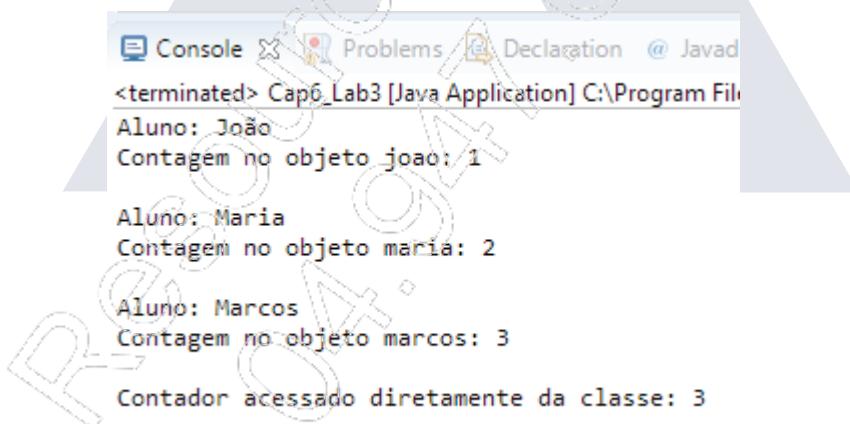
B - Criando a classe Cap6_Lab3 para testar a classe Aluno

1. Crie uma classe chamada **Cap6_Lab3** e garanta que o método **main** seja criado nessa classe;
2. Instancie três objetos **Aluno** com os nomes: “**joao**”, “**maria**” e “**marcos**”;

As ações descritas nos passos 3 a 6 devem ser repetidas, sequencialmente, para cada objeto instanciado.

3. Modifique o valor do atributo **nome da classe** para ficar igual ao do objeto (“**João**”, “**Maria**” ou “**Marcos**”), utilizando o método **setter** apropriado;
4. Incremente a variável **contadorDeAlunos**, usando como referência o identificador do objeto criado;
5. Invoque o método **imprimeAluno()** do respectivo objeto;
6. Imprima na tela o valor atual da variável estática **contadorDeAlunos** a partir do objeto atual;
7. Imprima na tela o valor da variável estática **contadorDeAlunos**, acessando-a por meio do nome da classe, da seguinte forma: **Aluno.contadorDeAlunos**;
8. Execute o programa.

A saída deverá ser como a seguinte:

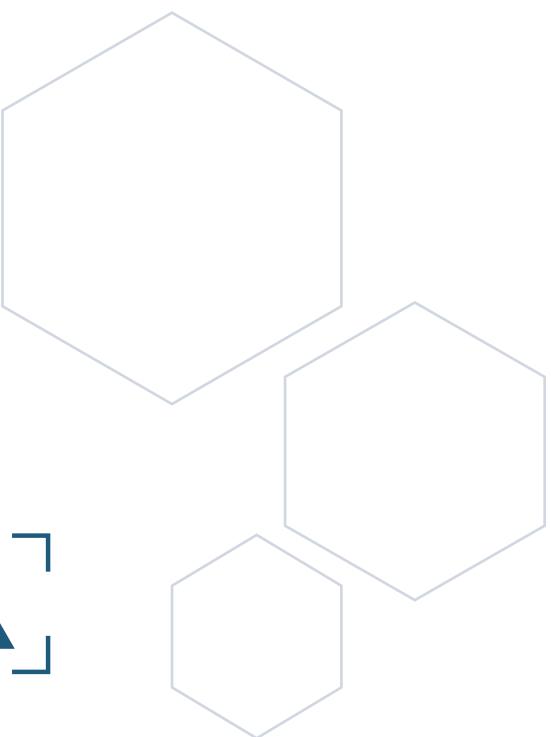
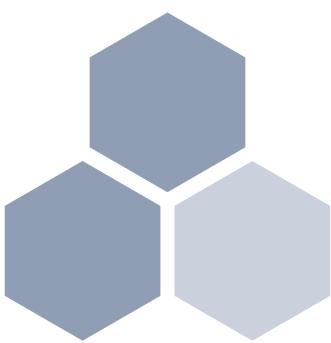


```
Console Problems Declaration @ Javad
<terminated> Cap6_Lab3 [Java Application] C:\Program File
Aluno: João
Contagem no objeto joao: 1
Aluno: Maria
Contagem no objeto maria: 2
Aluno: Marcos
Contagem no objeto marcos: 3
Contador acessado diretamente da classe: 3
```


7

Construtores

- ◆ Construtor padrão;
- ◆ Considerações sobre os construtores.



7.1. Introdução

Construtores são membros de classe cuja sintaxe é muito parecida com a de um método e destinam-se a construir instâncias da classe a que pertencem. Podem ser utilizados para inicializar valores ou executar tarefas no momento da criação do objeto. Para tanto, reservam espaço na memória, o qual será usado na manipulação de objetos. Além disso, possibilitam a criação de objetos mais complexos por conter chamadas para outros métodos.

Quando você declara um construtor, ele deve ter o mesmo nome da classe em que se localiza. Lembre-se que uma classe pode não conter nenhum construtor declarado, mas também pode conter quantos construtores declarados forem necessários, ou seja, os construtores podem ser sobre carregados da mesma forma que os métodos.

Os construtores são acionados apenas uma vez durante a vida de um objeto: no momento de sua criação. O comando adequado que efetivamente executa o construtor é o **new**. Depois desse momento, o código do construtor para esse objeto passa a ser inacessível.

Dependendo da necessidade, parâmetros podem ser recebidos pelo construtor.

7.2. Construtor padrão

Em toda classe Java, caso não seja definido qualquer construtor customizado, o próprio compilador Java se encarrega de criar um construtor, denominado construtor padrão: aquele que não recebe nenhum parâmetro e não apresenta nenhum código em seu interior. Esse construtor realiza, por padrão, a inicialização de todos os campos de sua classe com valores **default**: 0 para tipos **int**, 0.0 para tipos de **ponto flutuante**, **false** para **boolean**, \u0000 para **char** e **null** para objetos em geral.

Se você estiver trabalhando com classes simples, o construtor padrão definido pelo compilador do Java é muitas vezes eficaz. Já para classes mais complexas, o ideal é definir métodos construtores de maneira explícita.

Você pode identificar um construtor padrão como aquele cuja lista de parâmetros formais está vazia, conforme exemplo a seguir:

```

1
2 public class Calcado {
3     // este é o método construtor padrão da classe
4     public Calcado(){
5         // as instruções aqui serão opcionais
6     }
7
8 }

```

```

1
2 public class CriaCalcado {
3     public static void main(String args[]){
4         Calcado sandalia = new Calcado(); // instanciação ok
5     }
6 }

```

A partir do momento em que você define um construtor, o construtor padrão deixa de ser automaticamente criado pelo compilador Java. Quando uma nova instância da classe for feita, é necessário sempre informar os parâmetros exigidos pelo construtor definido anteriormente. É claro que sempre é possível, nesses casos, criar o construtor padrão de forma idêntica àquele criado automaticamente.

```

1
2 public class Calcado {
3     // agora este é o método construtor da classe
4     // exigindo parâmetros no ato da instanciação
5     public Calcado(String cor, int tamanho){
6         // este é o local onde entraria o tratamento dos parâmetros recebidos
7     }
8 }

```

```

1
2 public class CriaCalcado {
3     public static void main(String args[]){
4         Calcado sandalia = new Calcado(); // instanciação erro
5     }
6 }

```

Para que o processo que você acabou de ver seja bem-sucedido, proceda da seguinte forma:

```
Calcado sandalia = new Calcado("Preta", 36); // instanciação ok
```

A seguir, temos um exemplo de uso do construtor:

The screenshot shows a Java development environment with two open files. The top window is titled 'Carro.java' and contains the following code:

```
1 public class Carro {  
2     private String modelo;  
3     private int ano;  
4  
5     public Carro(String modelo, int ano){  
6         this.modelo = modelo;  
7         this.ano = ano;  
8     }  
9  
10    public void mostrar(){  
11        System.out.println(modelo + ano);  
12    }  
13}  
14}
```

The bottom window is titled 'ExemploUsoConstrutor.java' and contains the following code:

```
1 public class ExemploUsoConstrutor {  
2     public static void main(String args[]){  
3         Carro car = new Carro("Fiat Uno", 2000);  
4         car.mostrar();  
5     }  
6 }  
7 }
```

É importante ressaltar que um construtor padrão possui o mesmo modificador de acesso da classe e não possui argumentos. Ele inclui, ainda, uma chamada sem argumentos ao construtor da superclasse (`super()`). O conceito de superclasse é uma referência ao conteúdo de herança que será abordado nos próximos capítulos.

Um código simples, como o exibido adiante, não pode ser compilado sem a presença de um construtor padrão na classe **Calcado**:

The screenshot shows a Java development environment with two open files. The top window is titled 'Calcado.java' and contains the following code:

```
1 public class Calcado {  
2  
3     public Calcado(String c){  
4     }  
5  
6 }  
7  
8 }
```

The bottom window is titled 'Sandalia.java' and contains the following code:

```
1 public class Sandalia extends Calcado {  
2  
3 }  
4 }
```

A red arrow points to a yellow status bar at the bottom of the interface, which displays the message: "Implicit super constructor Calcado() is undefined for default constructor".

7.3. Considerações sobre os construtores

Para compreender adequadamente os construtores, é importante que você conheça as regras descritas a seguir:

- Os construtores podem utilizar qualquer modificador de acesso, incluindo o **private**. O modificador **private** (construtor privado) determina que apenas o código dentro da própria classe está capacitado a instanciar um objeto desse tipo. Uma utilização prática de construtores privados é a chamada deste construtor por outros construtores da mesma classe para execução de tarefas comuns a eles;
- Os construtores não devem possuir um tipo de retorno;
- Um método com o mesmo nome da classe é válido, mas isso não fará deste método um construtor. A presença de um tipo de retorno continuará caracterizando-o como um método;
- Se você já tiver inserido algum construtor com argumentos no código da classe e quiser ter um construtor sem argumentos, ele não poderá ser fornecido pelo compilador, você deverá codificá-lo manualmente (podendo deixar seu corpo vazio);
- A chamada a um construtor sobrecarregado (**this()**) ou ao construtor da superclasse (**super()**) deve ser a primeira instrução de todo construtor;
- Classes abstratas possuem construtores (assunto a ser estudado em capítulo posterior);
- Interfaces, por definição, não possuem construtores (assunto a ser estudado em capítulo posterior).

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

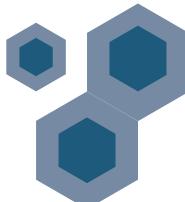
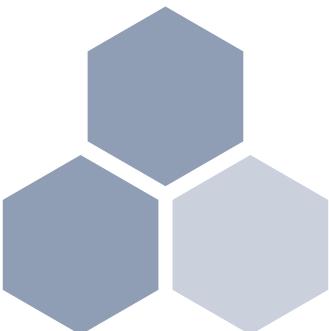
- Construtores são membros de classe cuja sintaxe é muito parecida com a de um método e destinam-se a construir instâncias da classe a que pertencem. Podem ser utilizados para inicializar valores ou executar tarefas no momento da criação do objeto. Para tanto, reserva espaço na memória, o qual será usado na manipulação de objetos. Além disso, possibilita a criação de objetos mais complexos por conter chamadas para outros métodos;
- Quando você declara um construtor, ele deve ter o mesmo nome da classe em que se localiza. Lembre-se que uma classe pode conter nenhum ou quantos construtores declarados forem necessários, ou seja, os construtores podem ser sobre carregados da mesma forma que os métodos;
- Para compreender adequadamente os construtores, você deve conhecer algumas regras descritas neste capítulo.



7

Construtores

Teste seus conhecimentos



1. O que é um construtor?

- a) Uma estrutura responsável por construir objetos com valores aleatórios.
- b) Uma estrutura responsável por construir objetos com valores determinados.
- c) Uma estrutura responsável por inicializar os atributos de uma classe.
- d) Uma estrutura responsável por construir objetos nulos.
- e) Nenhuma das alternativas anteriores está correta.

2. Qual das alternativas a seguir não está correta?

- a) Os construtores não devem possuir um tipo de retorno.
- b) Os construtores podem utilizar qualquer modificador de acesso, incluindo o private.
- c) Um método com o mesmo nome da classe é válido, mas isso não fará deste método um construtor.
- d) Os construtores não podem ter o mesmo nome da classe.
- e) Nenhuma das alternativas anteriores está correta.

3. O que é o construtor padrão?

- a) É um construtor que o próprio compilador de Java cria sempre que for definido explicitamente um método construtor mais sofisticado para a classe.
- b) É um construtor que o próprio compilador de Java cria caso não seja definido explicitamente um método construtor mais sofisticado para a classe.
- c) É um construtor que o próprio compilador de Java cria caso não seja definida explicitamente uma classe construtora.
- d) É um construtor que o próprio compilador de Java cria sempre que for definido explicitamente um atributo em uma classe.
- e) Nenhuma das alternativas anteriores está correta.

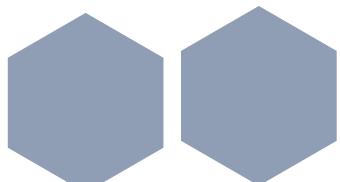


7

Construtores



Mãos à obra!



Laboratório 1

A – Criando a classe Cadastro com múltiplos construtores

1. Crie uma classe com o nome **Cadastro** e declare os atributos privados **nome**, **sobrenome** e **idade**;
2. Crie os assessores **getter** e **setter** para cada atributo;
3. Insira o construtor padrão da classe;
4. Crie um construtor que receba os parâmetros **nome** e **sobrenome** e atribua os valores desses parâmetros aos atributos da classe;
5. Crie outro construtor que receba os parâmetros **nome**, **sobrenome** e **idade** e atribua os valores desses parâmetros aos atributos da classe;
6. Crie um método chamado **mostrar** que exiba na tela o valor de cada atributo da classe;
7. Salve e compile a classe.

B – Testando a classe Cadastro

1. Crie uma classe com o nome **Cap7_Lab1** e insira a estrutura básica de um programa Java;
2. Crie três objetos da classe **Cadastro**. No primeiro, use o construtor padrão, no segundo, passe duas **strings** como parâmetros no construtor e, no terceiro, passe duas **strings** e um **int** como parâmetros;
3. Use o método **mostrar** da classe **Cadastro** para exibir os valores dos atributos de cada objeto;
4. Compile e execute o programa.

O resultado deve ser como o seguinte:

```
Console ×
<terminated> Cap7_Lab1 [Java Application]
Nome: null
Sobrenome: null
Idade: 0

Nome: Claudio
Sobrenome: Abreu
Idade: 0

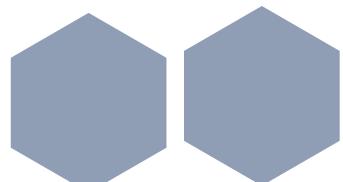
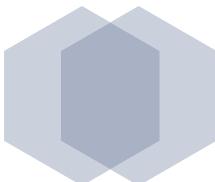
Nome: Lourdes
Sobrenome: Souza
Idade: 40
```




7

Construtores

Projeto Prático – Fase 2



Pacotes, construtores e métodos

Nesta fase, serão desenvolvidos os fluxos principais para implementação das funcionalidades do sistema baseados nos diagramas de sequência apresentados.

Como dados de entrada, serão apresentados diagramas de sequência do sistema.

Atividades

1. Definir as camadas (pacotes) da aplicação:

- Definir o pacote base da aplicação (**br.com.impacta.prateleiradigital**);
- Definir os seguintes subpacotes:
 - **apresentacao**: Comportará classes de interação com o usuário final;
 - **controle**: Comportará classes de controle de fluxos das informações e regras de negócio;
 - **persistencia**: Comportará classes que interagem com a base de dados;
 - **testes**: Comportará classes que testarão o sistema em tempo de desenvolvimento;
 - **utilitario**: Comportará classes não relacionadas aos requisitos do sistema, mas que apoiam uma ou mais classes de outros pacotes;
 - **negocio**: Comportará classes que representam os conceitos relacionados com o sistema. Neste pacote se encontrará a classe que representa um filme.

2. Implementar o fluxo de informações de acordo com os diagramas de sequência apresentados, definindo os métodos, porém sem implementar ainda a persistência dos dados:

- Utilizar, como entrada de dados na camada de apresentação, a classe **Scanner** para capturar as entradas do usuário final via console.

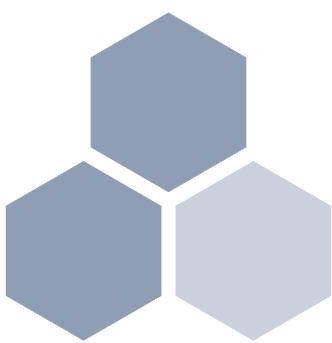
3. Definir dois construtores para a classe **Filme**:

- Construtor default;
- Construtor que recebe valores para todos os seus atributos.

8

Arrays

- Tipos de array;
- Acessando elementos de um array;
- Modos de inicializar e construir um array;
- Passando um array como parâmetro;
- Array de argumentos.



8.1. Introdução

Os arrays são variáveis indexadas. Isto quer dizer que você pode definir uma variável cujos elementos são referenciados por um índice. Assim, os arrays são definidos como objetos em cujo conteúdo encontram-se diversos valores de um mesmo tipo. Os tipos podem ser básicos ou tipos construídos.

O índice é um valor inteiro responsável por determinar uma posição dentro de um array. Podemos dizer, então, que elementos de um mesmo tipo são agrupados em arrays. Os arrays podem armazenar variáveis de tipos primitivos ou de referência.

8.2. Tipos de array

Um array pode ter uma ou mais dimensões, podendo ser classificado nos tipos descritos adiante.

8.2.1. Array unidimensional

A construção de arrays unidimensionais, também chamados de vetores, requer a utilização da palavra-chave `new`, a definição do tipo do array e seu tamanho, que é um número inteiro que deve estar entre colchetes.

Assim, em Java, os arrays são instâncias de objetos, os quais serão definidos de acordo com o tipo de dado que será manipulado.

A sintaxe de um array unidimensional é a seguinte:

```
<modificadores> <tipo> <nome>[ ] = new <tipo>[<dimensão>];
```

Em que:

- `<tipo>` se refere ao tipo de dado primitivo ou classe;
- `<nome>` é o nome atribuído ao array;
- `<dimensão>` se refere ao tamanho do array em número de elementos.

O exemplo a seguir mostra como construir um único objeto de array do tipo **Ponto**:

```
Ponto pontos[] = new Ponto[8];
```

Este código insere um único objeto na pilha, que é o array atribuído à variável de referência **pontos**. Embora esse objeto contenha oito variáveis de referência **pontos**, não há qualquer objeto **pontos** que tenha sido criado ou, ainda, atribuído a tais variáveis. É de extrema importância que você saiba identificar a quantidade de objetos que serão adicionados à pilha assim que o código ou a instrução `for` executado.

Ao criar um array, é necessário atribuir um tamanho a ele, uma vez que a JVM precisa ter essa informação a fim de alocar o espaço adequado. Sendo assim, o exemplo a seguir é inválido, uma vez que o tamanho não é especificado:

```
int qtdClientes[] = new int[]; // não será compilado, precisa de um tamanho
✖ Variable must provide either dimension expressions or an array initializer
```

Array com tipos construídos

Os arrays podem armazenar tipos primitivos e também tipos construídos. Veja, no exemplo a seguir, um array com tipos construídos:

```

Pessoa.java
1  public class Pessoa {
2      double altura;
3      double peso;
4  }
5

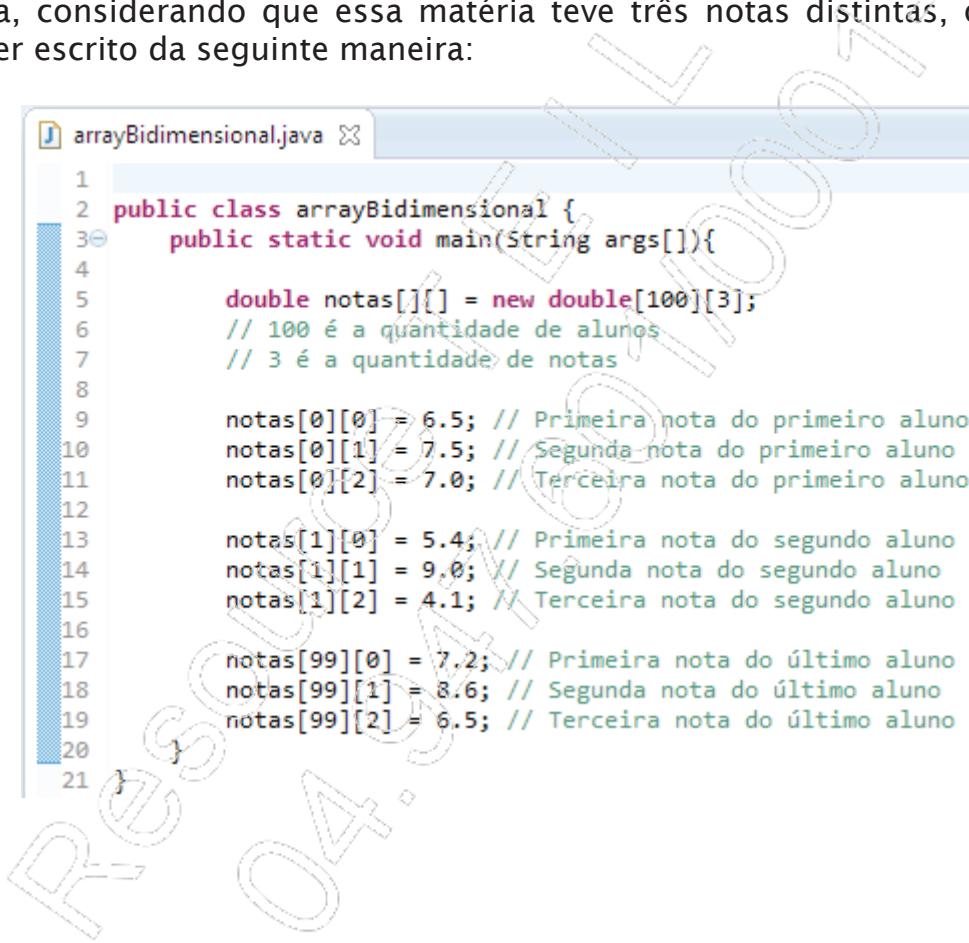
Cadastro.java
1  public class Cadastro {
2      public static void main(String args[]){
3          Pessoa pessoas[] = new Pessoa[2]; // declara que pessoas vai armazenar 2 elementos do tipo Pessoa
4          pessoas[0] = new Pessoa(); // cria instância para ocorrência 0
5          pessoas[0].altura = 1.68;
6          pessoas[0].peso = 55;
7
7          pessoas[1] = new Pessoa(); // cria instância para ocorrência 1
8          pessoas[1].altura = 1.77;
9          pessoas[1].peso = 60;
10     }
11 }
```

8.2.2. Array bidimensional

Um array bidimensional é aquele com dois índices e que possibilita o armazenamento de valores na forma de matrizes. Apesar de a linguagem Java não suportar arrays bidimensionais, é possível, entretanto, criar um array de arrays, o qual deve ser declarado da seguinte maneira:

```
<modificadores> <tipo> <nome>[][] = new <tipo> [<indice1>][<indice2>];
```

O exemplo a seguir demonstra a utilização de arrays bidimensionais. Para realizar o cadastro das notas finais de 100 alunos em uma determinada matéria, considerando que essa matéria teve três notas distintas, o código deve ser escrito da seguinte maneira:



```
J arrayBidimensional.java X
1
2 public class arrayBidimensional {
3     public static void main(String args[]){
4
5         double notas[][] = new double[100][3];
6         // 100 é a quantidade de alunos
7         // 3 é a quantidade de notas
8
9         notas[0][0] = 6.5; // Primeira nota do primeiro aluno
10        notas[0][1] = 7.5; // Segunda nota do primeiro aluno
11        notas[0][2] = 7.0; // Terceira nota do primeiro aluno
12
13        notas[1][0] = 5.4; // Primeira nota do segundo aluno
14        notas[1][1] = 9.0; // Segunda nota do segundo aluno
15        notas[1][2] = 4.1; // Terceira nota do segundo aluno
16
17        notas[99][0] = 7.2; // Primeira nota do último aluno
18        notas[99][1] = 8.6; // Segunda nota do último aluno
19        notas[99][2] = 6.5; // Terceira nota do último aluno
20
21    }
}
```

8.2.3. Array multidimensional

Os arrays bidimensionais são mais comuns, mas você também pode criar arrays de várias dimensões. Para que uma variável array multidimensional seja declarada, basta usar um novo par de colchetes para cada índice adicional, conforme a sintaxe a seguir:

```
<modificadores> <tipo> <nome>[][][] = new <tipo>
    [<indice1>][<indice2>][<indice3>];
```

O exemplo a seguir demonstra mais claramente como construir arrays multidimensionais. Para realizar o cadastro de uma sequência de notas finais de 100 alunos em 4 matérias, considerando que cada aluno tem 3 notas distintas para cada matéria, o código deve ser escrito da seguinte maneira:

```
arrayMultidimensional.java
1
2 public class arrayMultidimensional {
3     public static void main(String args[]){
4
5         double notas[][][] = new double[100][4][3];
6         // 100 é a quantidade de alunos
7         // 4 é a quantidade de matérias
8         // 3 é a quantidade de notas
9
10        notas[0][0][0] = 6.5; // Primeira nota do primeiro aluno para a primeira matéria
11        notas[0][0][1] = 7.5; // Segunda nota do primeiro aluno para a primeira matéria
12        notas[0][0][2] = 7.0; // Terceira nota do primeiro aluno para a primeira matéria
13
14        notas[0][1][0] = 6.5; // Primeira nota do primeiro aluno para a segunda matéria
15        notas[0][1][1] = 7.5; // Segunda nota do primeiro aluno para a segunda matéria
16        notas[0][1][2] = 7.0; // Terceira nota do primeiro aluno para a segunda matéria
17
18        notas[99][3][0] = 7.2; // Primeira nota do último aluno para a última matéria
19        notas[99][3][1] = 8.6; // Segunda nota do último aluno para a última matéria
20        notas[99][3][2] = 6.5; // Terceira nota do último aluno para a última matéria
21    }
22 }
```

8.3. Acessando elementos de um array

Para que você possa fazer uma referência a um vetor criado, é preciso definir o índice em que deseja realizar uma determinada operação. Dessa forma, ao alocar um array, qualquer um de seus elementos pode ser acessado, bastando, para isso, determinar entre colchetes o índice desejado.

A referência a um vetor pode ser feita tanto para incluir valores quanto para obter valores e utilizá-los em outra operação. Destacamos que os índices de um array são sempre iniciados em zero.

É importante que você tenha em mente como inicializar um array, ou seja, como adicionar elementos no array, que podem ser valores primitivos ou objetos.

Um array de objetos não armazena propriamente objetos; o que ele armazena de fato são referências a esses objetos. Sendo assim, um array de três strings, na verdade é um array de três referências a objetos **String**. Você deve saber se essas referências são nulas, ou seja, se não têm um objeto atribuído, ou se estão referenciando de fato objetos **String**.

Ao utilizar uma referência nula para chamar um método, ocorre a exceção **NullPointerException**.

Observe o exemplo a seguir, no qual é criado um array com três frutas. Nele, temos um objeto de array na pilha e três referências do tipo **Fruta** que são nulas:

```
Fruta alimento[] = new Fruta[3];
```

A seguir, criaremos alguns objetos do tipo **Fruta**, atribuindo os mesmos valores que determinam a posição de índice no array, o qual é referenciado por **alimento**. Veja:

```
alimento[0] = new Fruta();  
alimento[1] = new Fruta();  
alimento[2] = new Fruta();
```

Preste atenção em qualquer trecho de código que tente acessar um índice que não está dentro do intervalo de um array. A tentativa de acessar o índice 6 de um array de seis elementos é um exemplo típico. Isso não é possível, uma vez que o índice de um array inicia-se sempre em zero, ou seja, neste caso, o intervalo é de 0 a 5. Há circunstâncias em que é possível encontrar a tentativa de acesso a um índice por meio de um número negativo. Em ambas as situações, será lançada a exceção **ArrayIndexOutOfBoundsException**. Observe o exemplo a seguir:

The screenshot shows an IDE interface with two main panes. The top pane is a code editor titled "ExemploAcessoInvalido.java" containing the following code:

```
1 public class ExemploAcessoInvalido {  
2     public static void main(String args[]){  
3         int numeros[] = new int[6];  
4         numeros[6] = 3; // exceção em tempo de execução.  
5         // o último elemento está na posição de índice 5  
6  
7         int valores[] = new int[6];  
8         int indice = -5;  
9         valores[indice] = 3; // exceção em tempo de execução.  
10        // indice é um número negativo  
11    }  
12 }  
13 }  
14 }
```

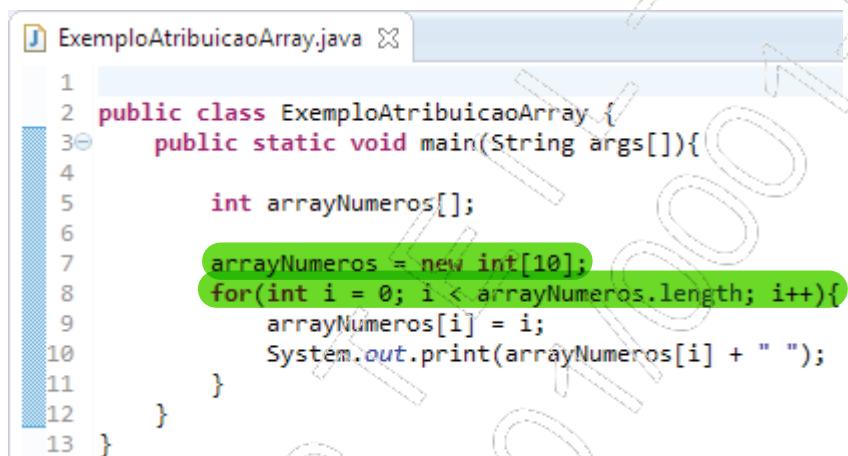
The bottom pane is a "Console" window showing the execution results:

```
Console Problems @ Javadoc Declaration  
<terminated> ExemploAcessoInvalido [Java Application] C:\Program Files\Java\jdk-9\bin\javaw.  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6  
at ExemploAcessoInvalido.main(ExemploAcessoInvalido.java:6)
```

8.3.1. Acesso aos elementos em um for tradicional

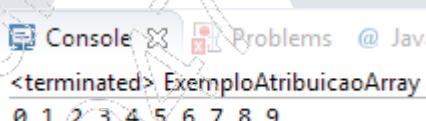
Os objetos de array possuem um único atributo público, `length`, cuja função é indicar a quantidade de elementos que o array contém, sem informar a respeito da inicialização de tais elementos.

O exemplo de código a seguir demonstra como inicializar os elementos de um array em um loop:



```
ExemploAtribuicaoArray.java
1
2 public class ExemploAtribuicaoArray {
3     public static void main(String args[]){
4
5         int arrayNumeros[];
6
7         arrayNumeros = new int[10];
8         for(int i = 0; i < arrayNumeros.length; i++){
9             arrayNumeros[i] = i;
10            System.out.print(arrayNumeros[i] + " ");
11        }
12    }
13 }
```

Depois de compilado e executado o código, o resultado será como exibido na figura a seguir:



```
Console Problems @ Java
<terminated> ExemploAtribuicaoArray
0 1 2 3 4 5 6 7 8 9
```

8.3.2. Acesso aos elementos usando enhanced for (foreach)

Outra forma de percorrer o array ou qualquer outra estrutura de coleção de objetos é utilizando a estrutura chamada **enhanced for**, mais conhecido no mundo computacional como **foreach**.

Neste caso, a estrutura percorre todos os elementos do array de forma que o desenvolvedor não tenha necessidade de manipular o índice do array como ocorre na utilização do **for** tradicional. A cada iteração, a variável local recebe uma cópia do elemento corrente.

É importante observar que não há como alterar a composição do array percorrido (apagar ou substituir elementos), uma vez que não temos posse do índice do elemento na iteração.

Abaixo segue um código que utiliza o **enhanced for** para percorrer um array:

```
ExemploForeach.java
1
2 public class ExemploForeach {
3
4     public static void main(String[] args) {
5
6         int[] numeros = new int[6];
7
8         numeros[0] = 10;
9         numeros[1] = 20;
10        numeros[2] = 30;
11        numeros[3] = 40;
12        numeros[4] = 50;
13        numeros[5] = 60;
14
15        //percorrendo e exibindo os elementos
16        for(int num: numeros) {
17            System.out.println(num);
18        }
19    }
20
21 }
```

8.4. Modos de inicializar e construir um array

A **inicialização** de um array refere-se à atribuição de valores aos seus elementos, e sua **construção** refere-se à instanciação do objeto de array. Tendo isso em mente, observe os exemplos desenvolvidos a seguir.

8.4.1. Por meio de uma única instrução

A **inicialização** e a **construção** de arrays são tarefas que podem ser realizadas em uma única instrução. Para isso, você pode utilizar dois atalhos específicos de arrays. Um deles será utilizado com a finalidade de criar, declarar e inicializar um array.

Por meio de uma instrução como a descrita a seguir, é possível declarar, criar e inicializar um array. Somente a sexta linha dessa instrução realiza quatro tarefas distintas:

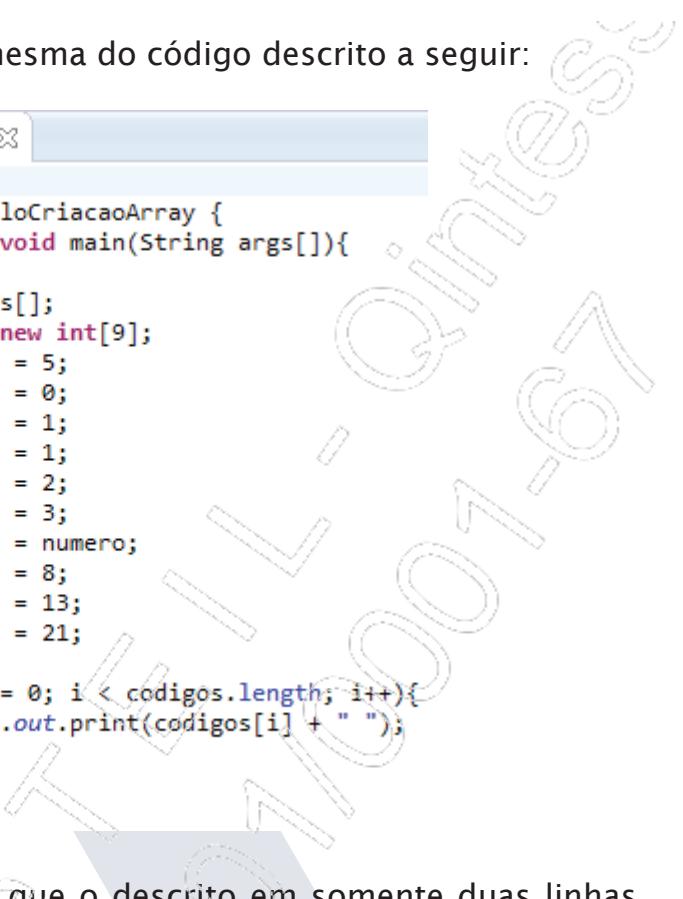
```
1  public class ExemploSimplesCriacaoArray {
2      public static void main(String args[]){
3          int numero = 5;
4
5          int codigos[] = {0, 1, 1, 2, 3, numero, 8, 13, 21};
6
7          for(int i = 0; i < codigos.length; i++){
8              System.out.print(codigos[i] + " ");
9          }
10     }
11 }
12 }
```

As tarefas realizadas pela sexta linha são:

- Criação de um array de nove elementos do tipo inteiro;
- Declaração de uma variável de referência do tipo **array**, de inteiros, a qual possui o nome **codigos**;
- Criação do objeto **array** que é atribuído a **codigos** e atribuição dos nove valores **0, 1, 1, 2, 3, 5, 8, 13** e **21** a esse array;
- Atribuição do objeto de array a **codigos**.

Observe que o tamanho do array é definido pela quantidade de itens que são colocados entre chaves.

A função da instrução anterior é a mesma do código descrito a seguir:

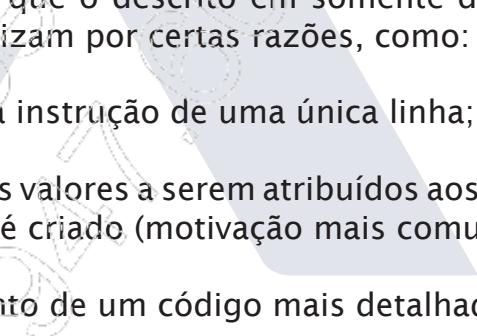


```
J ExemploCriacaoArray.java X
1
2 public class ExemploCriacaoArray {
3     public static void main(String args[]){
4
5         int codigos[];
6         codigos = new int[9];
7         int numero = 5;
8         codigos[0] = 0;
9         codigos[1] = 1;
10        codigos[2] = 1;
11        codigos[3] = 2;
12        codigos[4] = 3;
13        codigos[5] = numero;
14        codigos[6] = 8;
15        codigos[7] = 13;
16        codigos[8] = 21;
17
18        for(int i = 0; i < codigos.length; i++){
19            System.out.print(codigos[i] + " ");
20        }
21    }
22 }
```

Este código é muito mais longo do que o descrito em somente duas linhas, mas diversos desenvolvedores o utilizam por certas razões, como:

- Desconhecimento a respeito da instrução de uma única linha;
- Desconhecimento a respeito dos valores a serem atribuídos aos elementos no momento em que um array é criado (motivação mais comum);
- Preferência pelo desenvolvimento de um código mais detalhado.

Depois de compilado e executado qualquer um dos códigos anteriores, o resultado será como o exibido na figura a seguir:



```
Console X Problems @
<terminated> ExemploCriacaoArray
0 1 1 2 3 5 8 13 21
```

Em vez de tipos primitivos, também é possível criar um exemplo utilizando referências a objetos. Observe o exemplo a seguir e, posteriormente, sua explicação:

```
Usuario usuarioInfo = new Usuario("Ana");
Usuario novosUsuarios[] = {usuarioInfo, new Usuario("Ernesto"), new Usuario("Alfredo")};
```

Nesse código, temos a seguinte situação:

- A criação do array de nome **novosUsuarios**, que conterá referências para objetos do tipo **Usuario**. Seu tamanho corresponde a três elementos;
- O objeto **Usuario**, referenciado pela variável **usuarioInfo**, é atribuído ao primeiro elemento do array;
- A criação de dois novos objetos **Usuario** com os atributos **nome** atribuídos respectivamente via construtor: **Ernesto** e **Alfredo**;
- As duas instâncias que acabam de ser criadas são atribuídas aos dois últimos elementos da variável de referência do array **novosUsuarios**.

Outro exemplo, demonstrado a seguir, refere-se a um atalho contendo arrays bidimensionais:

```
ExemploSimplesArrayBidimensional.java
1  public class ExemploSimplesArrayBidimensional {
2      public static void main(String args[]){
3          int elementos[][] = {{0, 1, 1}, {3, 5, 8, 13}, {21, 34}};
4
5          for(int i = 0; i < elementos.length; i++){
6              for(int j = 0; j < elementos[i].length; j++){
7                  System.out.println(elementos[i][j]);
8              }
9          }
10     }
11 }
12 }
```

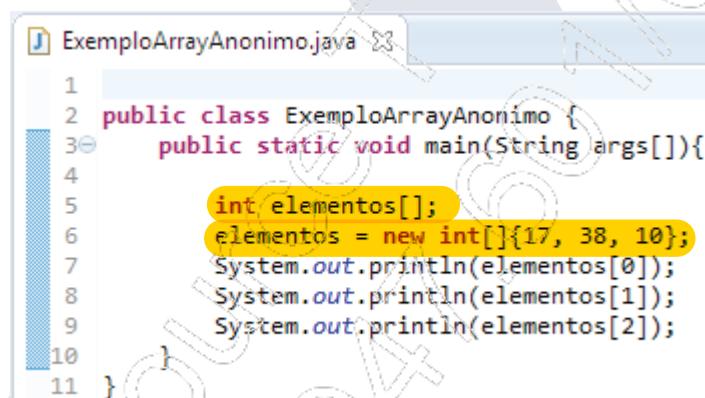
Neste código, temos as seguintes situações:

- Criação de quatro objetos na pilha;
- Construção do array que conterá elementos do tipo **int**. Ele contém outros objetos do tipo **array** e é o objeto que será atribuído à variável de referência **elementos**;
- O array **elementos** possui três elementos, os quais são representados pelos itens que estão entre as chaves externas, sendo que cada um deles é um array **int** cujo tamanho é determinado pela quantidade de itens presentes nas chaves internas, da seguinte forma:
 - O primeiro array tem tamanho igual a 3;
 - O segundo array tem tamanho igual a 4;
 - O terceiro array tem tamanho igual a 2.

- No total, até aqui são quatro objetos: um array, no qual estão contidos arrays **int**, e outros três arrays **int**, sendo que cada um de seus elementos refere-se a um valor do tipo **int**;
- Cada um dos itens que estão entre as chaves externas representa uma variável de referência a um array que conterá elementos do tipo **int**. Isto significa que os três arrays **int** construídos são atribuídos aos elementos do array **elementos**;
- Com os valores **int** presentes em suas chaves internas, os arrays foram inicializados.

8.4.2. Por meio de um array anônimo

A criação de um array anônimo é o segundo atalho para a construção e a inicialização de um array. Este atalho também permite que esse array seja atribuído a uma variável já declarada e que faça referência a outro array, como demonstrado no exemplo a seguir:



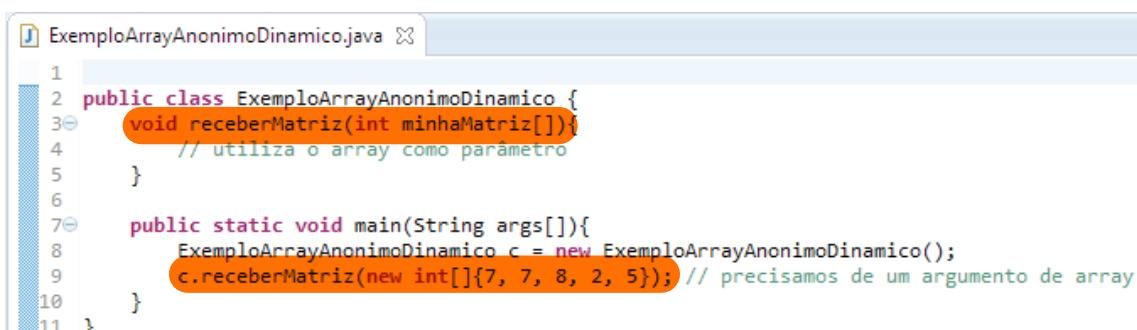
```

1  public class ExemploArrayAnonimo {
2      public static void main(String args[]){
3          int elementos[];
4          elementos = new int[]{17, 38, 10};
5          System.out.println(elementos[0]);
6          System.out.println(elementos[1]);
7          System.out.println(elementos[2]);
8      }
9  }

```

Este código permite criar um array **int** no qual estão contidos três elementos, bem como utiliza os valores 17, 38 e 10 para inicializar esses elementos. Além disso, atribui à variável de referência **elementos** um novo array.

Os procedimentos que você viu até aqui referem-se à criação de um array anônimo. Você verá no exemplo a seguir que os arrays que não são atribuídos a uma variável de referência podem ser utilizados na criação de um array **just-in-time**:



```

1  public class ExemploArrayAnonimoDinamico {
2      void receberMatriz(int minhaMatriz[]){
3          // utiliza o array como parametro
4      }
5
6
7      public static void main(String args[]){
8          ExemploArrayAnonimoDinamico c = new ExemploArrayAnonimoDinamico();
9          c.receberMatriz(new int[]{7, 7, 8, 2, 5}); // precisamos de um argumento de array
10     }
11 }

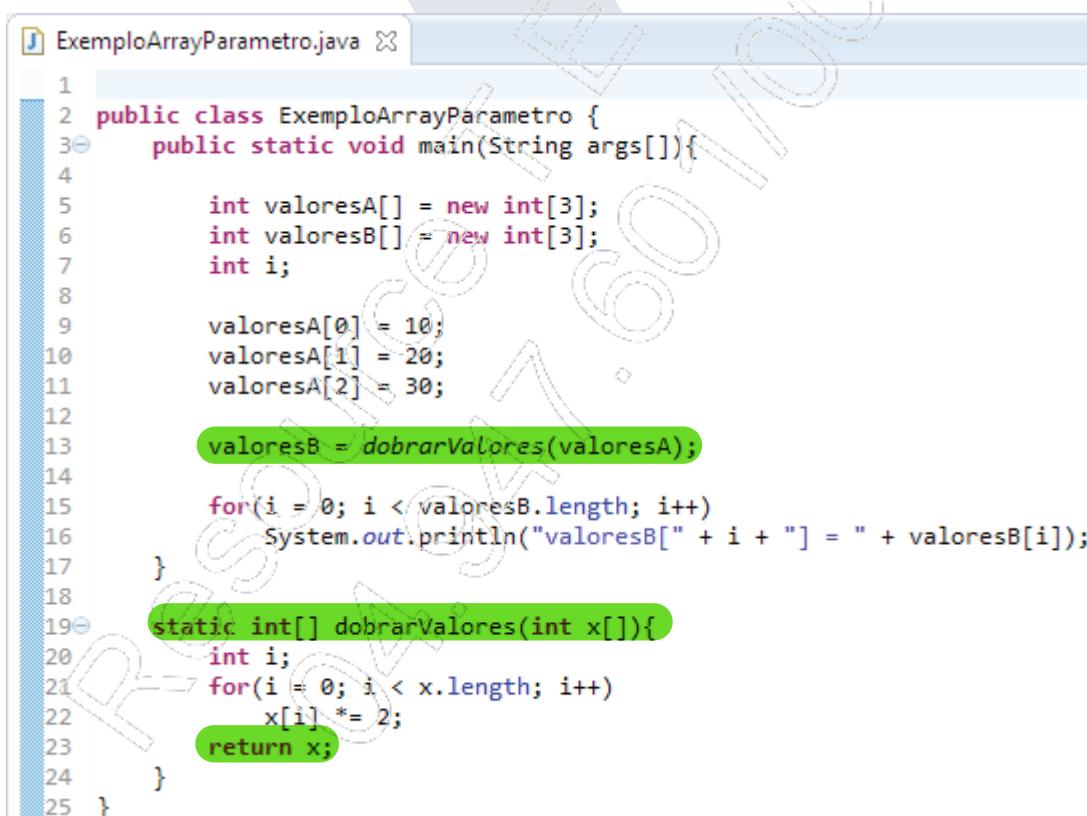
```

Um array **just-in-time** pode ser utilizado como argumento de um método, o qual, por sua vez, utiliza como parâmetro um array. Vale destacar que, quando você cria um array anônimo, o tamanho não deve ser determinado.

8.5. Passando um array como parâmetro

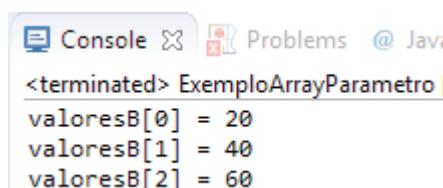
Um array pode ser passado e recebido como parâmetro. É importante destacar que o array original reflete quaisquer alterações realizadas nos valores do método que receber um array como parâmetro. Essa constatação provém da mesma regra já aplicada anteriormente com objetos em geral e comprova que arrays em Java são objetos como quaisquer outros, porém com sintaxe específica e diferenciada. Da mesma forma que os objetos, variáveis de referência para arrays são sempre passadas por valor.

Observe o exemplo a seguir, no qual o vetor das funções é passado e retornado:



```
J ExemploArrayParametro.java
1
2 public class ExemploArrayParametro {
3     public static void main(String args[]){
4
5         int valoresA[] = new int[3];
6         int valoresB[] = new int[3];
7         int i;
8
9         valoresA[0] = 10;
10        valoresA[1] = 20;
11        valoresA[2] = 30;
12
13        valoresB = dobrarValores(valoresA);
14
15        for(i = 0; i < valoresB.length; i++)
16            System.out.println("valoresB[" + i + "] = " + valoresB[i]);
17    }
18
19    static int[] dobrarValores(int x[]){
20        int i;
21        for(i = 0; i < x.length; i++)
22            x[i] *= 2;
23        return x;
24    }
25 }
```

O resultado será o seguinte:

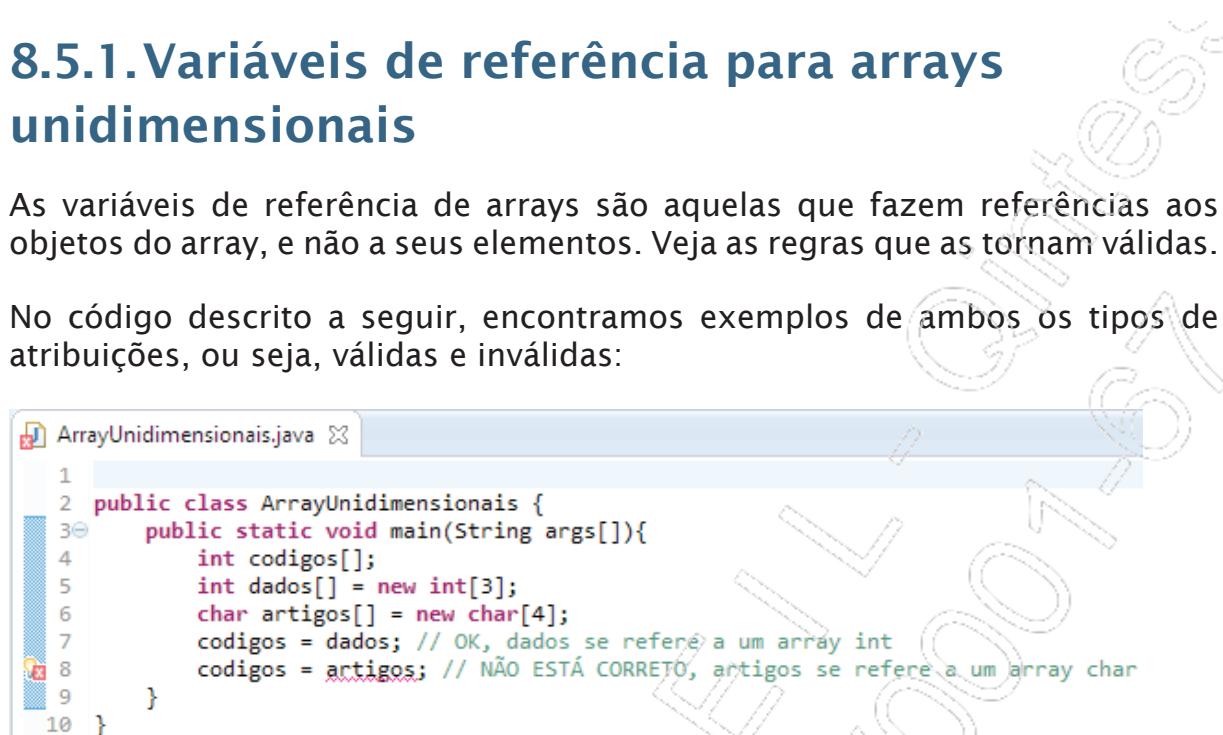


```
Console Problems @ Java
<terminated> ExemploArrayParametro
valoresB[0] = 20
valoresB[1] = 40
valoresB[2] = 60
```

8.5.1. Variáveis de referência para arrays unidimensionais

As variáveis de referência de arrays são aquelas que fazem referências aos objetos do array, e não a seus elementos. Veja as regras que as tornam válidas.

No código descrito a seguir, encontramos exemplos de ambos os tipos de atribuições, ou seja, válidas e inválidas:



```
ArrayUnidimensionais.java
1
2 public class ArrayUnidimensionais {
3     public static void main(String args[]){
4         int codigos[];
5         int dados[] = new int[3];
6         char artigos[] = new char[4];
7         codigos = dados; // OK, dados se refere a um array int
8         codigos = artigos; // NÃO ESTÁ CORRETO, artigos se refere a um array char
9     }
10 }
```

Para compreender o que ocorre com atribuições de referência de arrays de tipos primitivos, a regra é simples: somente é possível reatribuir arrays de um determinado tipo para outros arrays do mesmo tipo.

! Em razão de todos os arrays serem objetos, não é possível atribuir uma variável de referência de array a um tipo primitivo int.

8.5.2. Variáveis de referência para arrays multidimensionais

Neste item você verá a atribuição de variáveis de referência de arrays multidimensionais.

Como você já sabe, é possível atribuir um array a uma variável de referência de array já declarada. Porém, para isso, é preciso que tanto o array como a variável de referência estejam na mesma dimensão. Por conta disso, o código descrito a seguir é inválido:

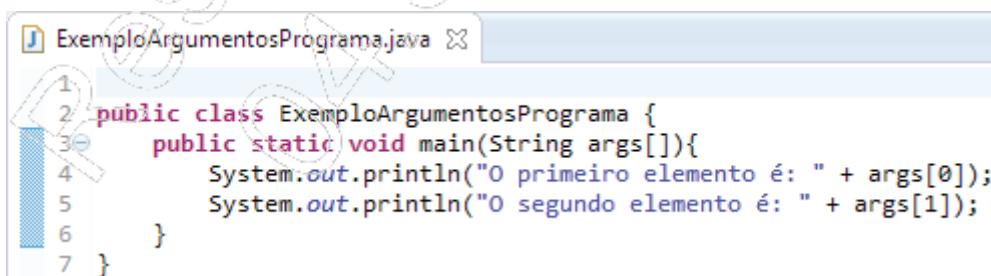
```
int codigos[];
int notas[][] = new int[4][];
codigos = notas; // NÃO É VÁLIDO, notas é um array
                  // bidimensional de arrays int
int dados[] = new int[8];
codigos = dados; // OK, dados é um array int
```

Como você observou neste último código, em que o array **int** é um array simples, não é possível atribuir um array bidimensional de arrays **int** a uma variável de referência de um array **int** que não seja bidimensional.

8.6. Array de argumentos

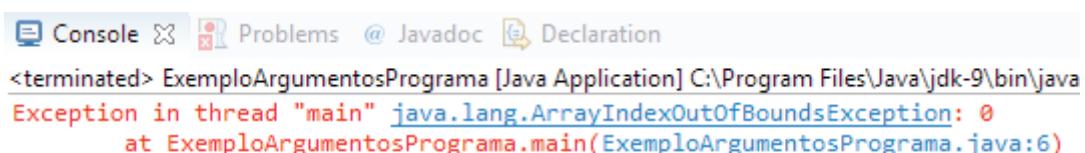
A Java Virtual Machine invoca o método principal, ou **main**, o qual deve empregar o parâmetro do array de **String**. Este último é responsável por alocar os argumentos que são enviados juntamente ao comando cuja função é executar o programa.

Veja o exemplo a seguir:



```
1  public class ExemploArgumentosPrograma {
2      public static void main(String args[]){
3          System.out.println("O primeiro elemento é: " + args[0]);
4          System.out.println("O segundo elemento é: " + args[1]);
5      }
6  }
```

Depois de compilar e executar o código anterior, a seguinte tela será exibida:



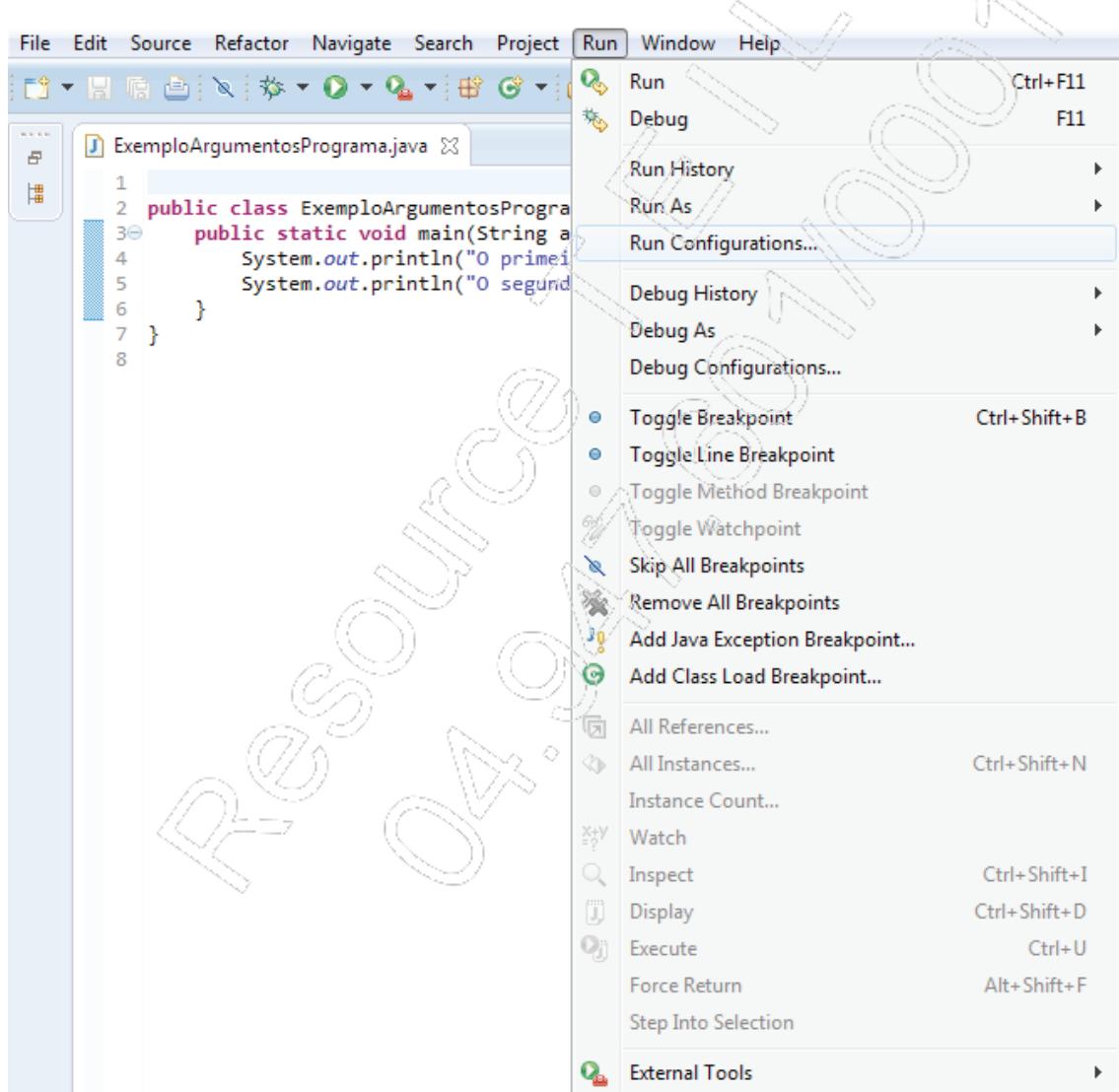
```
Console Problems @ Javadoc Declaration
<terminated> ExemploArgumentosPrograma [Java Application] C:\Program Files\Java\jdk-9\bin\java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at ExemploArgumentosPrograma.main(ExemploArgumentosPrograma.java:6)
```

Esse erro ocorre porque o array **args**, recebido pelo método **main** de seu programa, está vazio e, ao tentar acessar uma de suas posições, ocorre o erro de índice fora dos limites do array. Esse array tem o intuito de receber argumentos de linha de comando, passados pelo executor do programa, após o comando:

```
java <NomeDaClasse> <argumentos String separados por espaço>
```

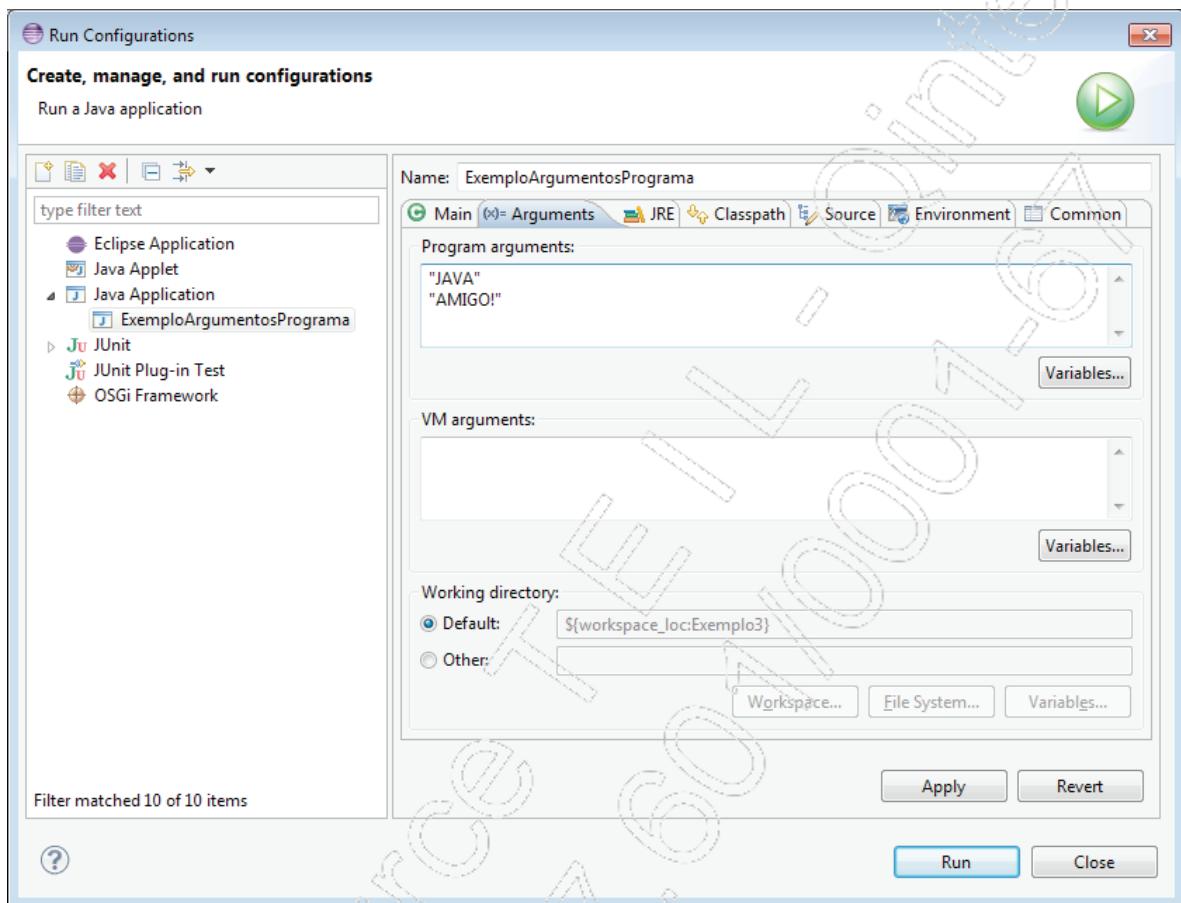
Veja como simular esse comportamento no Eclipse, de forma a passar argumentos no momento da execução do programa:

1. No menu **Run**, selecione a opção **Run Configurations...**:

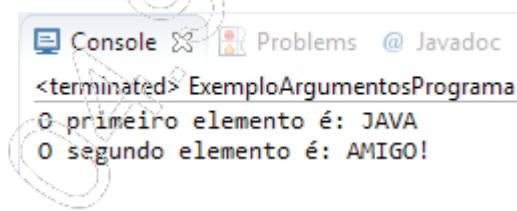


2. Uma janela com o mesmo nome será exibida. Observe se a classe **ExemploArgumentosPrograma** está selecionada à esquerda e escolha a aba **Arguments**;

3. Em **Program arguments**, digite “JAVA” “AMIGO!”, como mostra a imagem a seguir:



4. Após inserir os argumentos, clique no botão **Run**. O resultado será exibido no console:



O parâmetro do array de **String** não deve ser necessariamente nomeado como **arg** ou **args**, ele pode receber qualquer outro nome. Outro fato que deve ser considerado é que o array **args** é um simples array que foi passado para **main** pela Java Virtual Machine.

A quantidade de argumentos de linha de comando corresponde ao tamanho do array **args**. Veja:



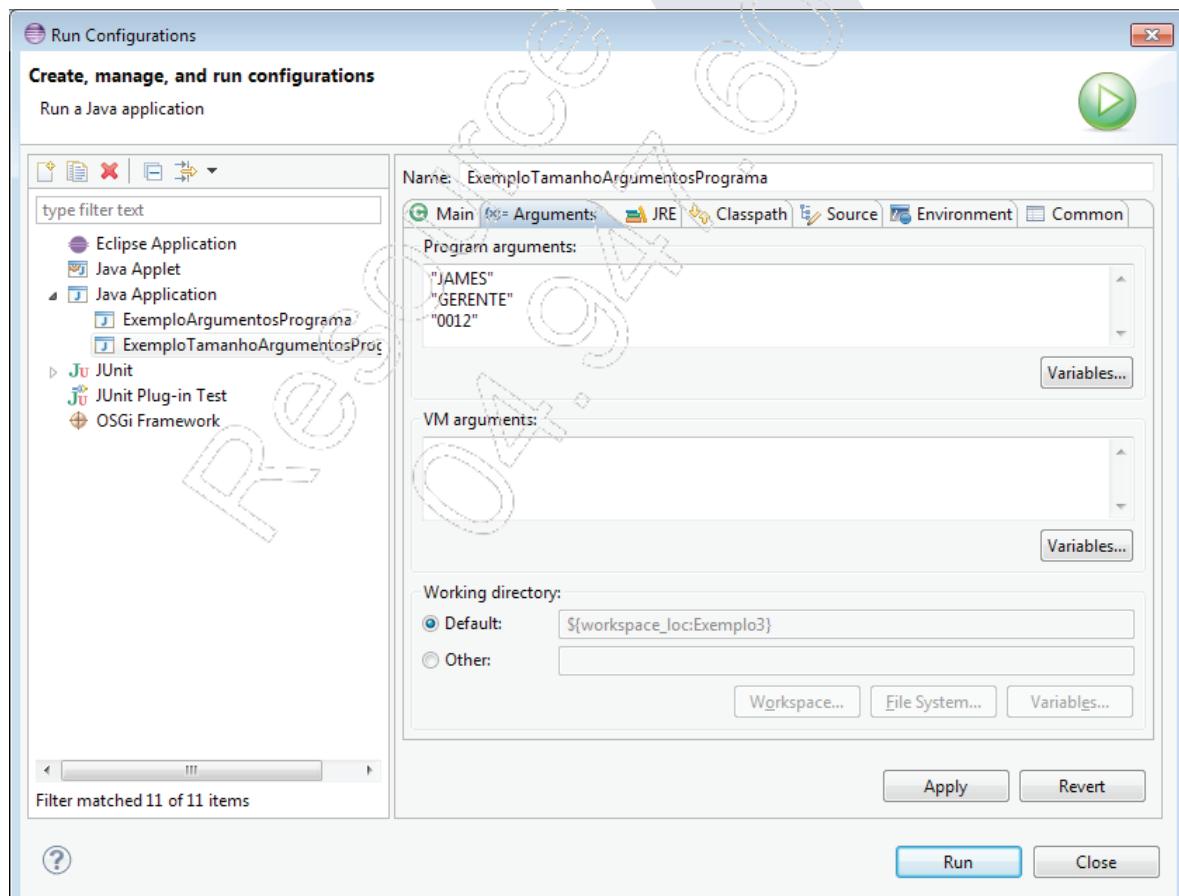
```

1  public class ExemploTamanhoArgumentosPrograma {
2      public static void main(String args[]){
3
4          if(args.length == 3){
5              System.out.println("Nome: " + args[0]);
6              System.out.println("Cargo: " + args[1]);
7              System.out.println("Código: " + args[2]);
8          }else{
9              System.out.println("Digite: [nome] [cargo] [codigo]");
10         }
11     }
12 }
13

```

1. Após compilar o código, selecione a opção **Run Configurations** no menu **Run**;

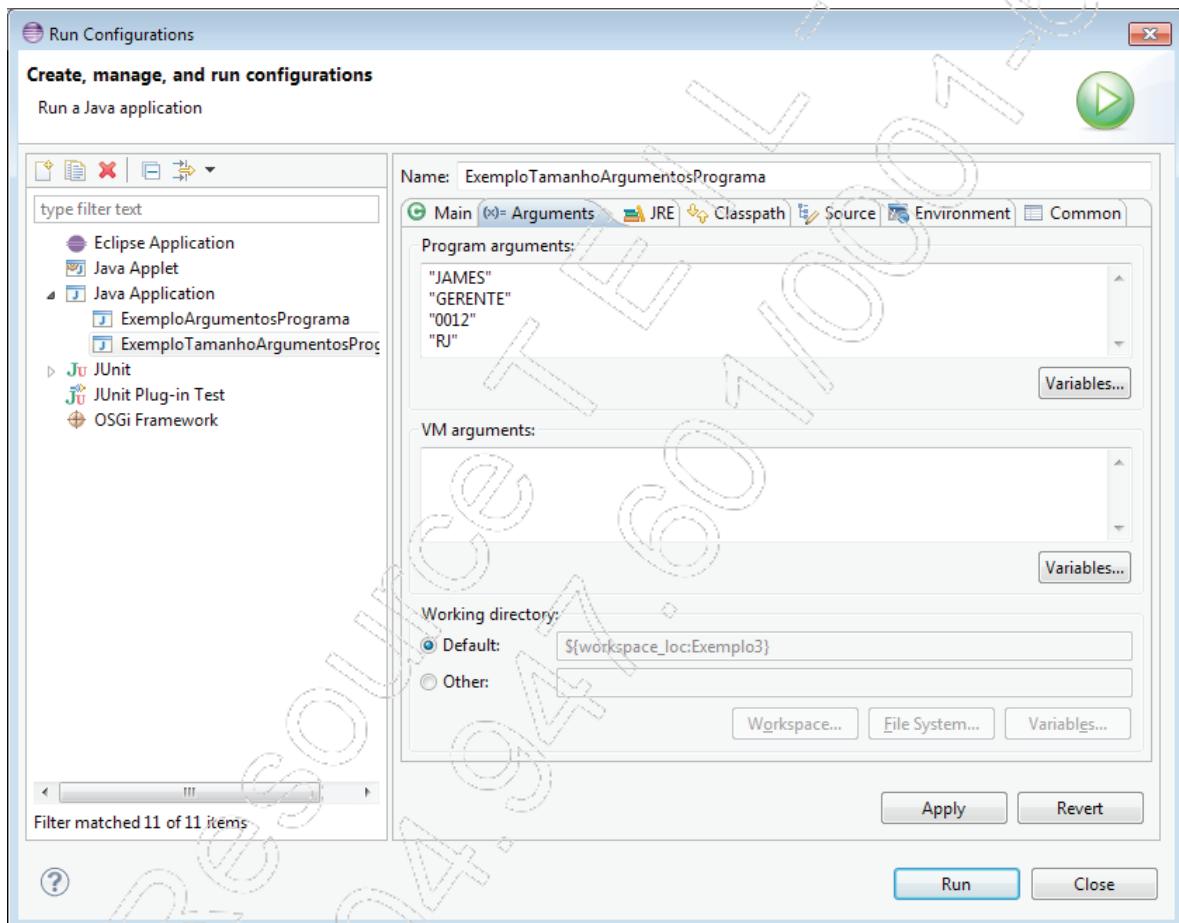
2. Na aba **Arguments**, da janela **Run Configurations**, insira os argumentos em **Program arguments**;



3. Em seguida, clique no botão Run. O resultado será exibido no console.

```
Console Problems Javadoc Declar
<terminated> ExemploTamanhoArgumentosPrograma
Nome: JAMES
Cargo: GERENTE
Código: 0012
```

A imagem a seguir ilustra a inserção de mais de três argumentos. Observe:



Ao executar o código, você terá o seguinte resultado:

```
Console Problems Javadoc Declar
<terminated> ExemploTamanhoArgumentosProgramma
Digite: [nome] [cargo] [codigo]
```

Esse código ilustra uma situação em que o comando **else** é executado e ocorre o encerramento do programa pela Virtual Machine, caso haja tentativa de acesso a algo que ultrapasse **length-1**.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

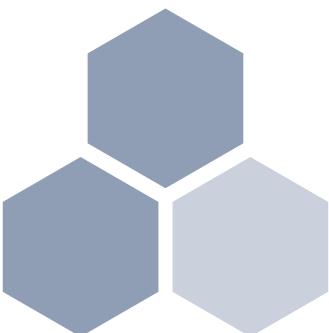
- Os arrays são variáveis indexadas. Isto quer dizer que você pode definir uma variável cujos elementos são referenciados por um índice. Assim, os arrays são definidos como objetos em cujo conteúdo encontram-se diversos valores de um mesmo tipo. Os tipos podem ser primitivos ou tipos construídos (objetos em geral);
- Um array pode ter uma ou mais dimensões, ou seja, pode ser classificado como unidimensional, bidimensional ou multidimensional;
- Para que você possa fazer uma referência a um array criado, é preciso definir o índice em que deseja realizar uma determinada operação. Dessa forma, após declarar e instanciar um array, para acessar ou modificar qualquer um de seus elementos, basta sufixar após o nome da variável de referência do array um par de colchetes com o índice desejado;
- A inicialização de um array refere-se à atribuição de valores aos seus elementos, e sua construção refere-se à instanciação do objeto de array. É possível inicializar um array logo na linha de instanciação ou posteriormente, sendo certo que o tamanho do array deve ser determinado antes de inicializá-lo (quando a inicialização ocorre na mesma linha, o compilador infere o tamanho pela contagem de objetos a serem inseridos);
- Um array pode ser passado e recebido como parâmetro. É importante destacar que o array original reflete quaisquer alterações realizadas nos valores do método que receber um array como parâmetro;
- O parâmetro do array de **String**, no método **main**, é responsável por alocar os argumentos que são enviados juntamente ao comando cuja função é executar o programa.



8

Arrays

Teste seus conhecimentos



1. Qual das alternativas a seguir não é um tipo de array?

- a) Unidimensional
- b) Bidimensional
- c) Adimensional
- d) Multidimensional
- e) Nenhuma das alternativas anteriores está correta.

2. Qual é a palavra-chave utilizada para criar um array unidimensional?

- a) array
- b) create
- c) malloc
- d) type
- e) new

3. Sobre as variáveis de referência de arrays, qual alternativa está correta?

- a) São aquelas que fazem referências aos elementos do array.
- b) São aquelas que fazem referências ao próprio objeto que representa o array.
- c) É impossível atribuir um array a uma variável de referência de array já declarada.
- d) Nenhuma das alternativas anteriores está correta.
- e) Há mais de uma alternativa correta.

4. Sobre arrays, qual alternativa está correta?

- a) São objetos em cujo conteúdo encontram-se diversos valores de um mesmo tipo.
- b) Os tipos de valores dentro de um array podem ser básicos ou tipos construídos.
- c) O índice de um array é um valor inteiro responsável por determinar uma posição dentro do array.
- d) Um array pode ter uma ou mais dimensões.
- e) Todas as alternativas anteriores estão corretas.

5. Qual é o único atributo público dos objetos de array?

- a) size
- b) length
- c) type
- d) index
- e) Nenhuma das alternativas anteriores está correta.



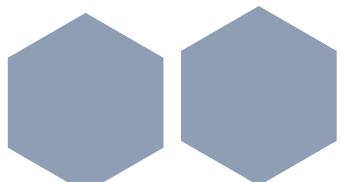
8

Arrays



Mãos à obra!

Resourceful
OASIS
TELL - Quintessential
60710007-67



Laboratório 1

Neste laboratório, vamos criar um método que receba um array do tipo **int** e retorne o maior número deste array.

A – Criando um método que retorne o maior número de um array

1. Crie uma classe com o nome **Cap8_Lab1** e, nela, crie um método estático chamado **maiorNumero** que retorne **int** e receba um array do tipo **int** como parâmetro;
2. A partir do array informado como parâmetro, encontre o maior número do array e retorne este número no método.

B – Executando o método criado

1. Insira a estrutura básica de um programa Java e, dentro dele, declare um array do tipo **int** com o nome **numeros** e o inicialize com quantos números quiser;
2. Faça a chamada ao método criado anteriormente passando o array como parâmetro e armazenando o retorno da chamada do método numa variável do tipo inteiro;
3. Imprima a variável na tela que, se tudo estiver certo, será o maior número do array informado;
4. Compile e execute o programa.

Laboratório 2

Neste laboratório, você criará um programa que aceitará argumentos da linha de comando representando idades de pessoas, calculará a média dessas idades e retornará esse resultado impresso.

A – Criando a estrutura do projeto e a classe principal

1. Crie um projeto denominado **Cap8Lab02** e uma classe nesse projeto, de nome **Cap8_Lab2**, com o método **main** declarado em seu interior;
2. Dentro do método **main**, crie uma estrutura de decisão **if**, que faça a verificação do tamanho do array **args**: Se for maior que zero, entraremos com a lógica a seguir, caso não seja maior que zero, imprima na tela uma mensagem de erro e auxílio do tipo: “**Entre com valores válidos para as idades**”;
3. Dentro do corpo do **if** que validou o tamanho do array como maior do que zero, crie uma variável local chamada **soma**, de tipo **int**, e inicialize-a com zero;
4. Ainda nesse bloco, crie um laço **for** para iterar nos elementos desse array, somando cada um ao anterior, acumulando esse valor na variável **soma**:

```
soma += Integer.parseInt(args[i]);
```

5. Após o fechamento do corpo deste laço, declare outra variável local adequada para guardar a média dos valores e calcule-a, usando a variável **soma** já calculada e o tamanho do array;
6. Exiba na tela o resultado da média conforme a saída a seguir:

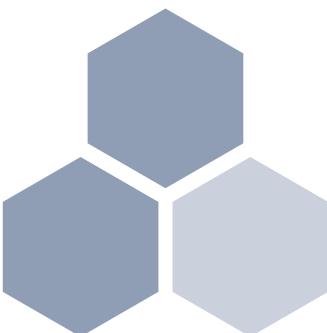
```
Console  
<terminated> Cap8_Lab2 [Java Application]  
A média das idades:  
33 26 54 16  
é de : 32.0 anos.
```

Note que as idades usadas para esse exercício foram inseridas em uma **Run Configuration** do Eclipse. Consulte o item sobre array de argumentos do Capítulo 8 para configurá-los no seu Eclipse.

9

Herança, classes abstratas e polimorfismo

- Herança e generalização;
- Estabelecendo herança entre classes;
- Herança e classes;
- Classes abstratas;
- Polimorfismo;
- UML – Associações entre classes.



9.1. Introdução

O mecanismo de herança é um dos maiores diferenciais entre a programação orientada a objetos e outros tipos de programação, como a programação estruturada. Por meio da herança, uma classe herda os atributos e métodos de outra. A classe que herda as características se chama classe filha (ou subclasse) e a classe que fornece as características se chama classe pai (ou superclasse).

O mecanismo de herança pressupõe uma estrutura hierárquica de classes. Quando temos uma hierarquia de classes planejada de forma adequada, temos a base para que um código possa ser reutilizado, estendido e modificado, o que permite poupar esforço e tempo no desenvolvimento.

A herança possibilita que as classes compartilhem atributos e métodos entre si. Para isso, adota um relacionamento hierárquico entre dois tipos principais de classe:

- **Superclasse:** A classe que está no nível mais elevado da hierarquia e concede as características e comportamentos a outra classe;
- **Subclasse:** A classe que está no nível mais baixo na hierarquia e que herda as características e comportamentos da superclasse.

Se uma classe tem alguns atributos encapsulados, uma subclasse dessa classe herdará esses atributos encapsulados e poderá acessá-los diretamente, desde que o modificador de acesso assim o permita. Além disso, a subclasse pode definir seus próprios atributos que são exclusivos dela (não herdados).

O fato de subclasses herdarem atributos de classes ancestrais assegura que programas orientados a objetos cresçam em complexidade de forma linear e não geométrica. Além disso, nenhuma nova subclasse terá interações imprevisíveis em relação ao restante do código do sistema.

De maneira natural, as pessoas visualizam o mundo formado por objetos, que estão relacionados hierarquicamente entre si. Veja, por exemplo, a relação entre animais: mamíferos e cachorros.

Os animais, sob uma descrição abstrata, apresentam atributos, como tamanho, inteligência e estrutura óssea, e aspectos comportamentais, como mover-se, dormir, respirar, alimentar-se, entre outros. Os atributos e aspectos comportamentais descritos definem a classe dos animais.

Se analisar os mamíferos, que estão inseridos na classe animais, você notará atributos mais particulares, como tipo de dente, pelos e glândulas mamárias.

Os mamíferos são classificados como uma subclasse dos animais, os quais, por sua vez, são uma superclasse de mamíferos.

Partindo do princípio de que uma subclasse recebe por herança todos os atributos de seus ancestrais, e levando-se em consideração a hierarquia de classes, a subclasse **mamíferos** recebe todos os atributos de **animais** e seus comportamentos (métodos).

Para ilustrar, apresentamos a hierarquia (na biologia, o termo usado é taxonomia) de classes do cachorro:

- Reino: Animal;
- Filo: Codata (cão, lobo, raposa, gato, cavalo, cobra, sapo, peixe);
- Classe: Mammalia (cão, lobo, raposa, gato, cavalo);
- Ordem: Carnívora (cão, lobo, raposa, gato);
- Família: Canídea (cão, lobo, raposa);
- Gênero: Canis (cão, lobo);
- Espécie: Canis familiaris (cão): dálmatas, pastor, dobermann etc.

Considere como exemplo um cão da raça dálmatas, pertencente à **espécie** Canis familiaris. Por herança, o dálmatas herda as características do **gênero** Canis, o qual, por sua vez, herda as características da **família** Canídea, que herda as características da **ordem** Carnívora e, assim, sucessivamente.

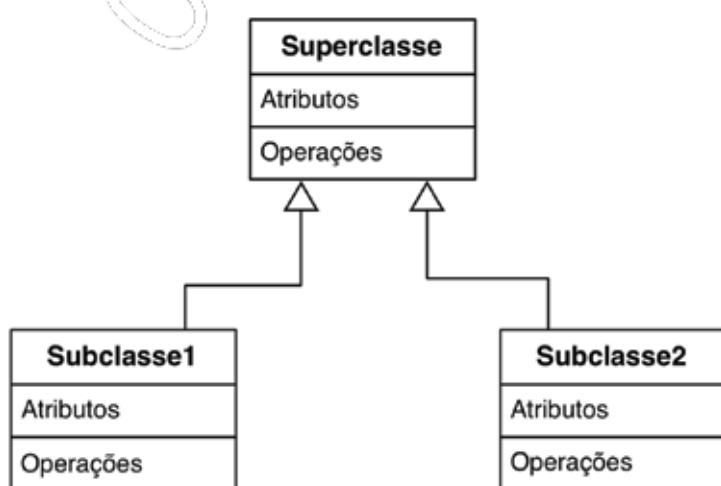
9.2. Herança e generalização

A generalização e a herança são abstrações que permitem que classes compartilhem similaridades ao mesmo tempo em que preservam outras características que as diferem.

A generalização também pode ser chamada de relacionamento **is-a** (ou seja, **é-um**), já que toda instância de classe derivada (subclasse) também é uma instância de classe base (superclasse). As características da superclasse são herdadas pela subclasse.



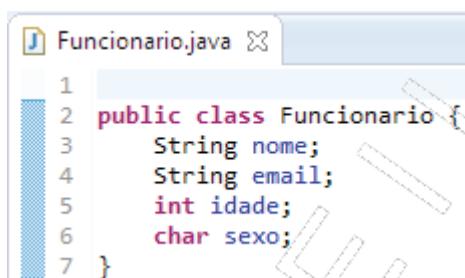
A generalização é a estrutura que permite a utilização do conceito de herança, sendo aplicada a diversos níveis. Veja, na imagem seguinte, a notação UML utilizada para representar a generalização: uma seta contínua com a ponta não preenchida, apontando sempre para a superclasse:



Uma característica de superclasse pode ser sobreposta por uma subclasse, caso esta defina uma característica própria com o mesmo nome. A característica própria da subclasse refinará e substituirá a característica da superclasse. A esse efeito damos o nome de **sobrescrição** (**overriding**).

9.3. Estabelecendo herança entre classes

Para demonstrar a criação de herança entre classes, considere o exemplo apresentado a seguir, que inicia com a criação da classe **Funcionario**:

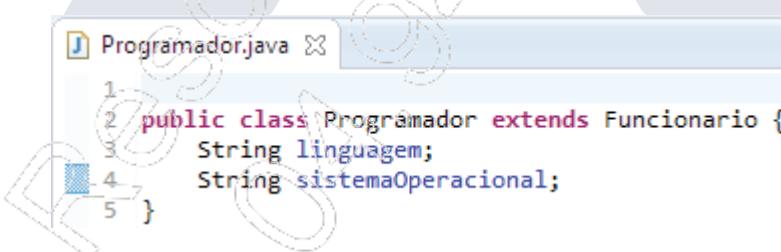


```

1
2 public class Funcionario {
3     String nome;
4     String email;
5     int idade;
6     char sexo;
7 }
```

Considere a criação de um cadastro de programadores de uma empresa, em que cada programador é um **Funcionario**, mas nem todo **Funcionario** é um programador. Em razão dessa relação, é necessário criar uma extensão da classe **Funcionario**, a fim de que ela contenha os dados específicos referentes aos programadores. Desse modo, a extensão criada acaba herdando os membros existentes na definição da classe **Funcionario**.

Veja, a seguir, a definição da classe **Programador**, que, pela herança, recebe as “características” da classe **Funcionario**, somadas à definição de seus próprios membros:



```

1
2 public class Programador extends Funcionario {
3     String linguagem;
4     String sistemaOperacional;
5 }
```

Repare que a classe **Programador** é uma extensão (também chamada de classe filha, derivada e subclasse) da classe **Funcionario** (também chamada de classe pai ou superclasse).

De acordo com o exemplo anterior, o **extends**, que define herança, faz com que a classe **Programador** herde as características e os métodos, caso existam, da classe **Funcionario**.

! Informação importante: Java não trabalha com herança múltipla, ao contrário de algumas linguagens.

Com a herança, você pode criar classes que herdem características de classes dotadas de funcionalidades próprias do Java, como componentes gráficos, multiprocessadores e outras mais.

A seguir, temos um exemplo de **Funcionario** e **Programador** utilizando o recurso de herança:

```
1  public class Funcionario {  
2      // atributos  
3      protected String nome;  
4      protected String email;  
5  
6      // construtores  
7      public Funcionario(){}  
8      public Funcionario(String nome, String email){  
9          this.nome = nome;  
10         this.email = email;  
11     }  
12 }  
  
1  public class Programador extends Funcionario {  
2      // atributos privados  
3      private String linguagem;  
4      private String sistemaOperacional;  
5  
6      // construtores  
7      public Programador(){}
8      public Programador(String n, String e, String linguagem, String sistemaOperacional){  
9          // Os atributos nome e email são HERDADOS da superclasse Funcionário  
10         nome = n;  
11         email = e;  
12         this.linguagem = linguagem;  
13         this.sistemaOperacional = sistemaOperacional;  
14     }  
15  
16     // métodos getters e setters  
17     public String getLinguagem() {  
18         return linguagem;  
19     }  
20     public void setLinguagem(String linguagem) {  
21         this.linguagem = linguagem;  
22     }  
23     public String getSistemaOperacional() {  
24         return sistemaOperacional;  
25     }  
26     public void setSistemaOperacional(String sistemaOperacional) {  
27         this.sistemaOperacional = sistemaOperacional;  
28     }  
29  
30     public void imprimirDados(){  
31         // Os atributos nome e email são HERDADOS da superclasse Funcionário  
32         System.out.println("Nome: " + nome);  
33         System.out.println("Email: " + email);  
34         System.out.println("Linguagem: " + linguagem);  
35         System.out.println("Sistema Operacional: " + sistemaOperacional);  
36     }  
37 }  
38 }
```

```
J TesteProgramador.java X
1
2 public class TesteProgramador {
3     public static void main(String args[]){
4         Programador junior = new Programador("Joãozinho", "joaozinho@xpto.com.br", "JAVA", "Linux");
5         junior.imprimirDados();
6     }
7 }
```

Após a compilação e a execução, o resultado será o seguinte:

```
Console X Problems @ Java
<terminated> TesteProgramador [Java]
Nome: Joãozinho
Email: joaozinho@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Linux
```

9.3.1. Acesso aos membros da superclasse

Os membros privados da superclasse não podem ser referenciados na subclasse, mas é possível contornar essa condição com os métodos **getters** e **setters**. Veja o acesso via métodos assessores, mostrado no exemplo a seguir:

```
J Funcionario.java X
1
2 public class Funcionario {
3     // atributos
4     private String nome;
5     private String email;
6
7     // construtores
8     public Funcionario(){}
9     public Funcionario(String nome, String email){
10         this.nome = nome;
11         this.email = email;
12     }
13
14     // métodos getters e setters
15     public String getNome() {
16         return nome;
17     }
18     public void setNome(String nome) {
19         this.nome = nome;
20     }
21     public String getEmail() {
22         return email;
23     }
24     public void setEmail(String email) {
25         this.email = email;
26     }
27 }
```

Java Programmer - Parte I

```
Programador.java
1  public class Programador extends Funcionario {
2      // atributos privados
3      private String linguagem;
4      private String sistemaOperacional;
5
6      // construtores
7      public Programador(){}
8      public Programador(String nome, String email, String linguagem, String sistemaOperacional){
9          // Os métodos setNome e setEmail são HERDADOS da superclasse Funcionário
10         setName(nome);
11         setEmail(email);
12         this.linguagem = linguagem;
13         this.sistemaOperacional = sistemaOperacional;
14     }
15
16     // métodos getters e setters
17     public String getLinguagem() {
18         return linguagem;
19     }
20     public void setLinguagem(String linguagem) {
21         this.linguagem = linguagem;
22     }
23     public String getSistemaOperacional() {
24         return sistemaOperacional;
25     }
26     public void setSistemaOperacional(String sistemaOperacional) {
27         this.sistemaOperacional = sistemaOperacional;
28     }
29
30
31     public void imprimirDados(){
32         // Os métodos getName e getEmail são HERDADOS da superclasse Funcionário
33         System.out.println("Nome: " + getName());
34         System.out.println("Email: " + getEmail());
35         System.out.println("Linguagem: " + linguagem);
36         System.out.println("Sistema Operacional: " + sistemaOperacional);
37     }
38 }
```



```
TesteProgramador.java
1  public class TesteProgramador {
2      public static void main(String args[]){
3          Programador junior = new Programador("Joãozinho", "joaozinho@xpto.com.br", "JAVA", "Linux");
4          junior.imprimirDados();
5      }
6  }
```

Após a compilação e execução do código, o resultado será o seguinte:

```
Console Problems @ Java
<terminated> TesteProgramador [Java]
Nome: Joãozinho
Email: joaozinho@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Linux
```

9.3.2. O operador super

Qualquer referência com o operador **super** dentro de uma subclasse será feita diretamente para a superclasse.

No exemplo a seguir, a subclasse **Programador** utiliza **super** no construtor e no método **imprimirDados()** para acessar métodos e atributos da superclasse **Fucionario**:

The image shows a Java code editor window with the file 'Funcionario.java' open. The code defines a class 'Funcionario' with attributes 'nome' and 'email', two constructors, and four methods for getting and setting these attributes. Annotations are present throughout the code: 'atributos' above the class definition, 'construtores' above the constructors, 'métodos getters e setters' above the methods, and 'Referencia ao super' with arrows pointing to the 'super' keyword in the constructor and the 'super' object reference 'super' in the 'imprimirDados()' method.

```
1  public class Funcionario {  
2      // atributos  
3      private String nome;  
4      private String email;  
5  
6      // construtores  
7      public Funcionario(){  
8      }  
9      public Funcionario(String nome, String email){  
10         this.nome = nome;  
11         this.email = email;  
12     }  
13  
14     // métodos getters e setters  
15     public String getNome() {  
16         return nome;  
17     }  
18     public void setNome(String nome) {  
19         this.nome = nome;  
20     }  
21     public String getEmail() {  
22         return email;  
23     }  
24     public void setEmail(String email) {  
25         this.email = email;  
26     }  
27 }
```

```
J Programador.java X
1  public class Programador extends Funcionario {
2      // atributos privados
3      private String linguagem;
4      private String sistemaOperacional;
5
6      // construtores
7      public Programador(){}
8      public Programador(String nome, String email, String linguagem, String sistemaOperacional){
9          // acessando os métodos setName e setEmail com o super
10         super.setName(nome);
11         super.setEmail(email);
12         this.linguagem = linguagem;
13         this.sistemaOperacional = sistemaOperacional;
14     }
15
16     // métodos
17     public String getLinguagem() {
18         return linguagem;
19     }
20     public void setLinguagem(String linguagem) {
21         this.linguagem = linguagem;
22     }
23     public String getSistemaOperacional() {
24         return sistemaOperacional;
25     }
26     public void setSistemaOperacional(String sistemaOperacional) {
27         this.sistemaOperacional = sistemaOperacional;
28     }
29
30     public void imprimirDados(){
31         // Os métodos getName e getEmail são HERDADOS da superclasse Funcionário
32         System.out.println("Nome: " + super.getName());
33         System.out.println("Email: " + super.getEmail());
34         System.out.println("Linguagem: " + linguagem);
35         System.out.println("Sistema Operacional: " + sistemaOperacional);
36     }
37 }
38 }
```

```
J TesteProgramador.java X
1
2  public class TesteProgramador {
3      public static void main(String args[]){
4          Programador junior = new Programador("Maria", "maria@xpto.com.br", "JAVA", "Windows");
5          junior.imprimirDados();
6      }
7  }
```

Vale lembrar que existem três possibilidades de acesso com a referência **super**: os construtores da classe pai com **super(<parâmetros>)**, campos ou variáveis de instância visíveis com **super.<variável>** ou ainda métodos com **super.<método>()**.

Lembre-se de que, em Java, é padrão que o nome da classe inicie com letra maiúscula e o objeto tenha o mesmo nome em letra minúscula.

Após a compilação e execução do código, o resultado será o seguinte:

```
Console X Problems @ Java
<terminated> TesteProgramador [Java]
Nome: Maria
Email: maria@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Windows
```

9.3.3. Chamada ao construtor da superclasse

Imagine uma classe cujo construtor exija que os parâmetros sejam passados no momento da instanciação. Para tal classe, suponha a criação de uma classe derivada. Quando criar alguma instância dessa classe derivada, ou filha, você terá um problema.

Pelo fato de o método construtor da superclasse exigir parâmetros, estes devem ser passados para a subclasse, que, por sua vez, os repassa para a superclasse.

O construtor padrão da superclasse será automaticamente chamado no momento da instanciação da subclasse. O fato é que o construtor padrão já não existe mais na superclasse, o que torna obrigatório chamar explicitamente o construtor, de forma a repassar ou empurrar seus parâmetros exigidos por meio da subclasse.

Veja, a seguir, como fazer a chamada ao construtor pai por meio do comando **super** dentro do construtor da subclasse:

```
1  public class Funcionario {
2      // atributos
3      private String nome;
4      private String email;
5
6      // construtores
7      public Funcionario(){}
8      public Funcionario(String nome, String email){
9          super();
10         this.nome = nome;
11         this.email = email;
12     }
13
14     // métodos getters e setters
15     public String getNome() {
16         return nome;
17     }
18     public void setNome(String nome) {
19         this.nome = nome;
20     }
21     public String getEmail() {
22         return email;
23     }
24     public void setEmail(String email) {
25         this.email = email;
26     }
27 }
```

```
Programador.java
1
2 public class Programador extends Funcionario {
3     // atributos privados
4     private String linguagem;
5     private String sistemaOperacional;
6
7     // construtor
8     public Programador(String nome, String email, String linguagem, String sistemaOperacional){
9         // chamada ao construtor pai que recebe 2 argumentos
10        super(nome, email);
11        this.linguagem = linguagem;
12        this.sistemaOperacional = sistemaOperacional;
13    }
14
15    // métodos
16    public String getLinguagem() {
17        return linguagem;
18    }
19    public void setLinguagem(String linguagem) {
20        this.linguagem = linguagem;
21    }
22    public String getSistemaOperacional() {
23        return sistemaOperacional;
24    }
25    public void setSistemaOperacional(String sistemaOperacional) {
26        this.sistemaOperacional = sistemaOperacional;
27    }
28
29    public void imprimirDados(){
30        // Os métodos getName e getEmail são HERDADOS da superclasse Funcionário
31        System.out.println("Nome: " + super.getName());
32        System.out.println("Email: " + super.getEmail());
33        System.out.println("Linguagem: " + linguagem);
34        System.out.println("Sistema Operacional: " + sistemaOperacional);
35    }
36 }
```

O exemplo adiante gerará um erro de compilação, pois o construtor padrão não existe na classe **Programador**:

```
TesteProgramador.java
1
2 public class TesteProgramador {
3     public static void main(String args[]){
4         Programador junior = new Programador();
5         junior.imprimirDados();
6     }
7 }
```

The constructor Programador() is undefined
3 quick fixes available:

Agora, veja a forma correta de criar a instância:

```
TesteProgramador.java
1
2 public class TesteProgramador {
3     public static void main(String args[]){
4         Programador junior = new Programador("Maria", "maria@xpto.com.br", "JAVA", "Windows");
5         junior.imprimirDados();
6     }
7 }
```

Após a compilação e execução do código, o resultado será o seguinte:



```
Console Problems @ Jav
<terminated> TesteProgramador [Java]
Nome: Maria
Email: maria@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Windows
```

9.4. Herança e classes

Veja, adiante, os conceitos de classe **Object**, classes abstratas e classes finais. Compreendê-los é fundamental para trabalhar com herança entre classes nas mais diversas situações.

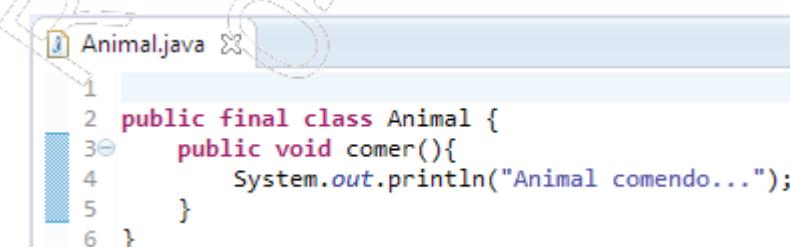
9.4.1. Classes finais

As classes que são declaradas com a palavra-chave **final**, as chamadas classes finais, não podem ser estendidas por outras classes.

 Se você tentar herdar de uma classe final, o compilador apresentará erros.

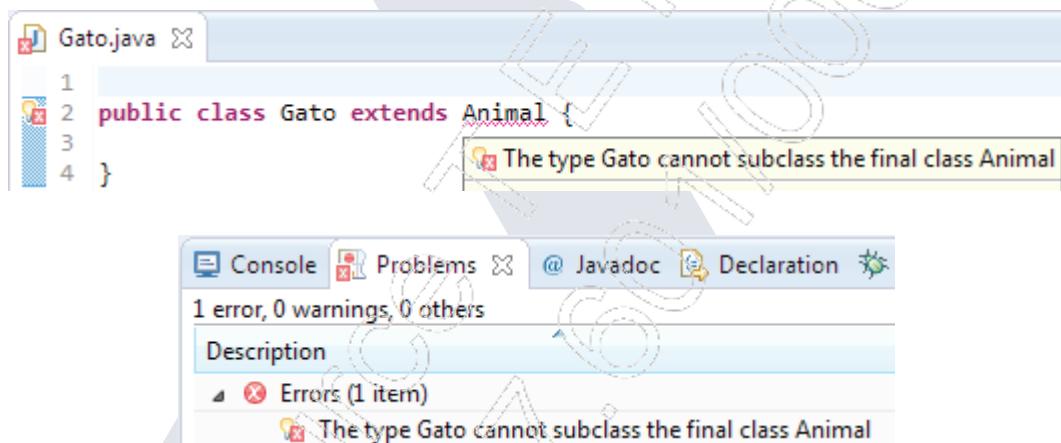
É importante ressaltar que você só deve criar uma classe final quando houver certeza de que ela realmente possui todas as definições que devem ser estabelecidas em seus métodos, pois outro programador não será capaz de estender essa classe e, tampouco, redefinir os métodos.

Os exemplos descritos a seguir mostram como declarar uma classe **final** **Animal** e, em seguida, o que acontece se você tentar compilar a subclasse **Gato**. Veja:



```
Animal.java
1
2 public final class Animal {
3     public void comer(){
4         System.out.println("Animal comendo...");
5     }
6 }
```

A compilação da subclasse **Gato** gera um erro que pode ser visualizado ao posicionar o cursor sobre a palavra sublinhada, ou, então, pela janela **Problems**.



Um grande exemplo de classe **final** em Java é a classe **String**, já vista em exercícios anteriores e largamente utilizada por desenvolvedores Java.

9.4.2. Classe Object

A classe **Object** é a raiz para a definição de todas as classes, uma vez que, de forma implícita, toda classe criada sem uma referência à sua superclasse é derivada diretamente da classe **Object**.

Pelo fato de as classes serem derivadas de forma direta ou indireta da classe **Object**, os objetos de todas as classes têm as definições da classe disponíveis.

Experimente criar uma classe qualquer e verifique os métodos disponíveis para uso após a sua instanciação.

The screenshot shows an IDE interface with a code editor window titled "HerdoObject.java". The code is as follows:

```
1  public class HerdoObject {  
2  
3  
4  public static void main(String[] args) {  
5  
6      HerdoObject herdo = new HerdoObject();  
7      herdo.|  
8  }  
9  
10 }  
11 }  
12 }
```

A code completion dropdown menu is open at the cursor position (line 7, character 5). It lists several methods from the **Object** class:

- clone() : Object - Object
- equals(Object obj) : boolean - Object
- finalize() : void - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object
- main(String[] args) : void - HerdoObject

9.5. Classes abstratas

Classes abstratas são aquelas a partir das quais não é possível criar qualquer tipo de instância. Além disso, classes abstratas podem conter membros que não são abstratos, como métodos e variáveis normais. De forma inversa, caso uma classe contenha ao menos um método abstrato, deverá ser declarada obrigatoriamente como abstrata.

Uma vez que os métodos abstratos não contêm corpo e, tampouco, implementação, a classe que estender a classe abstrata é responsável pela implementação de todos os métodos abstratos.

As classes abstratas somente podem ser estendidas, mas jamais instanciadas e a criação de um objeto a partir delas é um procedimento proibido pelo compilador.

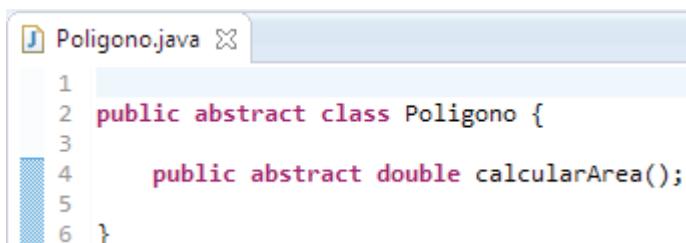
Além disso, caso um ou mais métodos abstratos estejam presentes nessa classe abstrata, a subclasse será, então, forçada a definir tais métodos, ou, de maneira diversa, declarar-se também abstrata. Essa árvore de herança pode crescer ao passo que for preciso, porém, uma regra deverá ser cumprida em todos os casos: a primeira classe concreta a existir na árvore de herança deverá implementar todos os métodos abstratos das classes que estão acima e que não tenham sido implementados ainda.

9.5.1. Métodos abstratos

Os métodos abstratos não possuem implementação e estão presentes somente em classes abstratas. Uma classe abstrata, no entanto, pode ter ou não métodos abstratos. A sintaxe deste tipo de método é a seguinte:

```
<modificadores> abstract <tipoRetorno>
<nomeMetodo>(listaParametros);
```

No exemplo a seguir, a classe **Polígono** será abstrata e terá duas subclasses concretas, **Quadrado** e **Triangulo**, as quais são obrigadas a implementar o método abstrato **calcularArea()**. Observe:



```
1  public abstract class Poligono {
2
3     public abstract double calcularArea();
4
5 }
```



A palavra **abstract** como prefixo da declaração da classe **Polygon** indica que esta classe é abstrata e pode conter métodos abstratos, contudo, quando o modificador **abstract** é utilizado no início da declaração do método **calcularArea()**, significa que este é um método abstrato e, portanto, não possui uma implementação.

O exemplo a seguir mostra como o método **calcularArea()** é implementado nas classes **Quadrado** e **Triangulo**. Veja, então, como definir as classes **Quadrado** e **Triangulo**:

- Definindo a classe **Quadrado**:

```

1
2 public class Quadrado extends Poligono {
3     private double lado;
4
5     public Quadrado(double lado){
6         this.lado = lado;
7     }
8
9     public double calcularArea(){
10        double resultado = lado*lado;
11        System.out.println("Área do quadrado: " + resultado);
12        return resultado;
13    }
14 }
```

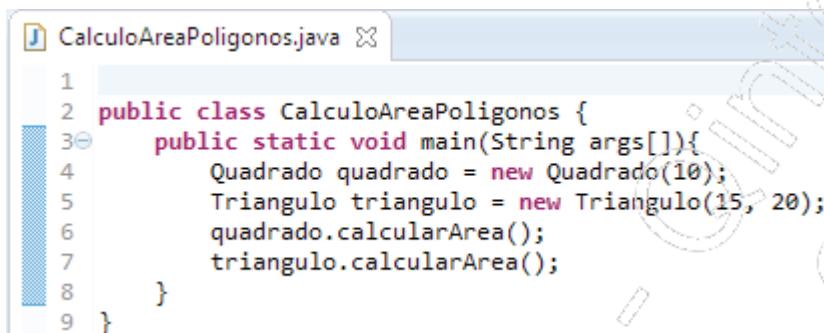
O método **calcularArea()**, que é abstrato na superclasse **Polygon**, é implementado na classe **Quadrado**.

- Definindo a classe **Triangulo**:

```

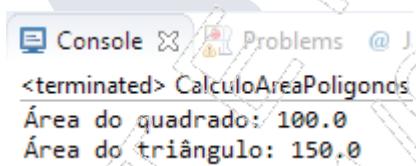
1
2 public class Triangulo extends Poligono {
3     private double base;
4     private double altura;
5
6     public Triangulo(double base, double altura){
7         this.base = base;
8         this.altura = altura;
9     }
10
11     public double calcularArea(){
12        double resultado = (base * altura) / 2;
13        System.out.println("Área do triângulo: " + resultado);
14        return resultado;
15    }
16 }
```

Assim como na classe **Quadrado**, na classe **Triangulo** também é implementado o método **calcularArea()**.



```
1 public class CalculoAreaPoligonos {
2     public static void main(String args[]){
3         Quadrado quadrado = new Quadrado(10);
4         Triangulo triangulo = new Triangulo(15, 20);
5         quadrado.calcularArea();
6         triangulo.calcularArea();
7     }
8 }
```

Depois de compilado e executado o código anterior, o resultado será como mostra a imagem a seguir:



```
Console Problems @ J
<terminated> CalculoAreaPoligonos
Área do quadrado: 100.0
Área do triângulo: 150.0
```

No exemplo anterior, foi descrita a utilização de classes abstratas. Além disso, você também pode utilizar classes abstratas em conjunto com polimorfismo (visto ainda neste capítulo).

Um método abstrato age como uma cláusula de cumprimento obrigatório a todas as classes que herdam da classe abstrata. Torna-se um modelo a ser seguido pelas classes filhas.

9.6. Polimorfismo

Polimorfismo é uma característica das linguagens orientadas a objetos, derivada do conceito de herança, que preconiza a possibilidade de construir objetos que se comportem, conforme o contexto em que forem empregados, de formas diferentes. Esse comportamento é atingido pela utilização de referências a superclasses ou interfaces (veremos em breve esse assunto) no momento da instanciação de classes derivadas comuns, na mesma árvore de herança.

Com essa possibilidade, um método definido na classe pai e herdado por todas as classes filhas poderá ser sobreescrito em cada uma delas com o fim de se obter comportamento específico e diferenciado. Quando a chamada a esse método for feita em uma referência polimórfica (referência à classe pai e instanciação da classe filha), a implementação mais específica é acionada.

O mecanismo de sobreescrita de métodos é usado para que todos os métodos existentes nas classes filhas tenham a mesma assinatura, o que se faz necessário no polimorfismo. Contudo, vale destacar que, ao utilizar o polimorfismo, o comportamento dos métodos somente será estabelecido em tempo de execução.

Por meio do mecanismo de ligação tardia (**late binding**), você seleciona o método que será utilizado, de acordo com o tipo da subclasse. Essa seleção é realizada em tempo de execução.

9.6.1. Ligação tardia (**late binding**)

Para que o polimorfismo possa ser utilizado, a linguagem de programação orientada a objetos deve suportar o mecanismo de ligação tardia. O conceito de ligação tardia indica que o método, que será invocado, é definido somente no decorrer da execução do programa.

Quando você trabalha com a linguagem de programação Java, a definição dos métodos que serão executados sempre ocorre por meio da ligação tardia, exceto nestas duas situações:

- **Métodos declarados como final:** A redefinição deste tipo de método não é possível. Sendo assim, seus descendentes são incapazes de invocá-lo de maneira polimórfica;
- **Métodos declarados como private:** Este tipo de método também não pode ser redefinido, em virtude de não ser visível às classes filhas.

O mecanismo de ligação prematura (**early binding**) é utilizado nas situações em que se define o método que será invocado no decorrer do processo de compilação do programa.

9.6.2. Polimorfismo em métodos declarados na superclasse

Polimorfismo é um termo que caracteriza várias formas. Estudaremos, neste momento, o uso do polimorfismo por meio de métodos declarados na superclasse e redefinidos ou sobrescritos na subclasse. Os métodos sobrescritos devem ter a mesma assinatura do método presente na superclasse.

Embora pareça estranho, em classes que herdam o comportamento de outras, pode ser essencial redefinir um método existente em outra classe e com a mesma assinatura. Caso o método seja redefinido na subclasse, uma chamada a esse método o executará, seja por meio da referência polimórfica de um tipo da classe pai ou por meio de referência específica. Se não houver sobreescrita desse método na subclasse, o método da superclasse será executado.

Um dos exemplos mais claros do uso de polimorfismo é a criação de uma classe derivada de **Frame**, cuja finalidade é a construção de uma janela. Perceba que a superclasse contém todos os métodos para a construção dessa janela, mas as informações adicionais necessárias para que o objeto complemente a janela são de responsabilidade da subclasse, a qual utiliza a implementação de um método específico.

Veja, a seguir, dois exemplos que demonstram o uso do polimorfismo:

- **Exemplo 1**

```
Animal.java
1 public class Animal {
2     public void comer(){
3         System.out.println("Animal comendo...");
4     }
5 }
Zebra.java
1 public class Zebra extends Animal {
2     public void comer(){
3         System.out.println("Zebra comendo...");
4     }
5 }
```

```

Leao.java
1
2 public class Leao extends Animal {
3     public void comer(){
4         System.out.println("Leão comendo...");
5     }
6 }

Zoologico.java
1
2 public class Zoologico {
3     public static void main(String args[]){
4         Animal a = new Animal(); // a está se referenciando à um objeto da classe base Animal
5         a.comer(); // aqui será executado o método comer da classe base Animal
6
7         Zebra z = new Zebra();
8         a = z; // a passa a referenciar um objeto da classe derivada Zebra
9         a.comer(); // aqui será executado o método comer da classe derivada Zebra
10
11        a = new Leao(); // a passar a referenciar o objeto da classe derivada Leao
12        a.comer(); // aqui será executado o método comer da classe derivada Leao
13    }
14 }

```

Após a compilação e execução do exemplo anterior, o resultado será como o indicado na imagem a seguir:

```

Console >
<terminated> Zoologico
Animal comendo...
Zebra comendo...
Leão comendo...

```

- Exemplo 2**

```

Desenho.java
1
2 public class Desenho {
3     public void imprimir(){
4         System.out.println("Figura não especificada");
5     }
6 }

Quadrado.java
1
2 public class Quadrado extends Desenho {
3     public void imprimir(){
4         System.out.println("Imprimindo Quadrado");
5     }
6 }

```

```
J Triangulo.java ×  
1  
2 public class Triangulo extends Desenho {  
3     public void imprimir(){  
4         System.out.println("Imprimindo Triângulo");  
5     }  
6 }  
  
J ImpressaoDesenho.java ×  
1  
2 public class ImpressaoDesenho {  
3     public static void main(String args[]){  
4         Desenho d = new Desenho();  
5         Desenho q = new Quadrado();  
6         Desenho t = new Triangulo();  
7  
8         d.imprimir();  
9         q.imprimir();  
10        t.imprimir();  
11    }  
12 }
```

Após a compilação e execução desse exemplo, o resultado será como o mostrado na imagem a seguir:

```
Console × Problems  
<terminated> ImpressaoDesenho  
Figura não especificada  
Imprimindo Quadrado  
Imprimindo Triângulo
```

9.6.3. Operador instanceof

O operador **instanceof** verifica, de modo dinâmico, se um objeto foi instanciado com base em uma determinada classe. Ele retorna resultados de tipo booleano e possui a seguinte sintaxe:

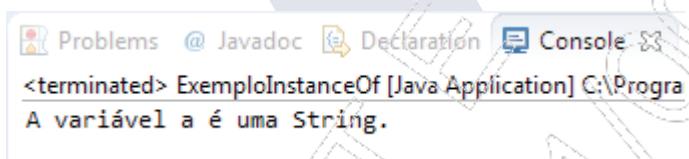
```
objeto instanceof class
```

O **instanceof** só pode ser usado com variáveis de referência a um objeto. Pode ser, contudo, que o objeto testado não corresponda a uma instanciação do tipo de classe. Nesse caso, o resultado apresentado pelo operador **instanceof** será verdadeiro somente se o objeto puder ser atribuído ao tipo.

Veja um exemplo:

```
J ExemploInstanceOf.java X
1
2 public class ExemploInstanceOf {
3     public static void main(String args[]){
4         String a = "teste";
5         if(a instanceof String){
6             System.out.println("A variável a é uma String.");
7         }
8         else{
9             System.out.println("A variável a não é uma String.");
10        }
11    }
12 }
```

A compilação e a execução do código terão a seguinte saída:



Problems @ Javadoc Declaration Console X
<terminated> ExemploInstanceOf [Java Application] C:\Progra
A variável a é uma String.

9.7. UML – Associações entre classes

Em qualquer sistema OO ocorre a associação entre as diferentes classes que compõem o sistema. O relacionamento entre estas classes podem ser de diferentes naturezas. Por exemplo, Uma classe **Motor** tem uma relação forte com a classe **Veiculo**, pois um motor faz parte de um veículo. Já uma classe **Cliente** não tem uma relação tão forte com uma classe **Empresa**, por exemplo.

Uma associação conecta duas instâncias de objeto, de forma física ou conceitual. Ela é uma instância de associação. Por exemplo, **Paulo** é **aluno** da **Impacta**.

Na notação UML, a representação de uma ligação é feita por uma linha conectando dois objetos (ou duas classes). Veja:



Alguns tipos de associações entre classes/objetos serão apresentados sucintamente a seguir.

9.7.1. Tipos de associação

Vejamos, nos subtópicos a seguir, tipos de associação.

9.7.1.1. Associação Simples

Uma associação define um conjunto de ligações entre instâncias de duas ou mais classes, ou um conjunto de ligações que compartilham a mesma semântica e estrutura. As associações descrevem grupos de ligações potenciais. Como exemplo de associação, podemos dizer: **Uma pessoa é aluna de uma faculdade.**

Na notação UML, uma associação é representada por uma linha conectando duas classes, como mostrado a seguir:



- **Atributo de ligação**

Em diversos casos, torna-se útil utilizar atributos que dizem respeito às associações realizadas. Neste caso, temos os números em cada ponta da associação, que denominamos cardinalidade. Esses números demonstram a grandeza lógica em que podem ocorrer as associações. Um identificador também pode ser utilizado para explicar a associação. Veja no esquema a seguir:

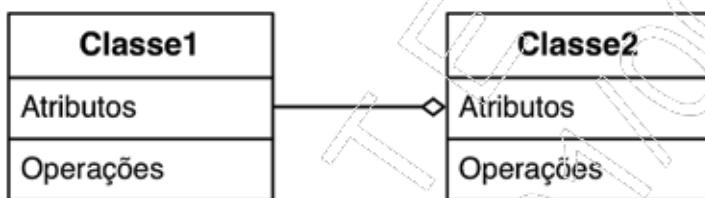


9.7.1.2. Agregação

A agregação é um tipo de associação em que um todo é composto por partes. Há coesão entre as partes, contudo, elas não são totalmente dependentes. Como exemplo, imagine a criação de uma classe **Família**, composta por vários membros da classe **Pessoa**. A classe **Família** não pode existir sem os membros da classe **Pessoa**. Eles, porém, podem existir fora da classe **Família**.

A agregação define um relacionamento do tipo **uma-parte-de**, em que objetos-parte representam componentes de um objeto-todo, mas não são contidos por ele. Isto significa que um componente que faz parte de outro pode existir isoladamente.

A seguir, você pode ver uma representação gráfica de acordo com a UML, que representa uma associação do tipo agregação:



9.7.1.3. Composição

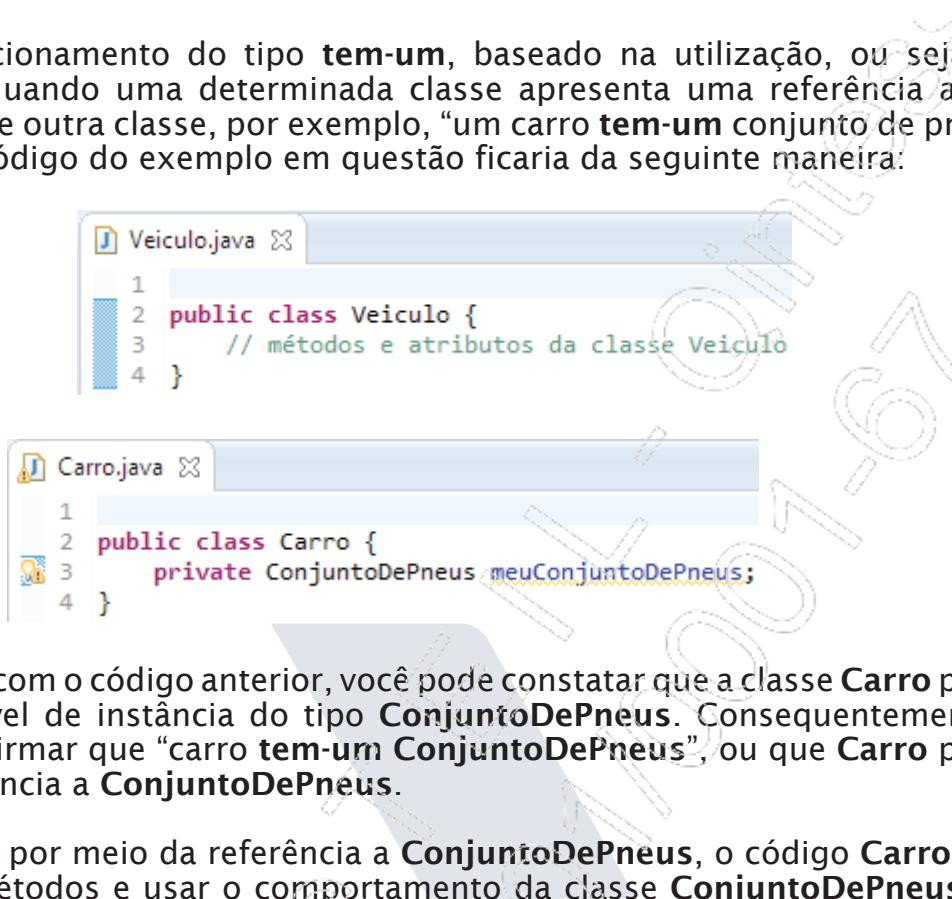
A composição é outro tipo de associação, em que há total dependência entre o todo e as partes que o compõem, de tal modo que as partes não existem isoladamente, sem o todo. É um relacionamento em que um objeto (contêiner) contém outros objetos, o que significa que os objetos contidos dependem do objeto contêiner para existir.

Imagine, por exemplo, uma nota fiscal com um objeto representando um item da nota fiscal (o produto comprado). Este objeto pertence a esta nota específica, e somente a ela. Não faz sentido ele existir se a nota fiscal não existe. Assim, se a nota é destruída, ele também será.

Na UML, a representação da composição é feita por uma linha que possui um losango preenchido no lado da classe dona do relacionamento. Veja:



É um relacionamento do tipo **tem-um**, baseado na utilização, ou seja, ele acontece quando uma determinada classe apresenta uma referência a uma instância de outra classe, por exemplo, “um carro **tem-um** conjunto de pneus”. Assim, o código do exemplo em questão ficaria da seguinte maneira:



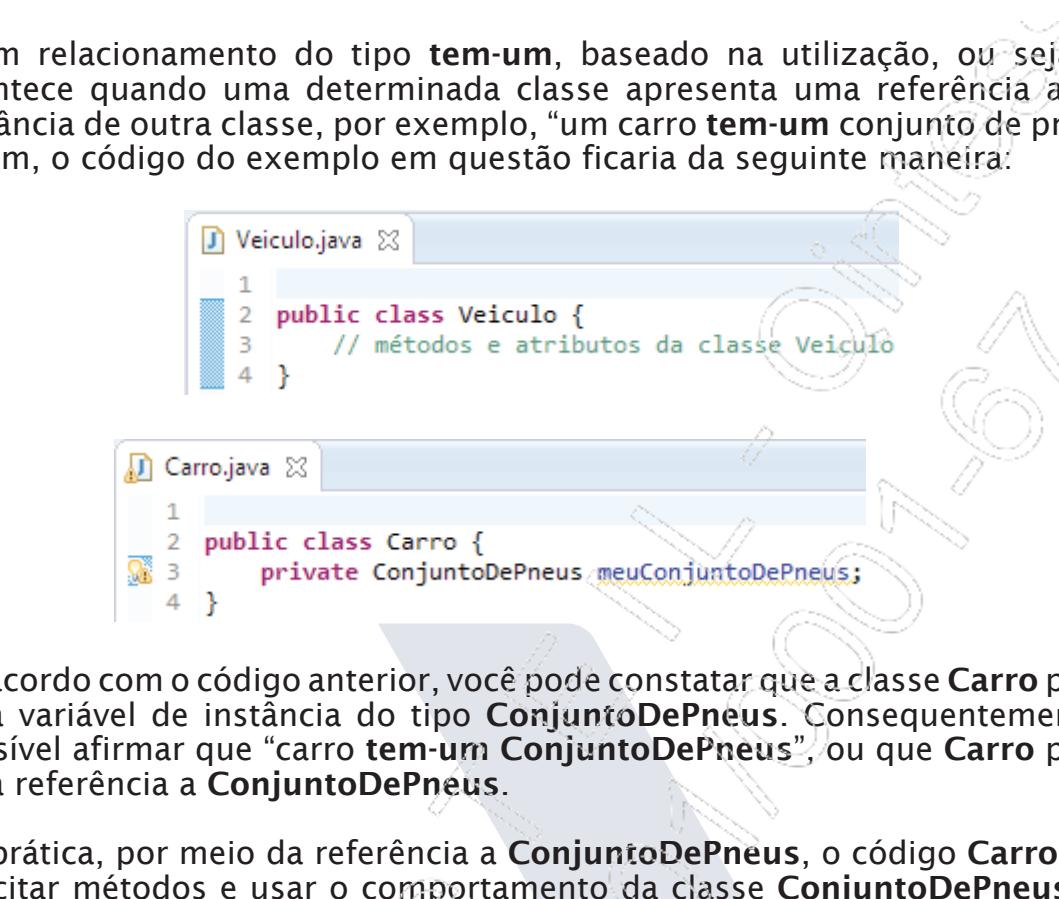
```
Veiculo.java
1
2 public class Veiculo {
3     // métodos e atributos da classe Veiculo
4 }

Carro.java
1
2 public class Carro {
3     private ConjuntoDePneus meuConjuntoDePneus;
4 }
```

De acordo com o código anterior, você pode constatar que a classe **Carro** possui uma variável de instância do tipo **ConjuntoDePneus**. Consequentemente, é possível afirmar que “carro **tem-um** **ConjuntoDePneus**”, ou que **Carro** possui uma referência a **ConjuntoDePneus**.

Na prática, por meio da referência a **ConjuntoDePneus**, o código **Carro** pode solicitar métodos e usar o comportamento da classe **ConjuntoDePneus** sem a necessidade da existência de um código que esteja relacionado à ela dentro da própria classe **Carro**.

Você deve perceber que uma vez que **Carro** contém uma referência a **ConjuntoDePneus**, os códigos que chamarem métodos de **Carro** entenderão como se esta classe tivesse o comportamento de **ConjuntoDePneus**. Na verdade, o que acontece é que o objeto **Carro** delegará a tarefa de chamada do método **rodar()** para a classe **ConjuntoDePneus**, por meio de **meuConjuntoDePneus.rodar()**. Veja a seguir a descrição de como ficaria essa delegação:



```
Carro.java
1
2 public class Carro {
3     private ConjuntoDePneus meuConjuntoDePneus;
4
5     public void rodar(){
6         // delega o comportamento de rodar para o objeto ConjuntoDePneus
7         meuConjuntoDePneus.rodar();
8     }
9 }

ConjuntoDePneus.java
1
2 public class ConjuntoDePneus {
3     public void rodar(){
4         // Realiza a tarefa atual de rodar aqui
5     }
6 }
```

Pela programação orientada a objetos, a classe **Carro** ocultará os detalhes da implementação, já que não é intenção informar aos códigos chamadores qual classe ou objeto realizará a tarefa solicitada por eles. Para esses códigos, a impressão é de que qualquer tarefa é feita pelo objeto **Carro**. Essa característica é também um dos motivadores do uso da composição em detrimento da herança, opções de arquitetura que dividem a preferência dos especialistas na área.

9.7.1.4. Herança

Na programação orientada a objetos, utiliza-se o **é-um**, um tipo de relacionamento baseado na herança. Este relacionamento é uma forma de classificar um item, por exemplo, “banana é-uma fruta” ou “cachorro é-um animal”. Utilize a palavra-chave **extends** para expressar o relacionamento **é-um**:

```
Cachorro.java
1 public class Cachorro {
2     // métodos e atributos da classe Cachorro
3 }
4 
```



```
Dalmata.java
1 public class Dalmata {
2     // coisas importantes e específicas referentes a Dalmata vão aqui
3     // não esquecer que Dalmata herda os membros acessíveis de Cachorro
4 }
5 
```

Veja a seguir a ordem da árvore das classes:

```
public class Animal { ... }

public class Cachorro extends Animal { ... }

public class Dalmata extends Cachorro { ... }
```

Assim, de acordo com a terminologia utilizada pela programação orientada a objetos, é correto dizer que:

- **Animal** é a superclasse de **Cachorro**;
- **Cachorro** é a subclasse de **Animal**;
- **Cachorro** é a superclasse de **Dalmata**;
- **Dalmata** é a subclasse de **Animal**;
- **Cachorro** herda de **Animal**;
- **Dalmata** herda de **Cachorro**;
- **Dalmata** herda de **Animal**;
- **Dalmata** é derivado de **Cachorro**;
- **Cachorro** é derivado de **Animal**.

9.7.2. Herança x Composição

Uma das características da programação orientada a objetos é a possibilidade de reutilizar códigos. Para isso são usados dois mecanismos: a herança e a composição. É importante que você conheça e compare as características de cada um para usá-los de modo adequado.

- **Herança**

A partir das características da herança vistas até aqui (uma subclasse pode ser especificada a partir de superclasses, herdando características desta e tendo outras próprias), destacamos alguns aspectos de sua utilização, para compará-los com o mecanismo da composição:

- Agrega o que é comum e isola as características particulares;
- Pode ser vista diretamente no código;
- Gera forte acoplamento, ou seja, qualquer mudança em uma superclasse implica em alteração em todas as subclasses;
- É um relacionamento estático, ou seja, as implementações herdadas por uma subclasse não podem ser alteradas em tempo de execução. Isso pode impedir algumas alterações necessárias;
- Permite maior reutilização de código do que a composição;
- É de mais fácil entendimento para os programadores.

- **Composição**

A composição permite que uma instância da classe existente seja usada como componente da outra classe. Veja a seguir algumas características do uso da composição:

- Faz com que um objeto instanciado e contido na classe que o instanciou seja acessado somente por sua interface;
- Depende menos de implementações;
- É dinâmica, podendo ser definida em tempo de execução e permitindo aos objetos mudanças de comportamento ao longo do seu tempo de vida;
- Cada classe é focada para uma tarefa, apenas;
- Possui código dinâmico e parametrizado. Por isso, é mais difícil de compreender.

O exemplo a seguir esclarece melhor o conceito de composição:

```
Pessoa.java
1 public class Pessoa {
2     private String nome;
3 }
4

Cachorro.java
1 public class Cachorro {
2     private String nome;
3     private Pessoa dono = new Pessoa(); // Composição
4 }
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A herança possibilita que as classes compartilhem seus atributos e métodos entre si. Para isso, a herança adota um relacionamento hierárquico entre dois tipos principais de classe: superclasse e subclasse;
- A generalização e a herança são abstrações que permitem que classes compartilhem similaridades ao mesmo tempo em que preservam outras características que as diferem;
- Com a herança, você pode criar classes que herdem características e comportamento de classes dotadas de funcionalidades próprias do Java, como criação de componentes gráficos, multiprocessadores e outras mais;
- É fundamental compreender os conceitos de classe Object, classes abstratas e classes finais para trabalhar com herança entre classes nas mais diversas situações;
- Polimorfismo é uma característica das linguagens orientadas a objetos, derivada do conceito de herança, que preconiza a possibilidade de construir objetos que se comportem, conforme o contexto em que forem empregados, de formas diferentes. Esse comportamento é atingido pela utilização de referências a superclasses ou interfaces, quando no momento da instanciação de classes derivadas comuns, na mesma árvore de herança;
- Uma associação conecta duas instâncias de objeto, de forma física ou conceitual. Ela é uma instância de associação;
- Uma das características da programação orientada a objetos é a possibilidade de reutilizar código. Para isso são usados dois mecanismos principais: a herança e a composição.



9

Herança, classes abstratas e polimorfismo

Teste seus conhecimentos



1. Com relação a herança, qual a alternativa correta?

- a) A herança possibilita que as classes compartilhem seus atributos e métodos entre si.
- b) A herança possibilita que as classes se comuniquem entre si, mas não compartilhem seus atributos e métodos.
- c) A herança possibilita que uma classe acesse apenas atributos de outra classe.
- d) A herança possibilita que as classes compartilhem apenas seus métodos entre si.
- e) Nenhuma das alternativas anteriores está correta.

2. Quais são os dois principais mecanismos que possibilitam a reutilização de código em Java?

- a) Herança e agregação.
- b) Herança e composição.
- c) Composição e agregação.
- d) Herança e generalização.
- e) Nenhuma das alternativas anteriores está correta.

3. Qual a função do operador super?

- a) Acessar métodos, atributos e construtores da subclasse.
- b) Acessar métodos, atributos e construtores da superclasse.
- c) Acessar apenas métodos e atributos da subclasse.
- d) Acessar apenas métodos e atributos da superclasse.
- e) Nenhuma das alternativas anteriores está correta.

4. O que é uma classe abstrata?

- a) É uma classe que não possui métodos.
- b) É uma classe que não possui construtor.
- c) É uma classe que não pode ser estendida.
- d) É uma classe que não pode ser instanciada.
- e) Nenhuma das alternativas anteriores está correta.

5. O que é uma classe final?

- a) É uma classe que não possui métodos.
- b) É uma classe que não possui construtor.
- c) É uma classe que não pode ser estendida.
- d) É uma classe que não pode ser instanciada.
- e) Nenhuma das alternativas anteriores está correta.

6. Qual a função do operador instanceof?

- a) Verificar se uma classe possui instâncias.
- b) Verificar se um objeto é instância de determinada classe.
- c) Verificar se um método pertence a determinado objeto.
- d) Verificar se um objeto pertence a determinado método.
- e) Nenhuma das alternativas anteriores está correta.



9

Herança, classes abstratas e polimorfismo

Mãos à obra!



Laboratório 1

Neste laboratório, vamos criar as classes **Professor** e **Aluno**, que herdarão atributos e métodos da classe abstrata **Pessoa**.

A – Criando a classe RG

1. Crie uma classe chamada **RG**, com os atributos privados **numero** do tipo **int** e **dataNasc** do tipo **String**;
2. Crie os métodos **get** e **set** para cada atributo;
3. Crie um construtor que receba um parâmetro do tipo **int** e um do tipo **String**, e os atribua a seus respectivos atributos;
4. Salve e compile a classe.

B – Criando a classe Pessoa

1. Crie uma classe abstrata chamada **Pessoa** com os seguintes atributos:
 - **nome** do tipo **String**;
 - **idade** do tipo **int**;
 - **sexo** do tipo **char**;
 - **rg** do tipo **RG**.
2. Crie os métodos **get** e **set** para cada atributo;
3. Crie um construtor que receba os parâmetros a seguir e os atribua a seus respectivos atributos:
 - **nome** do tipo **String**;
 - **idade** do tipo **int**;
 - **sexo** do tipo **char**;
 - **rg** do tipo **RG**.
4. Crie dois métodos abstratos com os nomes **falar**, que receba o parâmetro **fala** do tipo **String**, e **mostrarDados**;
5. Salve e compile a classe.

C – Criando a classe Professor

1. Crie uma classe com o nome **Professor**, que deve ser subclasse de **Pessoa**;
2. Declare os atributos privados **salario** do tipo **float** e **disciplina** do tipo **String**;
3. Crie os métodos **get** e **set** para cada atributo;
4. Crie um construtor que receba os seguintes parâmetros:
 - **nome** do tipo **String**;
 - **idade** do tipo **int**;
 - **sexo** do tipo **char**;
 - **numeroRG** do tipo **int**;
 - **dataNasc** do tipo **String**;
 - **salario** do tipo **float**;
 - **disciplina** do tipo **String**.

5. Dentro do construtor, chame o operador **super** passando os parâmetros **nome**, **idade**, **sexo** e um objeto do tipo **RG** com os parâmetros **numeroRG** e **dataNasc**;

Atenção: Você deve criar a instância de **RG** no mesmo ponto onde o parâmetro **RG** é esperado na chamada ao construtor da superclasse.

6. Atribua os parâmetros **salario** e **disciplina** a seus respectivos atributos;
7. Implemente o método abstrato **falar** e, dentro dele, imprima uma frase contendo o atributo **nome** da pessoa e o valor do parâmetro informado;
8. Implemente o método abstrato **mostrarDados** e, dentro dele, imprima todos os atributos da classe e da superclasse;
9. Salve a classe e compile.

D – Criando a classe Aluno

1. Crie uma classe com o nome **Aluno**, que deve ser subclasse de **Pessoa**;
2. Declare os atributos privados **mensalidade**, do tipo **float**, e **curso**, do tipo **String**;
3. Crie os métodos **get** e **set** para cada atributo;
4. Crie um construtor que receba os seguintes parâmetros:
 - **nome** do tipo **String**;
 - **idade** do tipo **int**;
 - **sexo** do tipo **char**;
 - **numeroRG** do tipo **int**;
 - **dataNasc** do tipo **String**;
 - **mensalidade** do tipo **float**;
 - **curso** do tipo **String**.

5. Dentro do construtor, chame o operador **super** passando os parâmetros **nome**, **idade**, **sexo** e um objeto do tipo **RG** com os parâmetros **numeroRG** e **dataNasc**;

Atenção: Você deve criar a instância de RG no mesmo ponto onde o parâmetro RG é esperado na chamada ao construtor da superclasse.

6. Atribua os parâmetros **mensalidade** e **curso** a seus respectivos atributos;
7. Implemente o método abstrato **falar** e, dentro dele, imprima uma frase contendo o atributo **nome** da pessoa e o valor do parâmetro informado;
8. Implemente o método abstrato **mostrarDados** e, dentro dele, imprima todos os atributos da classe e da superclasse;
9. Salve a classe e compile.

E – Testando todas as classes criadas

1. Crie uma classe chamada **Cap9_Lab1** e insira a estrutura básica de um programa Java;
2. Crie três objetos do tipo **Pessoa**. Para dois objetos, utilize o construtor da classe **Aluno** e, para o outro, utilize o construtor da classe **Professor**, passando os devidos parâmetros;
3. Utilize o método **falar** com cada objeto e, depois, mostre os dados de cada objeto com o método **mostrarDados**;
4. Compile e execute o programa:

```
Console Problems @ javadoc
<terminated> Cap9_Lab1 [Java Application]
Rafael: Manuel?
Manuel: Presente
Rafael: Claudia?
Claudia: Presente

--- Professor: Rafael ---
Idade: 38
Sexo: M
Salário: 2500.0
Disciplina: Português
Número RG: 261454789
Data de Nascimento: 05/02/1974

--- Aluno: Manuel ---
Idade: 19
Sexo: M
Mensalidade: 1099.0
Curso: Ciência da Computação
Número RG: 521234567
Data de Nascimento: 15/06/1993

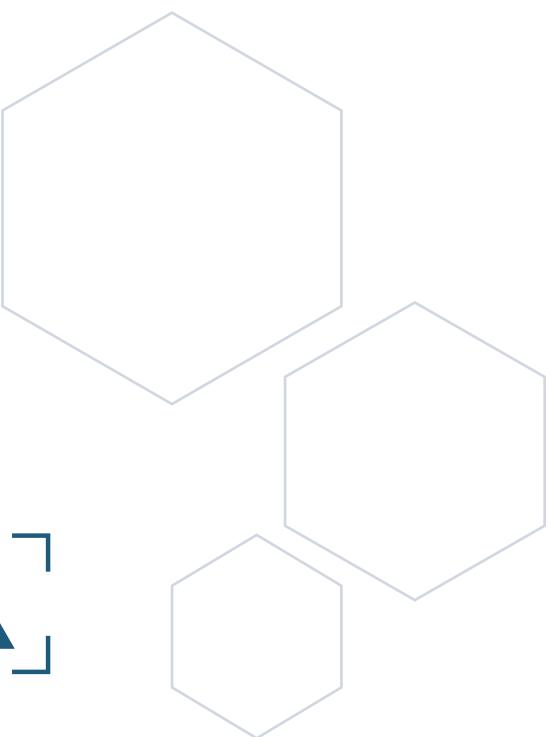
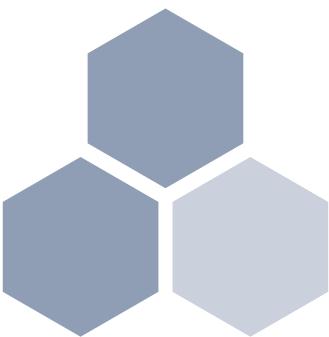
--- Aluno: Claudia ---
Idade: 22
Sexo: F
Mensalidade: 799.0
Curso: Administração
Número RG: 415678912
Data de Nascimento: 12/08/1990
```




10

Interfaces

- ◆ O conceito de interface;
- ◆ Métodos em interfaces.



10.1. O conceito de interface

Como você já sabe, a linguagem Java não usa herança múltipla, o que simplifica o uso de objetos e evita alguns erros de programação. Para prover mecanismo semelhante à herança múltipla com a segurança devida, ela utiliza interfaces.

A herança múltipla possibilita que uma determinada subclasse herde atributos e/ou métodos de duas ou mais superclasses.

As interfaces são estruturas compostas por um conjunto de métodos abstratos, estáticos, default ou privados e constantes estáticas (tipos **final static**). Por meio das interfaces, você pode especificar as funcionalidades a serem desenvolvidas pelas classes que implementem a interface, propiciando grande potencial ao programador na abstração de objetos.

Semelhante à herança/generalização, o uso de interfaces também segue o princípio do relacionamento **is-a** (é-um), porém, em UML, tal relacionamento é chamado de implementação ou realização.

Você vai perceber que uma interface é bem parecida com uma classe. A seguir, listamos as principais diferenças entre elas:

- Uma classe pode conter características e implementações de várias interfaces;
- Uma interface não pode ser instanciada por ser semelhante a uma classe completamente abstrata. Interfaces são criadas para servirem de modelo e nunca de implementação física para objetos;
- Em uma interface não podem existir atributos, mas sim constantes. Qualquer variável declarada em uma interface deve, obrigatoriamente, possuir um valor de inicialização, recebendo implicitamente os modificadores **public**, **static** e **final**;
- Os métodos de uma interface devem ser implementados pela classe que implementa a interface. Além disso, todos os seus métodos são implicitamente **public** e **abstract**, ainda que não explicitamente declarados dessa forma;
- As interfaces podem ser usadas livremente, pois se encontram em uma hierarquia independente.

Como você viu, as interfaces possibilitam o estabelecimento de um conjunto de métodos, constantes ou variáveis que podem ser implementados em qualquer classe, o que significa que tornam possível o uso de classes abstratas puras.

O exemplo a seguir mostra o uso de interfaces:

```

Eletrodomestico.java
1 |
2 public interface Eletrodomestico {
3     void ligar();
4     void desligar();
5 }
6

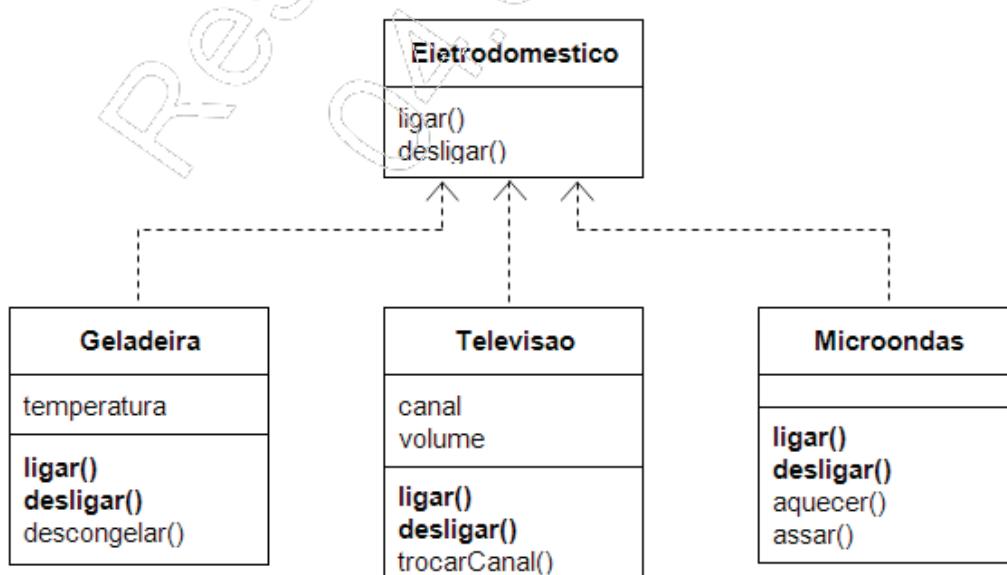
Geladeira.java
1 |
2 public class Geladeira implements Eletrodomestico {
3
4     public void ligar() {
5         // implementação do método ligar declarado na interface Eletrodomestico
6     }
7
8     public void desligar() {
9         // implementação do método desligar declarado na interface Eletrodomestico
10    }
11 }
12

```

10.1.1. Variáveis de referência

Uma interface é usada como tipo para criação de variáveis de referência a objetos, seguindo o padrão de relacionamento **is-a** (é-um). Quando você cria uma variável dessa maneira, pode utilizá-la para referenciar uma instância de qualquer classe que implemente a interface.

Veja um esquema exemplificativo:



Para chamar polimorficamente um método, ou seja, o método do tipo específico da instância criada, você pode utilizar uma referência abstrata, do tipo de uma interface para chamar o método. Assim, a instância da interface que está sendo referenciada será tomada como base. Observe:



```
1 | 
2 | public class Executando {
3 | 
4 |     public static void main(String[] args) {
5 | 
6 |         Eletrodomestico aparelho;
7 | 
8 |         aparelho = new Geladeira();
9 |         aparelho.ligar();
10 | 
11 |         aparelho = new Televisao();
12 |         aparelho.ligar();
13 | 
14 |         aparelho = new Microondas();
15 |         aparelho.ligar();
16 |     }
17 |
18 }
```

Perceba que o método a ser executado é resolvido no tempo de execução, de forma dinâmica, da mesma forma que ocorre com a herança. Essa é a característica mais marcante do polimorfismo.

Apenas os métodos que são declarados na interface são de conhecimento da variável de referência à interface.

10.1.2. Constantes

Uma interface pode conter atributos inicializados, os quais funcionam como constantes, porém, ela não contém variáveis de instância. Em razão disso, as variáveis que funcionam como constantes são compartilhadas pelas classes que implementam esta interface.

Na linguagem Java, você pode definir uma interface apenas com as constantes, sem utilizar os métodos. Neste caso, as constantes são importadas pela classe, ou seja, ela não implementa coisa alguma. Essas constantes importadas pela classe fazem o papel de variáveis finais estáticas.

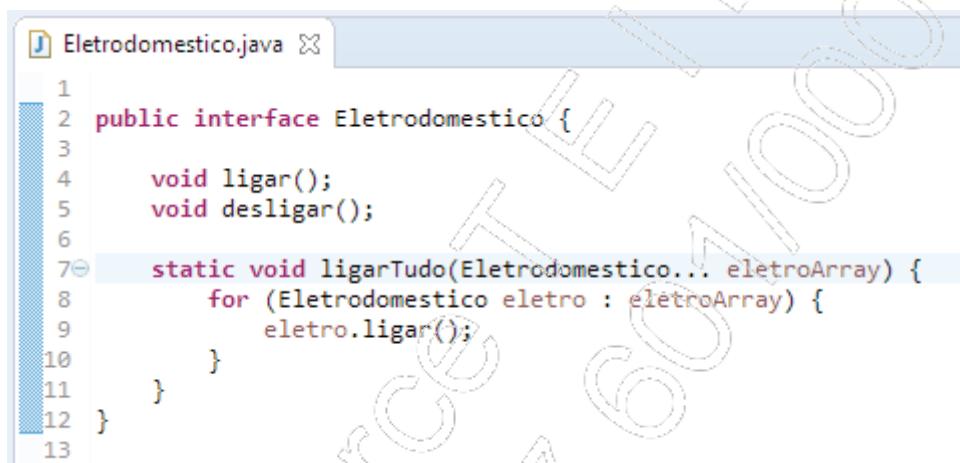
10.2. Métodos em interfaces

Vejamos, nos subtópicos adiante, métodos em interfaces.

10.2.1. Métodos estáticos

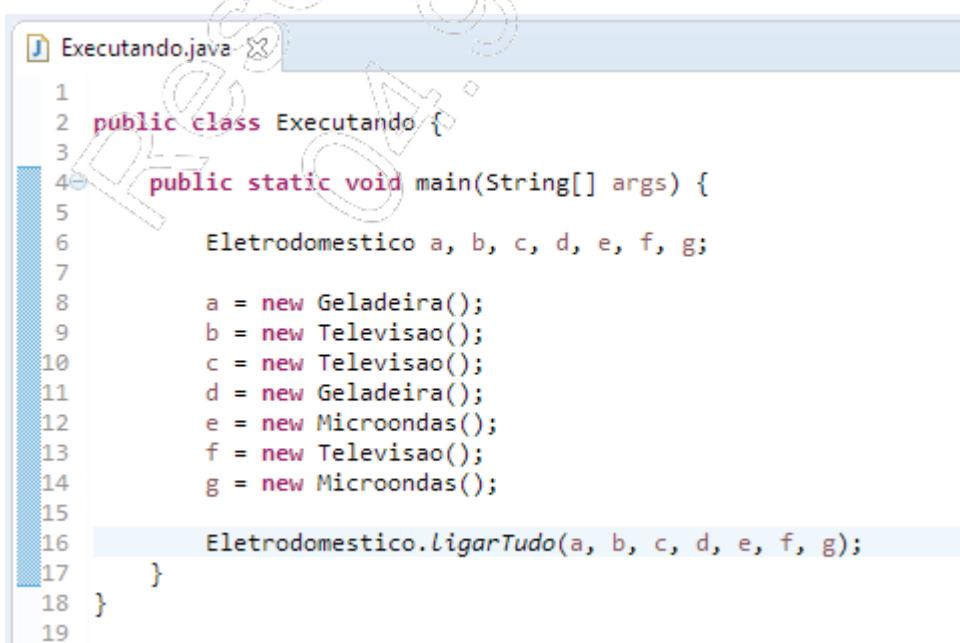
Umas das inovações da versão 8 do Java é a capacidade de implementar métodos estáticos dentro de uma interface. O objetivo deste novo recurso é permitir que sejam criados, na própria interface, métodos de tratamento para entidades que implementam a própria interface.

Observe:



```
1  public interface Eletrodomestico {  
2      void ligar();  
3      void desligar();  
4  }  
5  static void ligarTudo(Eletrodomestico... eletroArray) {  
6      for (Eletrodomestico eletr : eletroArray) {  
7          eletr.ligar();  
8      }  
9  }
```

Por serem estáticos, estes métodos são chamados a partir do nome da interface, sem a necessidade de nenhuma variável de referência:



```
1  public class Executando {  
2      public static void main(String[] args) {  
3          Eletrodomestico a, b, c, d, e, f, g;  
4  
5          a = new Geladeira();  
6          b = new Televisao();  
7          c = new Televisao();  
8          d = new Geladeira();  
9          e = new Microondas();  
10         f = new Televisao();  
11         g = new Microondas();  
12  
13         Eletrodomestico.ligarTudo(a, b, c, d, e, f, g);  
14     }  
15 }
```

10.2.2. Métodos default

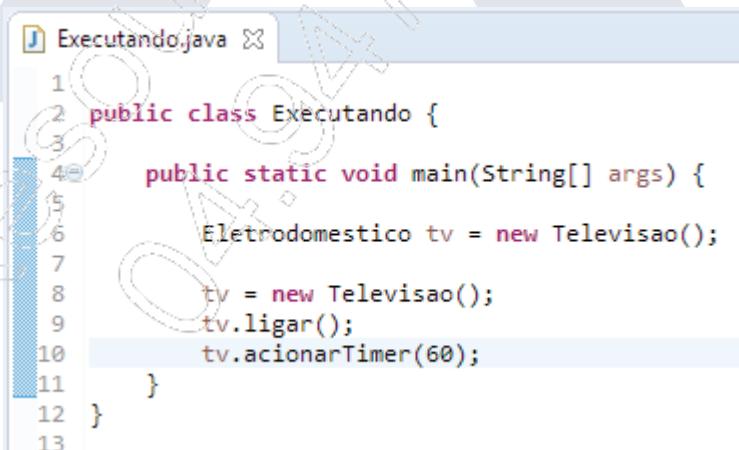
Outra inovação interessante no Java 8 é o método **default** (padrão). Trata-se de um método comum, criado dentro da interface, que permite criar um comportamento padrão ou implementação padrão para todas as classes que implementam aquela interface.

Observe:



```
Eletrodomestico.java
1  public interface Eletrodomestico {
2
3      void ligar();
4
5      void desligar();
6
7      default void acionarTimer(int minutos) {
8          /*
9             * Aguarda a quantidade de minutos informada.
10            */
11         try { Thread.sleep(minutos * 60000); } catch (Exception e) {}
12
13         /*
14            * Executa o método desligar conforme implementado na classe.
15            */
16         desligar();
17     }
}
```

Este método pode ser normalmente utilizado por todos os objetos daquele tipo, sem que a classe implementadora possua tal método:



```
Executando.java
1  public class Executando {
2
3      public static void main(String[] args) {
4
5          Eletrodomestico tv = new Televisao();
6
7          tv = new Televisao();
8          tv.ligar();
9          tv.acaoarTimer(60);
10     }
11 }
12
13 }
```

10.2.3. Métodos privados

A partir do Java 9, abriu-se a possibilidade da criação de métodos privados nas interfaces. Estes métodos podem ser utilizados como apoio a métodos default, que podem compartilhar determinada funcionalidade definida na própria interface.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

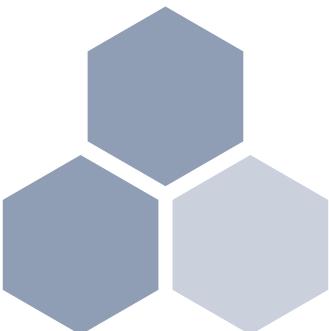
- As interfaces são estruturas compostas por um conjunto de métodos abstratos, estáticos ou default e também por constantes estáticas (tipos **final static**). Por meio das interfaces, você pode especificar quais são as funcionalidades a serem definidas pelas classes, propiciando grande potencial ao programador na abstração de objetos;
- Uma interface é usada como tipo para criação de variáveis de referência a objetos. Quando você cria uma variável dessa maneira, pode usá-la para referenciar uma instância de qualquer classe que implemente a interface;
- Uma interface pode conter constantes inicializadas, porém ela não contém variáveis de instância. Em razão disso, as constantes são compartilhadas pelas classes que implementam esta interface;
- Os métodos estáticos permitem que sejam criados, na própria interface, métodos de tratamento para entidades que implementam a interface. Por serem estáticos, eles são chamados a partir do nome da interface, sem a necessidade de nenhuma variável de referência;
- O método **default** (padrão) é um método comum, criado dentro da interface, que permite criar um comportamento padrão ou implementação padrão para todas as classes que implementam aquela interface. Este método pode ser normalmente utilizado por todos os objetos daquele tipo, sem que a classe implementadora possua tal método;
- Além de **default**, métodos da interface podem receber os modificadores **static** e **private**.



10

Interfaces

Teste seus conhecimentos



1. Com relação a interfaces, qual das alternativas a seguir está incorreta?

- a) Uma classe pode conter características de várias interfaces.
- b) Uma interface não possui atributos, mas pode conter constantes estáticas.
- c) Uma interface pode ser instanciada por um programa.
- d) As interfaces podem ser usadas livremente, pois se encontram em uma hierarquia independente.
- e) Nenhuma das alternativas anteriores está correta.

2. Qual assinatura de método não pode existir em uma interface?

- a) public static void metodo().
- b) protected void metodo().
- c) private void metodo().
- d) default void metodo().
- e) static void metodo().

3. Qual das alternativas a seguir não qualifica as variáveis em uma interface e seus atributos?

- a) public
- b) static
- c) final
- d) private
- e) Todas as alternativas anteriores estão corretas.



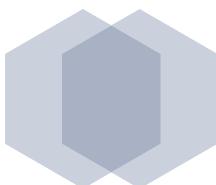
10

Interfaces



Mãos à obra!

Resourcetell - Quintessential
10007-67



Laboratório 1

Neste laboratório, vamos criar uma interface e duas classes que a implementam, de forma a testar as propriedades polimórficas presentes.

A – Criando a interface Imprimivel

1. Crie uma nova interface de nome **Imprimivel**;
2. Crie, nessa interface, um único método chamado **imprimir()**, com retorno do tipo **void** e sem parâmetros;
3. Salve e compile.

B – Criando a classe Relatorio

1. Crie uma classe chamada **Relatorio**, que implemente a interface **Imprimivel**;
2. Sobrescreva o método **imprimir()**, demandado pela interface, e, nesse método, imprima uma linha com os dizeres: “**Relatorio sendo impresso**”;
3. Salve e compile.

C – Criando a classe Grafico

1. Crie uma classe chamada **Grafico**, que implemente a interface **Imprimivel**;
2. Sobrescreva o método **imprimir()**, demandado pela interface, e, nesse método, imprima uma linha com os dizeres: “**Grafico sendo impresso**”;
3. Salve e compile.

D – Criando a classe principal Cap10_Lab1

1. Crie uma classe que contenha o método **main()**, denominada **Cap10_Lab1**;
2. Dentro do método **main()**, declare e instancie dois objetos de tipo **Imprimivel**, porém, instanciando e atribuindo à cada referência um tipo específico: **Relatorio** e **Grafico**;
3. Invoque o método **imprimir()** de cada instância e observe a saída:

```
Console Problems
<terminated> Cap10_Lab1 [Java Application]
Grafico sendo impresso!
Relatorio sendo impresso!
```