

Java Programmer – Módulo I (online)

Cód.: TE 1725/0_EAD

Créditos

2

Copyright © Impacta Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este material de estudo (textos, imagens, áudios e vídeos) não pode ser copiado, reproduzido, traduzido, baixado ou convertido em qualquer outra forma eletrônica, digital ou impressa, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Impacta 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 – Módulo I (online)

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Supervisão de Desenvolvimento Digital

Alexandre Hideki Chicaoka

Produção, Gravação, Edição de Vídeo e Finalização

Bruno Michel Vasconcellos de Andrade
(Impacta Produtora)

Luiz Felipe da Silva Porto (Impacta Produtora)

Xandros Luiz de Oliveira Almeida (Impacta Produtora)

Roteirização

Sandro Luiz de Souza Vieira

Aula ministrada por

Sandro Luiz de Souza Vieira

Edição e Revisão final

Fernanda Monteiro Laneri

Diagramação

Bruno de Oliveira Santos

Edição nº 1 | 1725/0_EAD

junho/2015

*Este material é uma nova obra derivada da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Set/2014: Java Programmer
Autoria: Sandro Luiz de Souza Vieira
Nº de registro BN: 696137*

1.	Introdução à linguagem Java	09
1.1.	Histórico	10
1.2.	Características	11
1.3.	Edições disponíveis	12
1.4.	Java Development Kit (JDK)	13
1.4.1.	Java Virtual Machine (JVM)	13
1.5.	Ambientes de desenvolvimento (IDEs)	13
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	16
1.7.6.	Palavras reservadas	17
1.8.	Compilando e executando um programa	18
	Teste seus conhecimentos.....	23
2.	Introdução ao Eclipse IDE	27
2.1.	Instalando o Eclipse	28
2.2.	Iniciando o Eclipse	28
2.3.	Criando um projeto	30
2.4.	Criando uma classe	32
2.5.	Compilando e executando o código	33
3.	Tipos de dados, valores literais e variáveis	35
3.1.	Introdução	36
3.2.	Tipos de dados	36
3.2.1.	Tipos primitivos	37
3.2.1.1.	String	38
3.3.	Literais	38
3.3.1.	Literais inteiros	39
3.3.2.	Literais de ponto flutuante	40
3.3.3.	Literais booleanos	40
3.3.4.	Literais de caracteres	41
3.3.4.1.	Caracteres de escape	41
3.3.5.	Literais de strings (cadeia de caracteres)	42
3.4.	Variáveis	43
3.4.1.	Definindo uma variável	43
3.4.2.	Declarando uma variável	44
3.4.2.1.	Usando o qualificador final	45
3.4.3.	Escopo de variáveis	45
3.4.3.1.	Aninhando escopos	46
3.5.	Casting	47
	Teste seus conhecimentos.....	49

Java Programmer – Módulo I (online)

4

4.	Operadores	53
4.1.	Introdução	54
4.2.	Operadores aritméticos	56
4.2.1.	Operadores aritméticos de atribuição reduzida	57
4.3.	Operadores incrementais e decrementais	58
4.4.	Operadores relacionais.....	60
4.5.	Operadores lógicos	61
4.6.	Operador ternário	62
4.7.	Precedência dos operadores.....	63
	Teste seus conhecimentos.....	65
5.	Controle de fluxo	69
5.1.	Introdução	70
5.1.1.	if/else	70
5.1.2.	switch	73
5.2.	Estruturas de repetição	76
5.2.1.	While.....	76
5.2.2.	Do/while	77
5.2.3.	For	78
5.3.	Outros comandos.....	83
5.3.1.	Break	83
5.3.1.1.	Instruções rotuladas.....	84
5.3.2.	Continue	86
	Teste seus conhecimentos.....	87
6.	Métodos	91
6.1.	Introdução	92
6.2.	Estrutura de um método	92
6.2.1.	Comando return.....	93
6.2.2.	Um método na prática.....	93
6.3.	Chamando um método	94
6.4.	Passagem de parâmetros	96
	Teste seus conhecimentos.....	99
7.	Classes e orientação a objetos	103
7.1.	Introdução	104
7.2.	Objetos	104
7.2.1.	Atributo	104
7.2.2.	Método	105
7.2.3.	Mensagem	106
7.2.4.	Classe	106
7.3.	Classes	109
7.3.1.	Pacotes	109
7.3.1.1.	Criando um pacote.....	111
7.3.1.2.	Acessando uma classe em outro pacote	112
7.3.2.	Considerações ao declarar uma classe	113
7.3.3.	Encapsulamento.....	114
7.3.4.	Tipos construídos	114
7.3.5.	Instanciação	115

7.3.6.	Atribuição entre objetos de tipos construídos	115
7.3.7.	Variáveis não inicializadas.....	118
7.3.7.1.	Variáveis locais não inicializadas	119
7.3.7.2.	Variáveis de classe não inicializadas.....	119
7.3.8.	Acesso	120
7.3.8.1.	Padrão (Default)	121
7.3.8.2.	Público (Public)	122
7.3.9.	UML – Diagrama de classes	123
7.3.9.1.	Diagrama de classes em detalhes.....	124
7.4.	Métodos.....	124
7.4.1.	Acesso a métodos	124
7.4.2.	Modificadores de métodos	125
7.4.3.	this	126
7.4.4.	Métodos recursivos	127
7.4.5.	Métodos acessores.....	128
7.4.5.1.	Método getter	129
7.4.5.2.	Método setter	130
7.4.6.	Método main().....	130
7.4.7.	Sobrecarga de métodos.....	132
	Teste seus conhecimentos.....	135

8.	Construtores	141
8.1.	Introdução	142
8.2.	Construtor padrão.....	142
8.3.	Considerações sobre os construtores	145
	Teste seus conhecimentos.....	147

9.	Membros estáticos	151
9.1.	Modificador static	152
9.2.	Atributos estáticos	152
9.3.	Métodos estáticos	153
9.4.	Exemplos práticos de membros estáticos	155
	Teste seus conhecimentos.....	157

10.	Herança	161
10.1.	Introdução	162
10.2.	Herança e generalização	163
10.3.	Ligações	165
10.4.	Associação	165
10.4.1.	Tipos de associação	166
10.4.1.1.	Agregação.....	166
10.4.1.2.	Composição	167
10.5.	Herança e composição	167
10.6.	Estabelecendo herança entre classes	168
10.6.1.	Acesso aos membros da superclasse	171
10.6.2.	O operador super.....	173
10.6.3.	Chamada ao construtor da superclasse	175
10.7.	Relacionamentos	177
10.7.1.	Relacionamento baseado na herança	177
10.7.2.	Relacionamento baseado na utilização (Composição)	178

Java Programmer – Módulo I (online)

6

10.8.	Herança e classes.....	180
10.8.1.	Classe Object	180
10.8.2.	Classes abstratas	180
10.8.2.1.	Métodos abstratos	181
10.8.3.	Classes finais	183
10.9.	Polimorfismo	185
10.9.1.	Ligação tardia (late binding)	185
10.9.2.	Polimorfismo em métodos declarados na superclasse	186
10.9.3.	Operador instanceof	189
	Teste seus conhecimentos.....	191
11.	Interfaces	195
11.1.	O conceito de interface	196
11.2.	Variáveis de referência	197
11.3.	Variáveis inicializadas	198
11.4.	Métodos estáticos	199
11.5.	Métodos default	200
	Teste seus conhecimentos.....	201

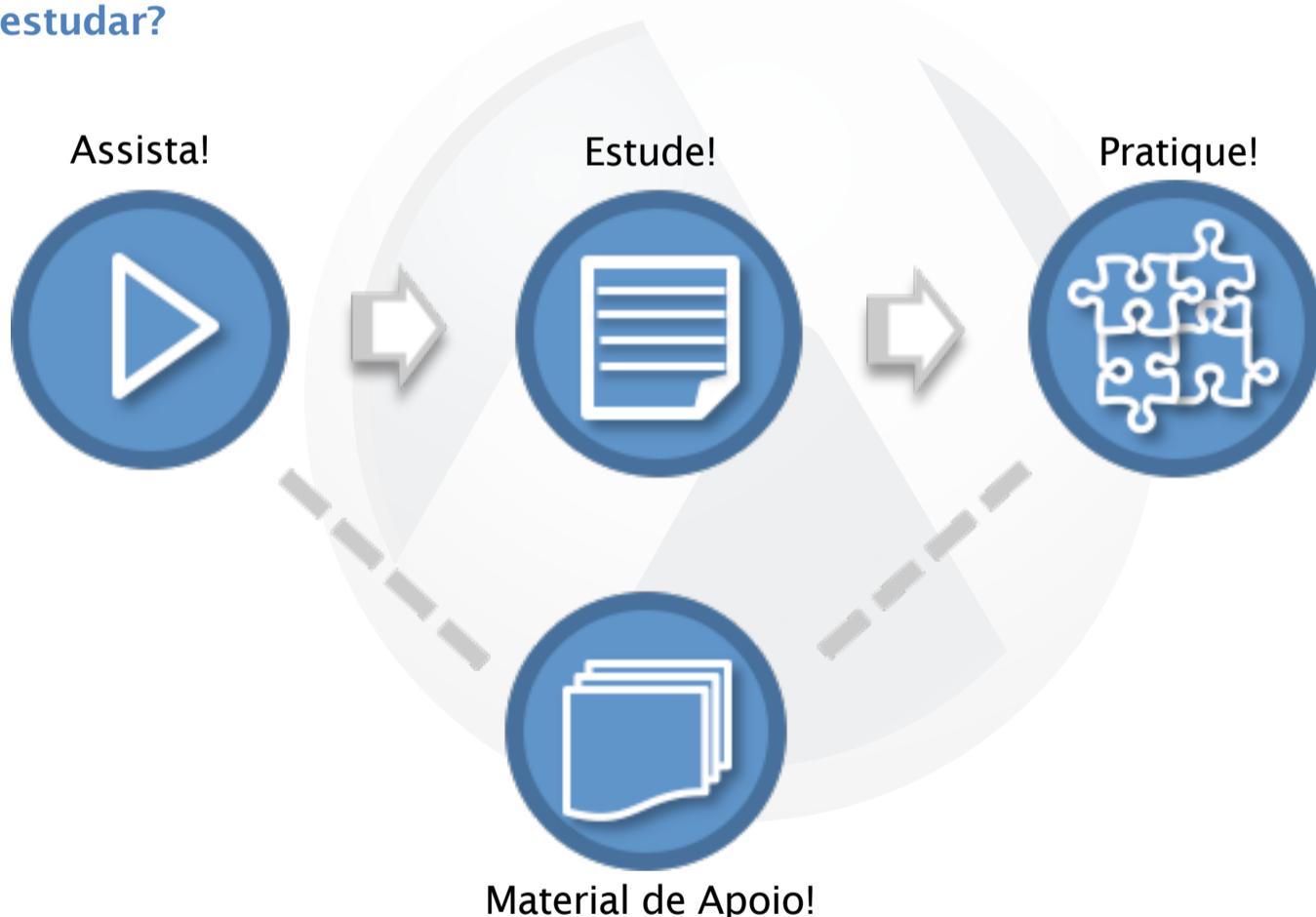
Bem-vindo!

É um prazer tê-lo como aluno do nosso curso online de Java Programmer – Módulo I. Este é o curso ideal para você, experiente ou não em outras linguagens de programação, que deseja começar a utilizar a linguagem Java no desenvolvimento de aplicações.

No decorrer das aulas, você estudará características e recursos essenciais da linguagem Java. Conhecerá tipos de dados, valores literais e variáveis, além dos tipos de operadores mais utilizados em Java. Também, verá como utilizar estruturas condicionais e de repetição, métodos, classes e orientação a objetos, construtores, membros estáticos e herança. Por fim, verá como utilizar interfaces, um recurso que garante o uso de métodos específicos por parte das classes.

Para ter um bom aproveitamento deste curso, é imprescindível que você tenha participado do nosso curso de Introdução à Lógica de Programação, ou possua conhecimentos equivalentes. Bom aprendizado!

Como estudar?



Este curso conta com:

- Videoaulas sobre os assuntos que você precisa saber no primeiro módulo de Java Programmer.
- Parte teórica, com mais exemplos e detalhes para você que quer se aprofundar no assunto da videoaula.
- Exercícios de testes para você pôr à prova o que aprendeu.
- Material de apoio para você testar os exemplos das videoaulas.

Introdução à linguagem Java

1

- ✓ Histórico;
- ✓ Características;
- ✓ Edições disponíveis;
- ✓ Java Development Kit;
- ✓ Ambientes de desenvolvimento;
- ✓ Estrutura básica de um programa Java;
- ✓ Características do código;
- ✓ Compilação e execução de um programa.

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 amplamente 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 relativamente 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 (J2EE), 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.
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 a desempenho, estabilidade, segurança, ao Java Plug-in e à Java Virtual Machine.
8	É a versão atual da linguagem. Traz as expressões Lambda como uma de suas maiores inovações, além de uma nova API para manipulação de data e hora.



A Java 8 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. Hoje, quase todas as linguagens trabalham dessa forma.

- **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. Esse 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 plataformas 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: primeiro, a ausência de ponteiros (característicos do C++), evitando falhas comuns a esse tipo de recurso; checagens realizadas em tempo de execução; a declaração explícita de métodos; 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; a sandbox, que mantém um código executado em um espaço isolado, com diversas restrições, evitando que ele danifique o ambiente Java; 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.

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 e máquinas mais simples;
- **JEE (Java Enterprise Edition)**: Edição destinada ao desenvolvimento de aplicações baseadas em servidor;
- **JME (Java Micro Edition)**: Edição para dispositivos com menor poder de processamento e memória, como dispositivos móveis;
- **JavaFX**: Plataforma para desenvolvimento de aplicações ricas, executáveis em diferentes dispositivos, permitindo interfaces gráficas mais dinâmicas. Possibilita a criação de interfaces gráficas para diversos ambientes (desktops, browsers, celulares etc.), suportando qualquer biblioteca Java.

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**, códigos utilizados como exemplos e o **appletviewer**, que permite a execução de applets.



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 de desenvolvimento gráfico que podem auxiliar no desenvolvimento 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. Pode ser baixada em <http://www.embarcadero.com/products/jbuilder>.

- **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 amplamente 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 (_), cífrão (\$), ou por letras. Para os demais caracteres, podem ser incluídos, na lista anterior, os números. Dentre os identificadores, temos as classes, os métodos e as variáveis, entre outros.

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.

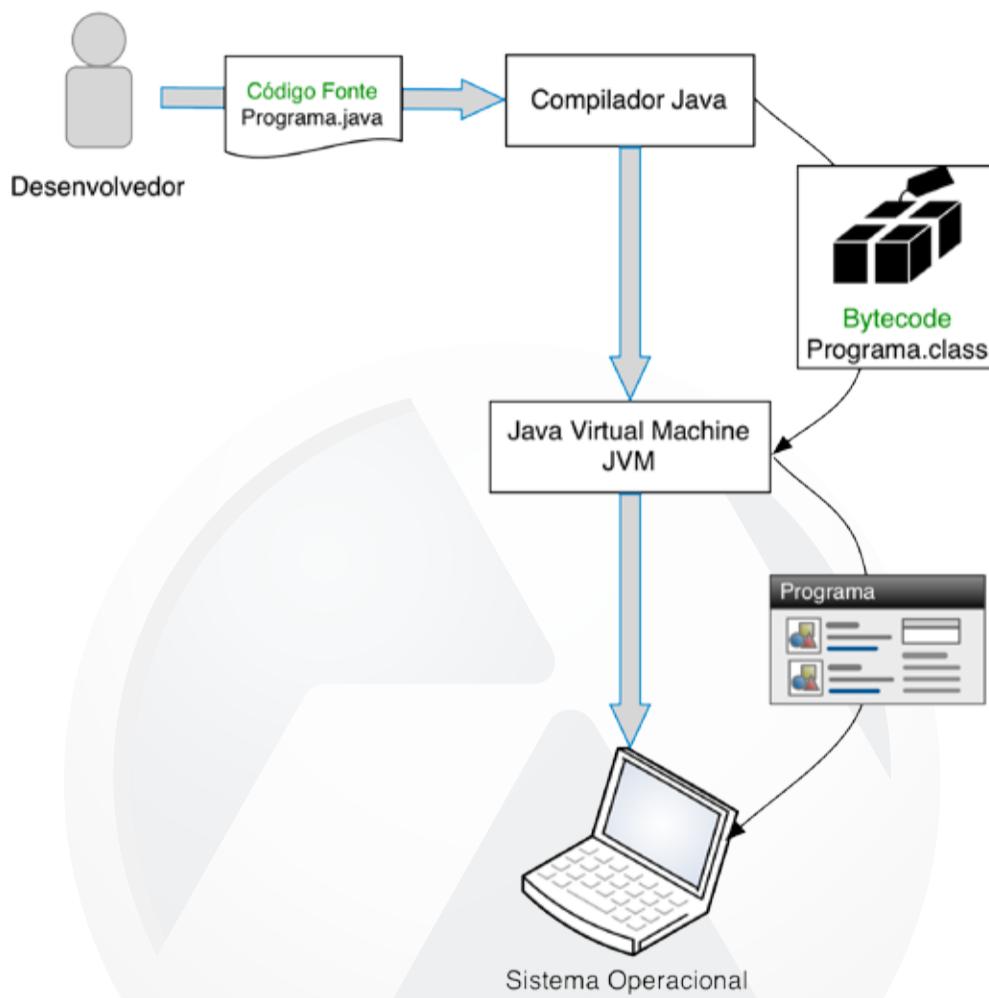
A seguir, as palavras reservadas em Java 8:

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.

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:

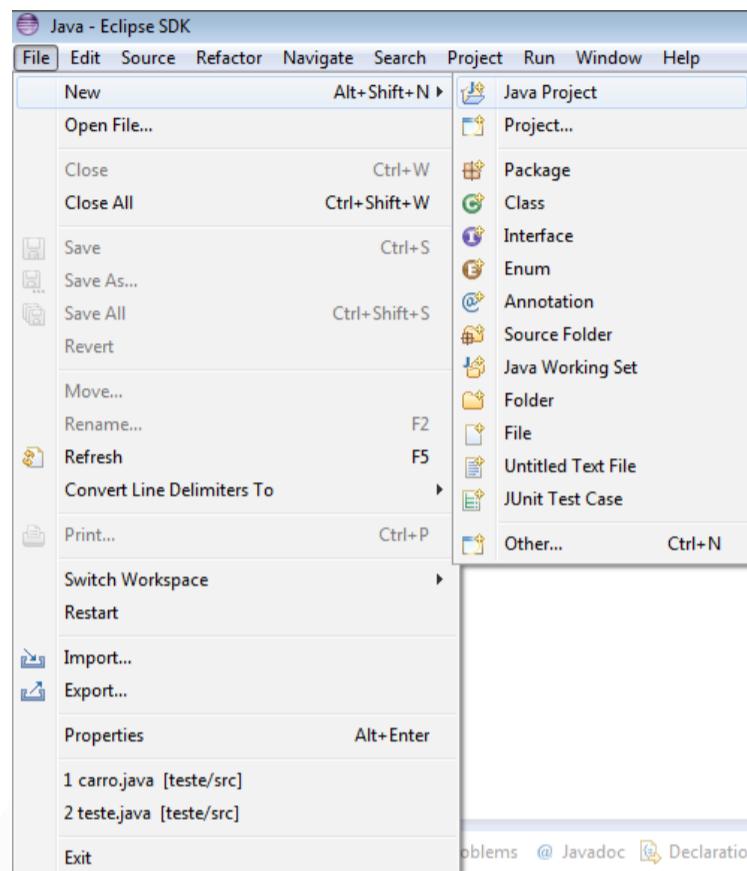


Note que, ao passar pelo compilador Java, também conhecido como “javac”, o código-fonte existente no arquivo **Programa.java** é inteiramente analisado e, caso não haja erros, o compilador gera um arquivo de mesmo nome, porém com a extensão **.class**: **Programa.class**. Esse arquivo contém instruções 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.

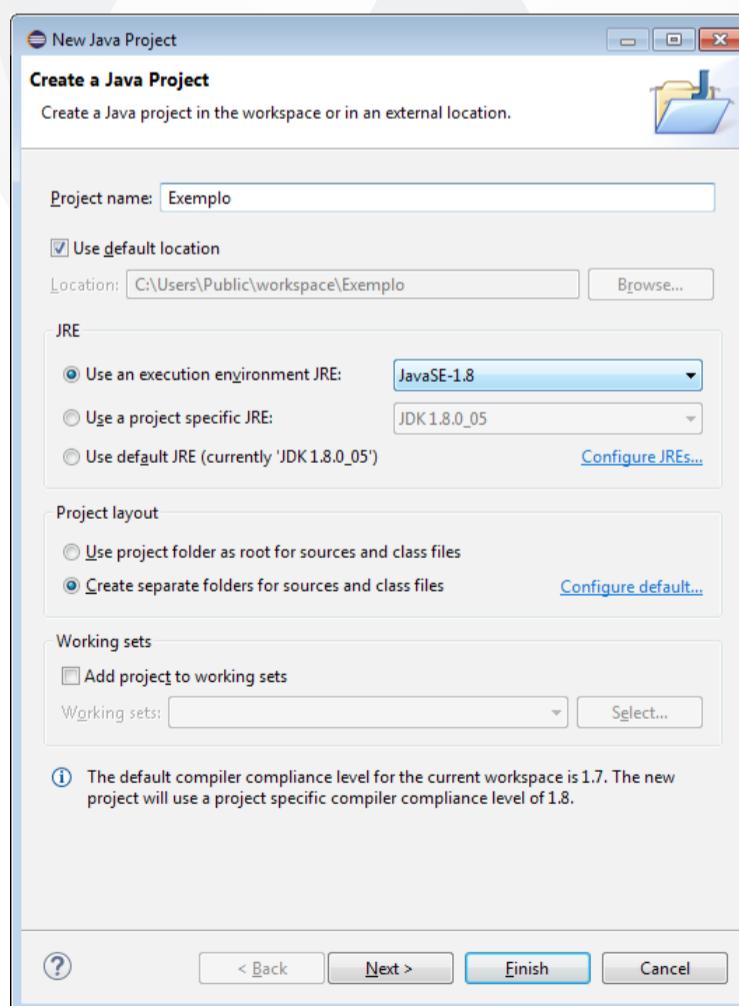
Após o processo de compilação, o bytecode gerado é executado pela JVM, que o transforma em binário executável para o Sistema Operacional de base. Nessa etapa, acontece a interpretação do código, feita linha a linha e, na incidência de erros, a JVM lança as chamadas exceções, que serão abordadas posteriormente.

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

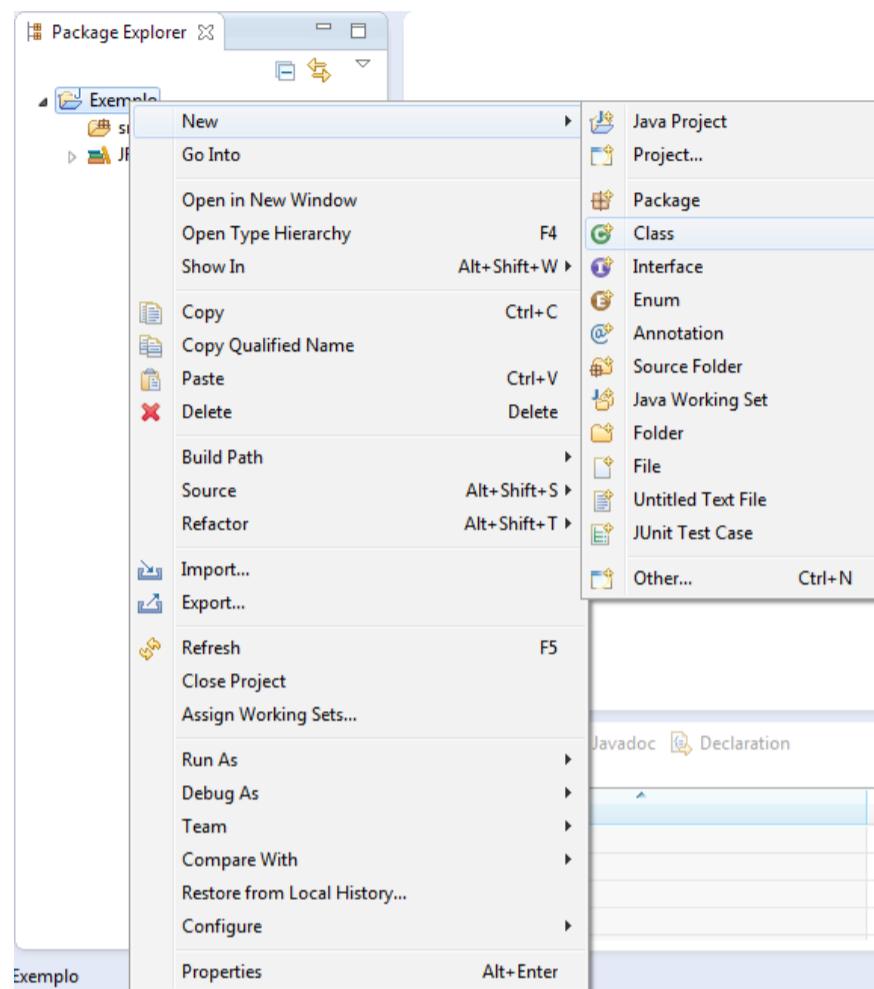
1. No menu File, selecione New / Java Project:



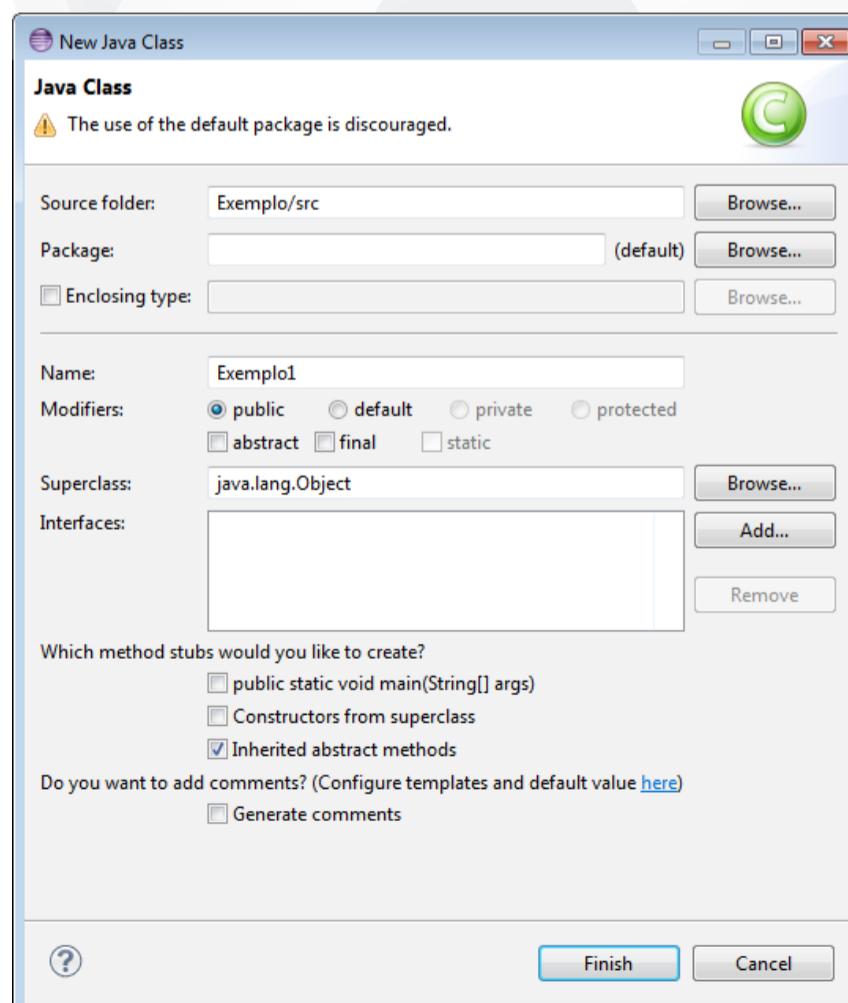
2. Na janela New Java Project, nomeie o projeto e clique em Finish:



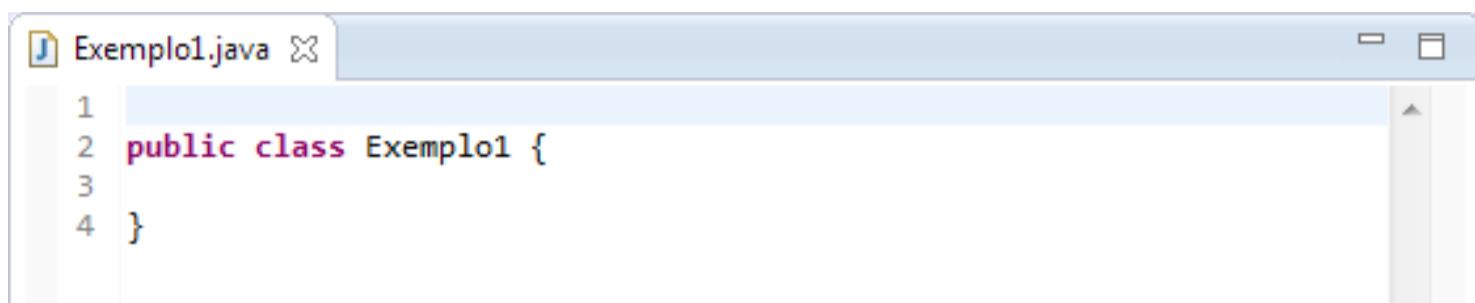
3. Clique com o botão direito do mouse no projeto criado e escolha New / Class:



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

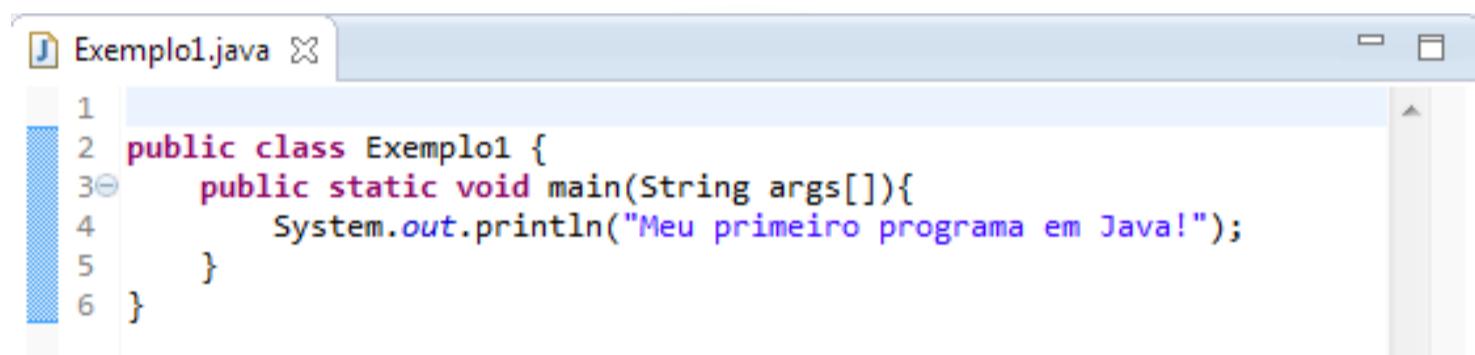


A classe será criada e aparecerá da seguinte forma no editor de código:



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

5. Digite o seguinte código no editor:



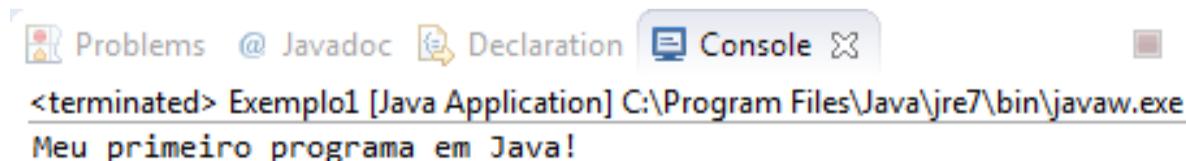
```
1  
2 public class Exemplo1 {  
3     public static void main(String args[]){  
4         System.out.println("Meu primeiro programa em Java!");  
5     }  
6 }
```

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:



```
<terminated> Exemplo1 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe  
Meu primeiro programa em Java!
```


Teste seus conhecimentos

Introdução à linguagem Java

1

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. Qual é o tipo de extensão de um arquivo Java?

- a) .java
- b) .jvm
- c) .jav
- d) .sun
- e) Nenhuma das alternativas anteriores está correta.

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.

Introdução ao Eclipse IDE

2

- ✓ Instalando o Eclipse;
- ✓ Iniciando o Eclipse;
- ✓ Criando um projeto;
- ✓ Criando uma classe;
- ✓ Compilando e executando o código.

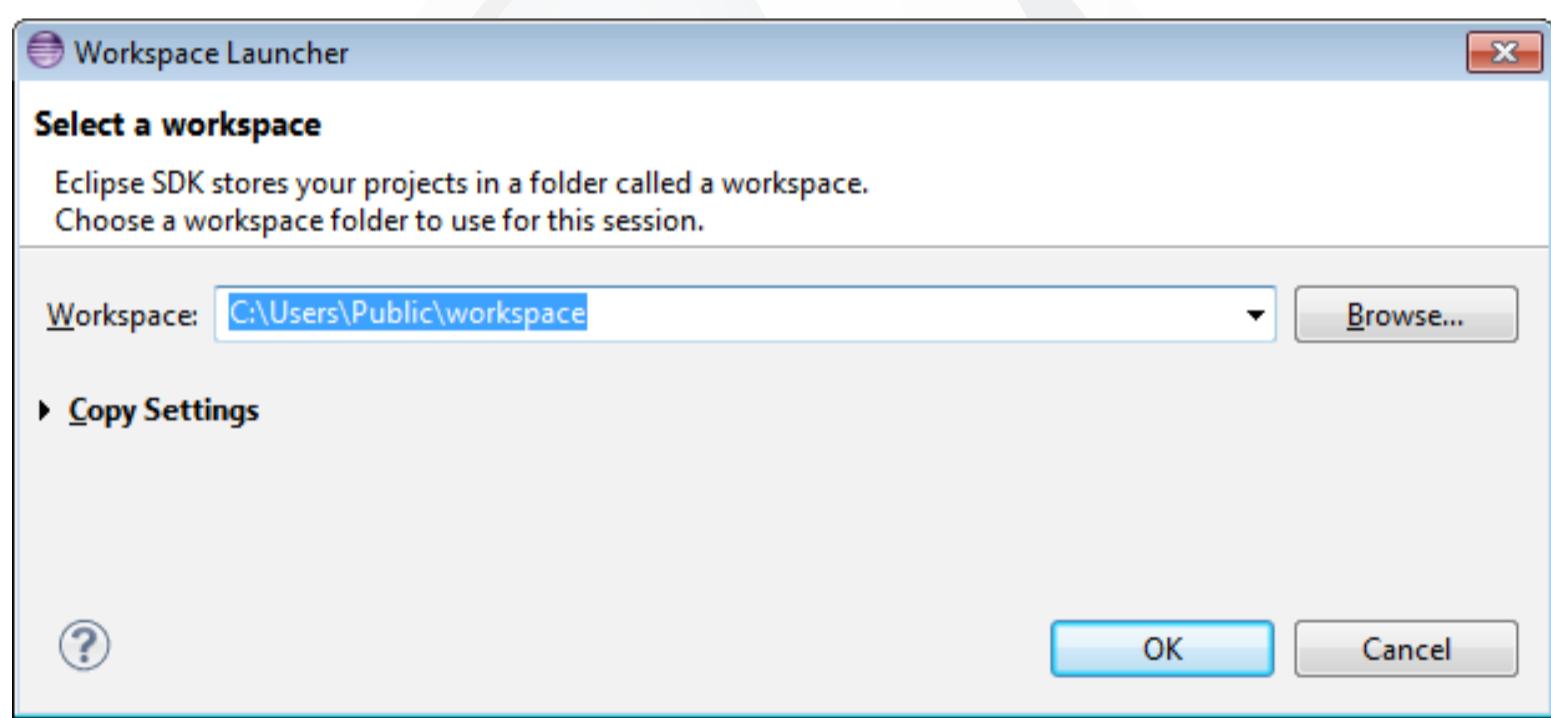
2.1. Instalando o Eclipse

Neste treinamento, vamos utilizar o Eclipse IDE, cujo download pode ser feito a partir do link www.eclipse.org. A versão empregada neste treinamento é **Luna 4.4**.

Concluído o download, basta descompactar o Eclipse em um diretório. Recomendamos que você utilize os drives **C:**, **E:**, **F:**, entre outros, evitando descompactá-lo na área de trabalho.

Após descompactá-lo em um diretório, será criada uma pasta denominada **eclipse**. Para acessar o Eclipse, então, basta clicar no ícone **eclipse.exe**.

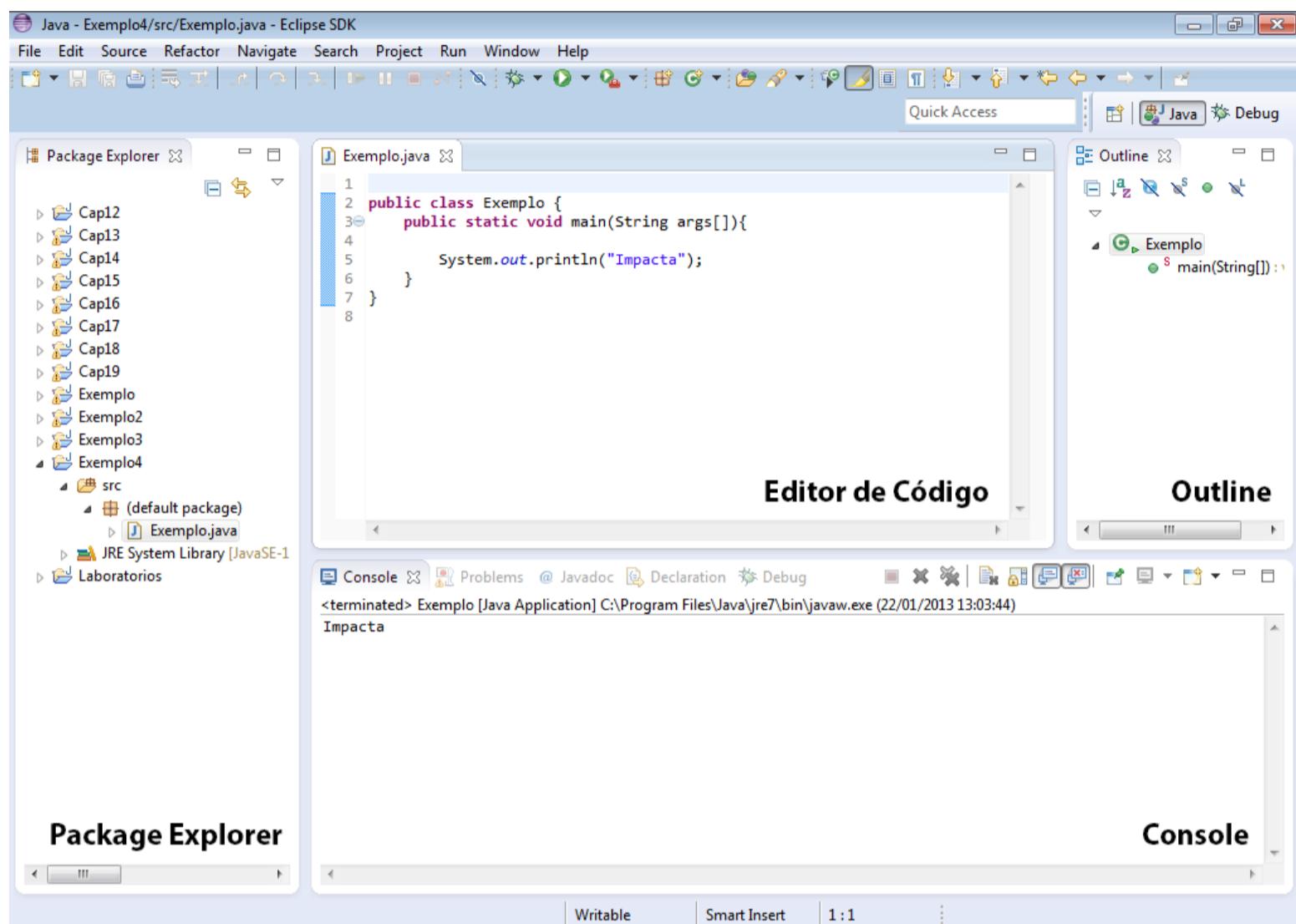
Em sua execução inicial, o Eclipse pede para nomear o workspace, ou seja, a área de trabalho. Para isso, basta alterar o caminho para o diretório desejado, conforme apresentado na imagem adiante:



2.2. Iniciando o Eclipse

Ao clicar no ícone de execução do Eclipse, será exibida uma tela de boas-vindas ao programa. Ao fechá-la, será exibida a tela inicial do ambiente, a partir da qual iremos trabalhar os projetos e códigos Java.

A tela inicial do ambiente Eclipse é dividida em algumas seções, como mostra a imagem a seguir:



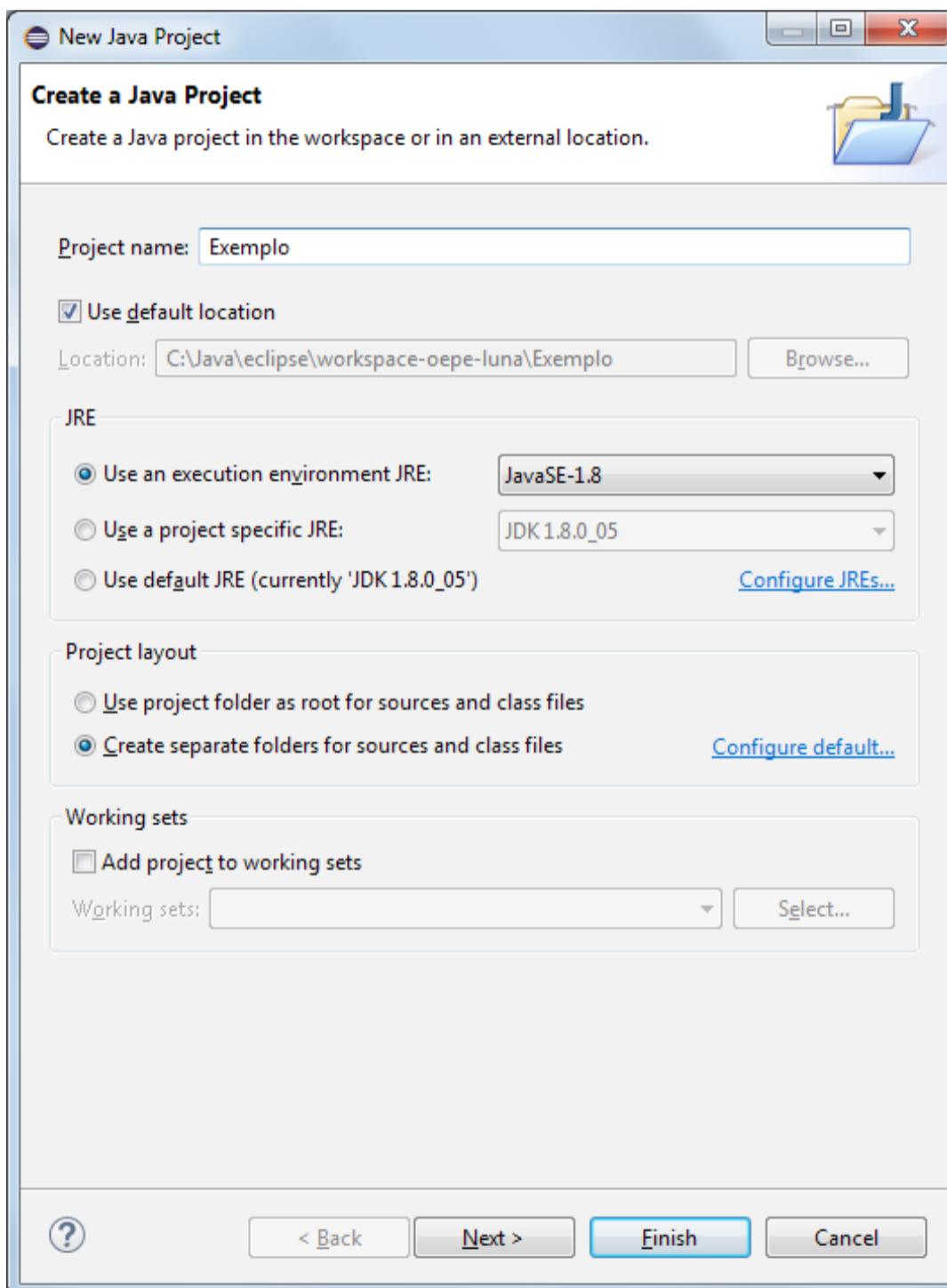
Estas são as principais seções da tela inicial do Eclipse:

- **Package Explorer:** Aqui você pode visualizar o projeto e navegar por toda a sua estrutura;
- **Editor de código:** Nesta seção, você insere e edita os códigos do programa, de acordo com a funcionalidade a ser atingida;
- **Outline:** Esta seção é semelhante ao Package Explorer, mas centrada no acesso à estrutura interna do arquivo Java. As seções do arquivo são destacadas por ícones;
- **Console:** Nesta área, você visualiza a saída gerada após a compilação e execução do código.

2.3. Criando um projeto

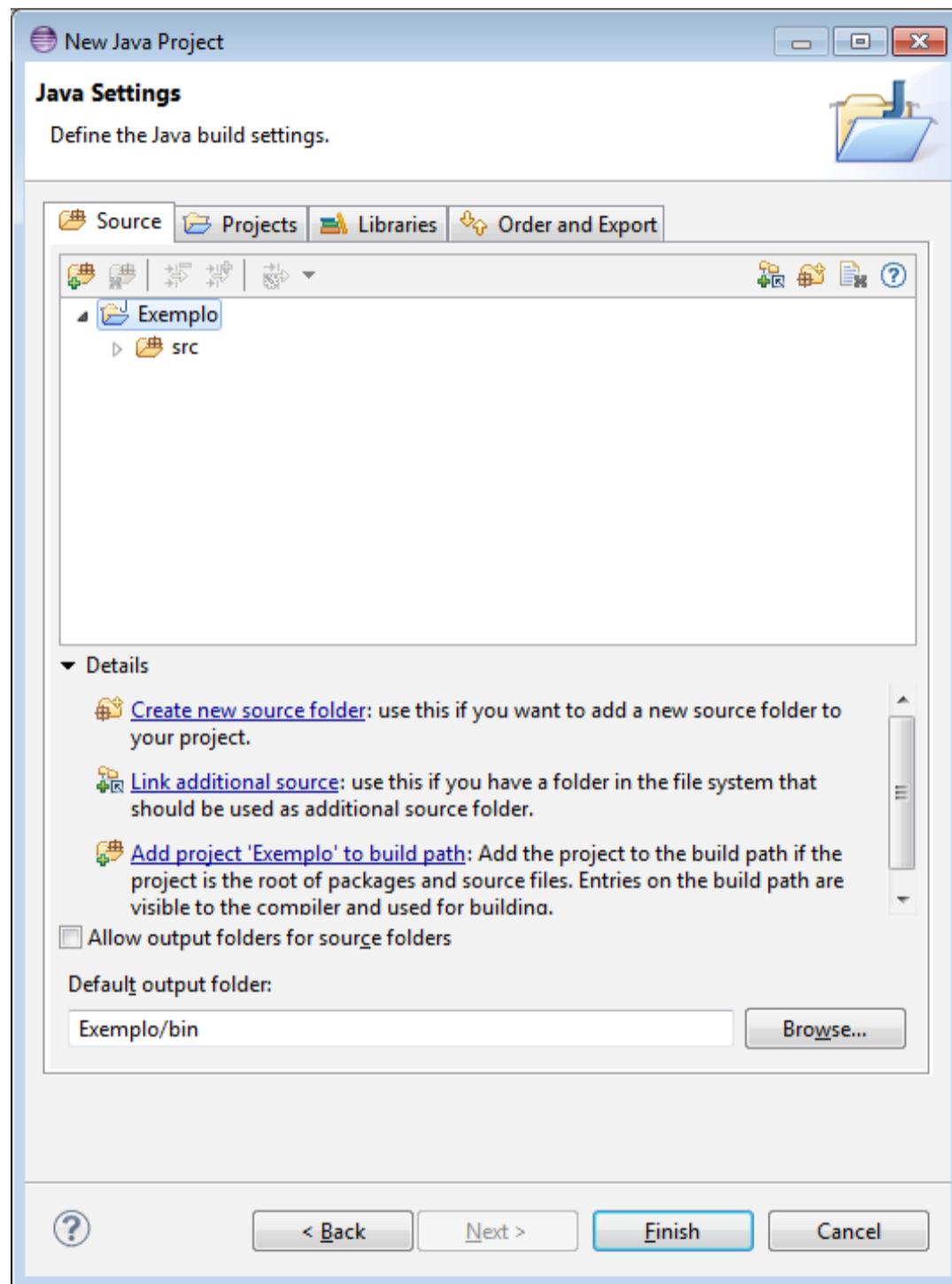
Para criar um projeto, utilize o seguinte procedimento:

1. Clique em **File / New / Java Project**:



2. Na janela **New Java Project**, defina um nome para o projeto e confirme as opções oferecidas (**JRE** e **Project layout**);

3. Clique em **Next**. A seguinte tela será exibida:

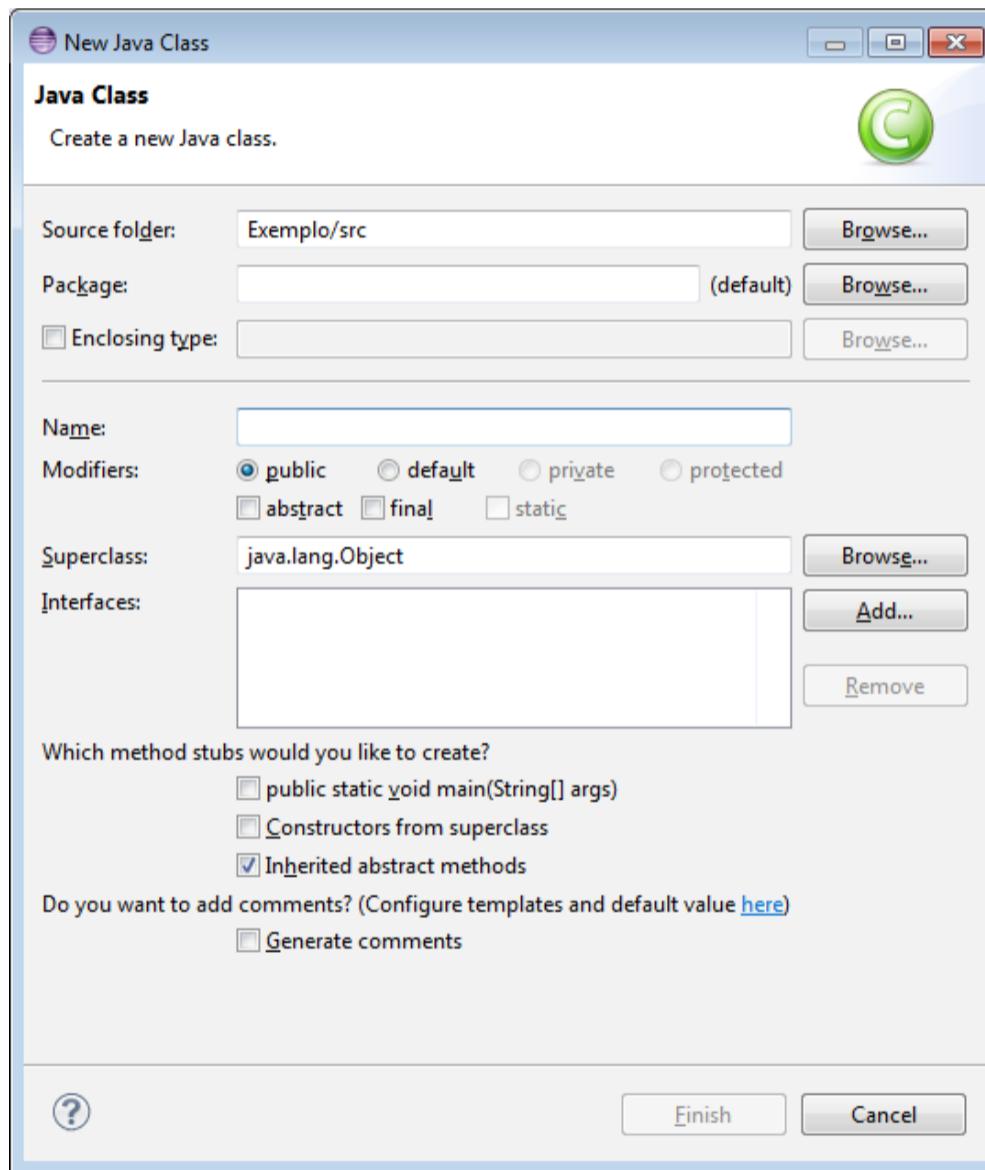


4. Ajuste as configurações de Java e, então, clique em **Finish**. O projeto será criado e exibido no **Package Explorer**.

2.4. Criando uma classe

Veja os procedimentos que você deve adotar para a criação de classes:

1. Clique no menu **File / New / Class**. A mesma opção pode ser acessada ao clicar com o botão direito do mouse sobre o **Package Explorer**:



2. Defina um nome para o pacote e a classe;
3. Clique em **Finish**. O arquivo será criado na pasta **src** e aberto para edição.

2.5. Compilando e executando o código

Depois que você escrever e salvar o código, automaticamente o Eclipse compilará e exibirá eventuais erros. Esse tipo de compilação é denominado Incremental.

Para executar o código, basta clicar no botão de execução (botão verde com símbolo de play), disposto na barra de ferramentas do Eclipse. Você também pode utilizar o menu **Run / Run**.

Na execução inicial, selecione a opção **Java Application**. O resultado da execução será exibido no **Console** do Eclipse.

Tipos de dados, valores literais e variáveis

3

- ✓ Tipos de dados;
- ✓ Literais;
- ✓ Variáveis;
- ✓ Casting.

3.1. Introdução

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

3.2. Tipos de dados

Java é uma linguagem estaticamente tipada, ou seja, cada expressão e variável possuem 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.

Nesta leitura complementar, abordaremos apenas os tipos primitivos.

3.2.1. Tipos primitivos

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 extensos.
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 extensos.
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.

3.2.1.1. String

Além dos oito tipos primitivos apresentados, Java oferece suporte para sequências de caracteres por meio do tipo especial **String**.

Quando você coloca um string 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.

Tecnicamente falando, **String** não é um tipo primitivo de dado, mas uma classe. Contudo, ele parece funcionar desse modo e nós acabamos pensando nele dessa maneira, por causa do suporte especial que recebe no contexto da linguagem.

3.3. Literais

Um literal é uma representação, em código-fonte, de um valor fixo. Eles não necessitam de nenhuma operação computacional para serem representados diretamente; 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.

3.3.1. 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 é de tipo **int**, que permite criar tipos **byte**, **short** e **long**, além de **int**. É possível, também, criar literais inteiros de tipo **long**. Eles servem para criar valores do tipo **long** 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**) 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
```

3.3.2. Literais de ponto flutuante

Os números de ponto flutuante representam valores decimais compostos de fração. Podem ser representados por um numeral inteiro, seguido por 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 por 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** pode ser usados para a mesma finalidade, porém, diferentemente do caso anterior, não é obrigatório.

Veja um exemplo de código com literal de ponto flutuante:

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

3.3.3. 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;
```

3.3.4. 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'
```

3.3.4.1. Caracteres de escape

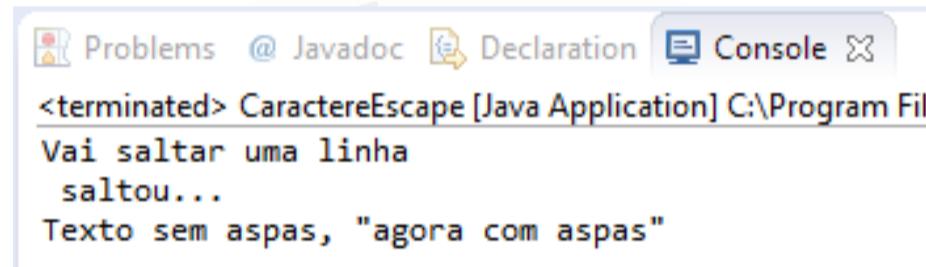
Existem, contudo, caracteres especiais que não podem ser representados diretamente dentro de aspas simples. Não podem ser apresentados visualmente, mas servem para executar alguma ação predefinida. Eles são conhecidos como caracteres ou sequências de escape. Eles são compostos por uma barra invertida 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  
<terminated> CaractereEscape [Java Application] C:\Program Fil  
Vai saltar uma linha  
saltou...  
Texto sem aspas, "agora com aspas"
```

3.3.5. Literais de strings (cadeia de caracteres)

Um literal de string é composto por zero ou mais caracteres entre aspas duplas. Eles podem ser representados, também, por sequências de escape. Como são um tipo de objeto e não arrays de caracteres, Java dispõe de recursos avançados e práticos para gerenciamento de strings.

Literais de string ocupam sempre uma mesma linha. Não há uma sequência de escape para continuação de linha, como em outras linguagens.

Veja um exemplo de literal de string:

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

3.4. 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.

3.4.1. 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:

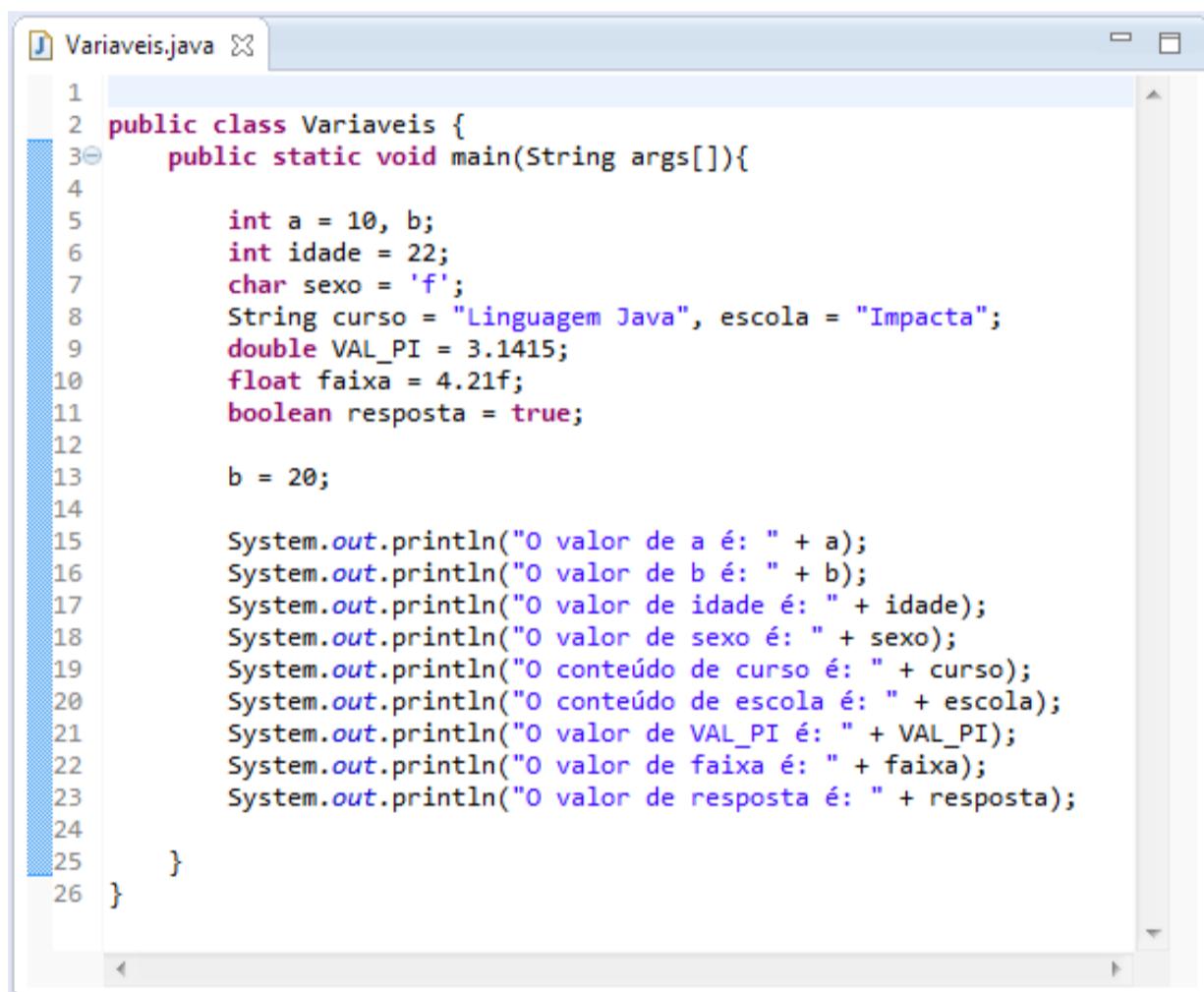
- Ela deve iniciar, obrigatoriamente, com um caractere alfabético (letra) maiúsculo ou minúsculo, underline (_) ou cifrão (\$);
- Os demais caracteres do identificador podem ser letras maiúsculas ou minúsculas, números, underlines (_) ou cifrões (\$);
- Não podem ser usados espaços em branco nem quaisquer 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.

3.4.2. Declarando uma variável

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

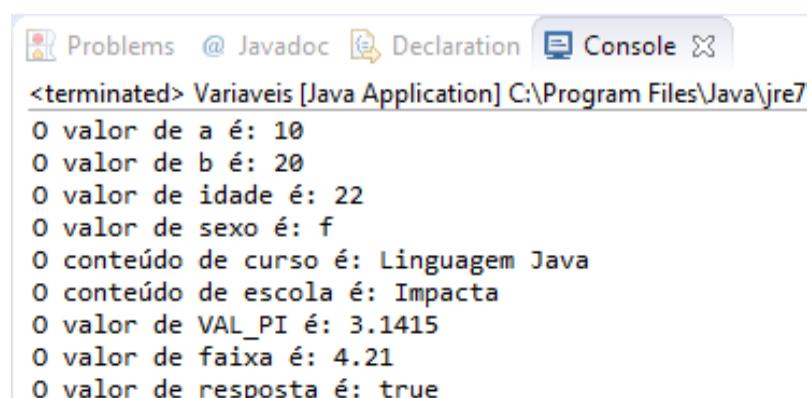
```
<tipo> <nomeVariável>;  
<tipo> <nomeVariável> = <valor>;  
<tipo> <nomeVariável1>, <nomeVariável2>;  
<tipo> <nomeVariável1> = <valor>, <nomeVariável2> = <valor>;
```

Veja alguns exemplos de declaração de variáveis:



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

Segue o resultado:



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

3.4.2.1. 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 amplamente 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 à frente de uma classe, tornando-a privada, evitando que seus atributos e métodos sejam herdados; e à frente de um método, evitando que ele seja sobreescrito.

3.4.3. 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.

Uma variável pode ser declarada em qualquer ponto dentro de qualquer bloco iniciado. Uma variável dentro de um escopo é protegida contra acessos e alterações não autorizadas e é inacessível para um código definido fora desse escopo.

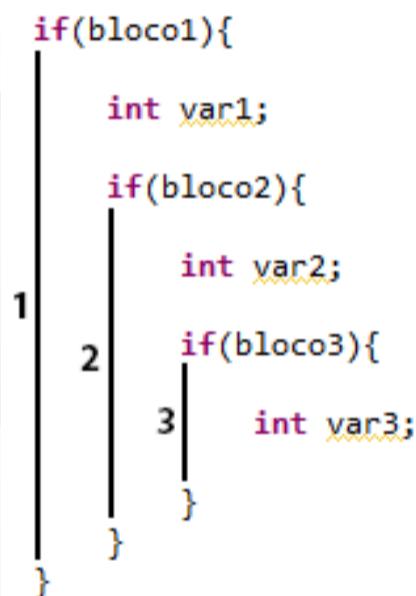
Os dois escopos mais usualmente empregados em Java são o escopo definido por classe e o escopo definido por método. Uma variável é criada assim que seu escopo é acessado por um programa. Ao sair do escopo, a variável é destruída e, consequentemente, não preserva seu valor. O valor de uma variável também não é mantido quando esta sai de um bloco, o que significa que o escopo é o fator decisivo para a duração da variável. Quando temos variáveis declaradas dentro de um método, chamadas variáveis locais, elas não preservam seus valores para duas chamadas feitas a um mesmo método.

Uma variável declarada no início de um método estará acessível para o código completo desse método, e uma variável declarada que possua um inicializador será reinicializada toda vez que o bloco onde ela estiver for executado.

3.4.3.1. Aninhando escopos

Quando criamos um bloco de código, o Java gera um novo escopo aninhado. Isso significa que o escopo externo engloba o escopo interno, ou seja, o código contido no escopo interno pode visualizar os objetos que foram declarados no escopo externo. Já os objetos declarados no escopo interno não podem ser vistos fora do mesmo.

A seguir, temos uma representação de blocos, com as variáveis declaradas dentro dos mesmos:



De acordo com o exemplo anterior, os números 1, 2 e 3 indicam os blocos existentes. O **bloco 1** contém o **bloco 2**, que, por sua vez, contém o **bloco 3**. O **bloco 1** é o mais externo e contém os outros dois blocos. O **bloco 3** é o mais interno.

Dessa forma, **var1** pode ser referenciado dentro dos blocos 1, 2 e 3, **var2** só pode ser referenciado dentro dos blocos 2 e 3, e **var3** somente é visível no próprio bloco 3. Nesses casos, temos uma variável declarada em um bloco mais externo, com referência em um bloco mais interno. Se tentássemos, porém, referenciar **var2** no **bloco 1**, isso geraria um erro de compilação.

3.5. 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;
```

 Type mismatch: cannot convert from double to float

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

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

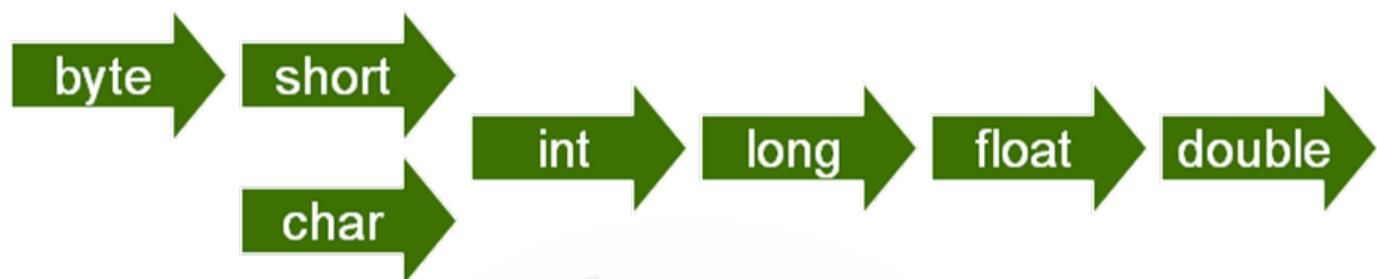
```
long ano = 2014;
int anoAtual = (int) ano;

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

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



Teste seus conhecimentos

Tipos de dados, valores literais e variáveis

3

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.

Operadores

4

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

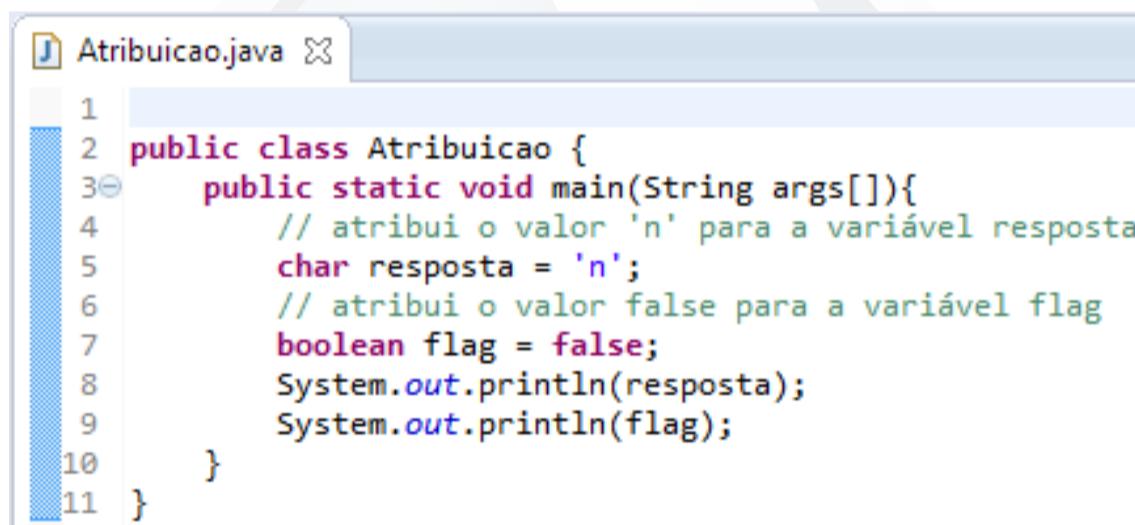
4.1. Introdução

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 desta leitura complementar, veremos os principais tipos de operadores.

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:

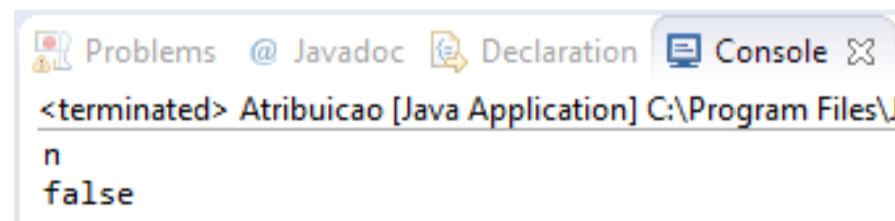
```
<tipo> <nomeVariávelOuConstante> = <valorAtribuição>;
```

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
```

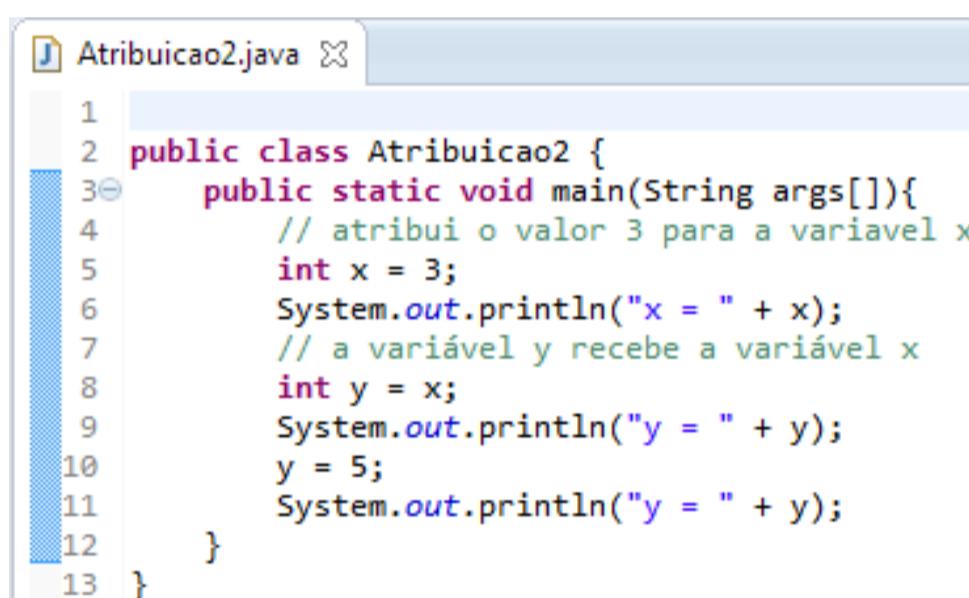
Uma das possibilidades que temos com esse operador é atribuir uma variável primitiva a outra variável primitiva. Veja como isso ocorre no exemplo a seguir:

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

Em que:

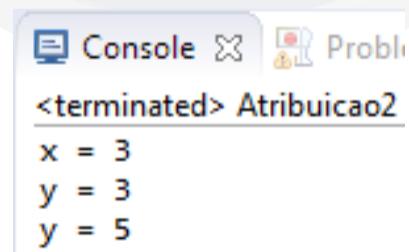
- **x** recebe o valor **3**;
- **y** recebe a variável **x**, logo, **y** contém o valor **3**.

Neste momento, as duas variáveis (**x** e **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:



```
J Atribuicao2.java X
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 variavel y recebe a variavel 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:



```
Console X Proble
<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:

```
endereco = "Rua X";
```

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

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

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

Dessa 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**.

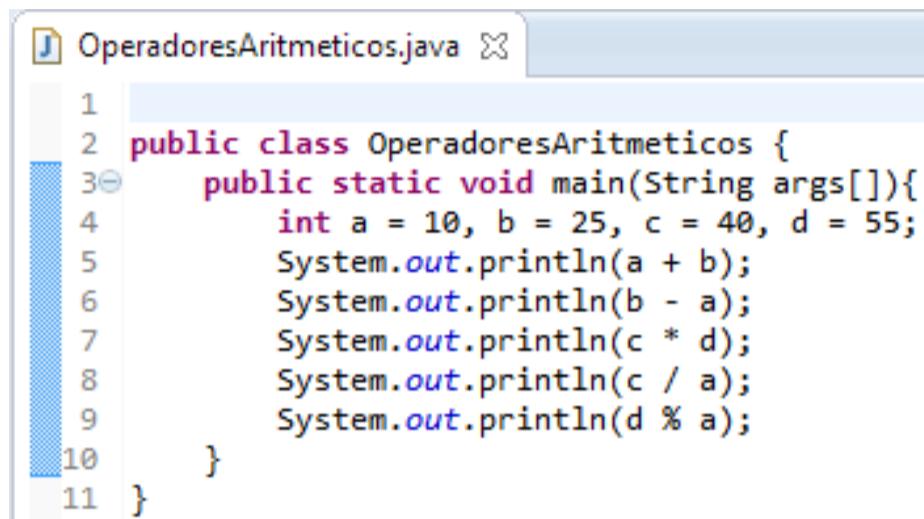
4.2. Operadores aritméticos

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

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

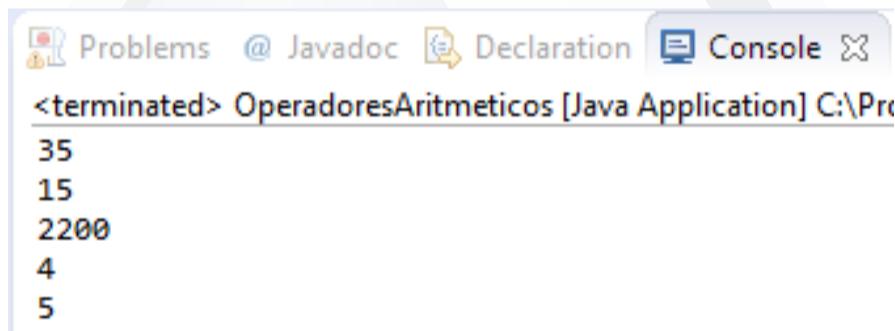
 Os operadores + e - também podem significar, respectivamente, positivo e negativo, ou manutenção e inversão de sinal, em casos como +x e -y, em que x e y são as variáveis.

O exemplo a seguir mostra como podemos usar esses operadores:



```
1
2 public class OperadoresAritmeticos {
3     public static void main(String args[]){
4         int a = 10, b = 25, c = 40, d = 55;
5         System.out.println(a + b);
6         System.out.println(b - a);
7         System.out.println(c * d);
8         System.out.println(c / a);
9         System.out.println(d % a);
10    }
11 }
```

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



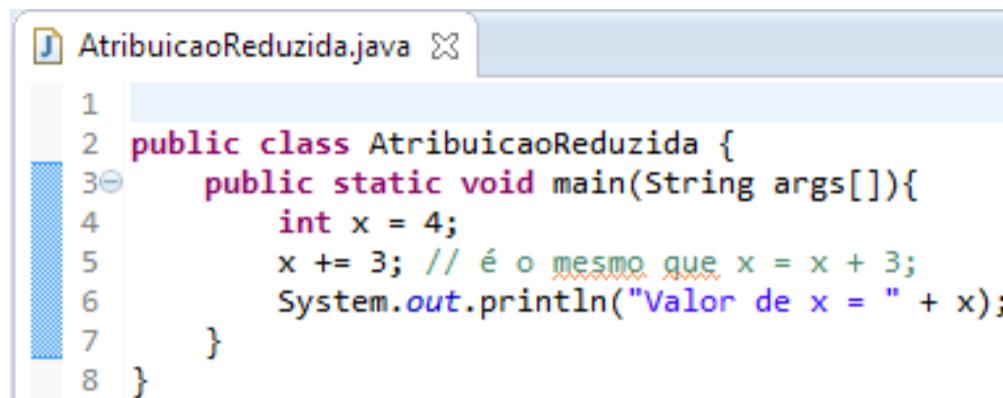
```
35
15
2200
4
5
```

4.2.1. 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:

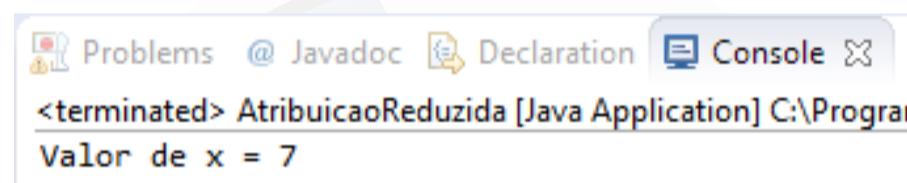
Operador aritmético	Operação equivalente	Descrição
$x += 3$	$x = x + 3$	Mais igual
$x -= 3$	$x = x - 3$	Menos igual
$x *= 3$	$x = x * 3$	Vezes igual
$x /= 3$	$x = x / 3$	Dividido igual
$x %= 3$	$x = x \% 3$	Módulo igual

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



```
<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.

4.3. Operadores incrementais e decrementais

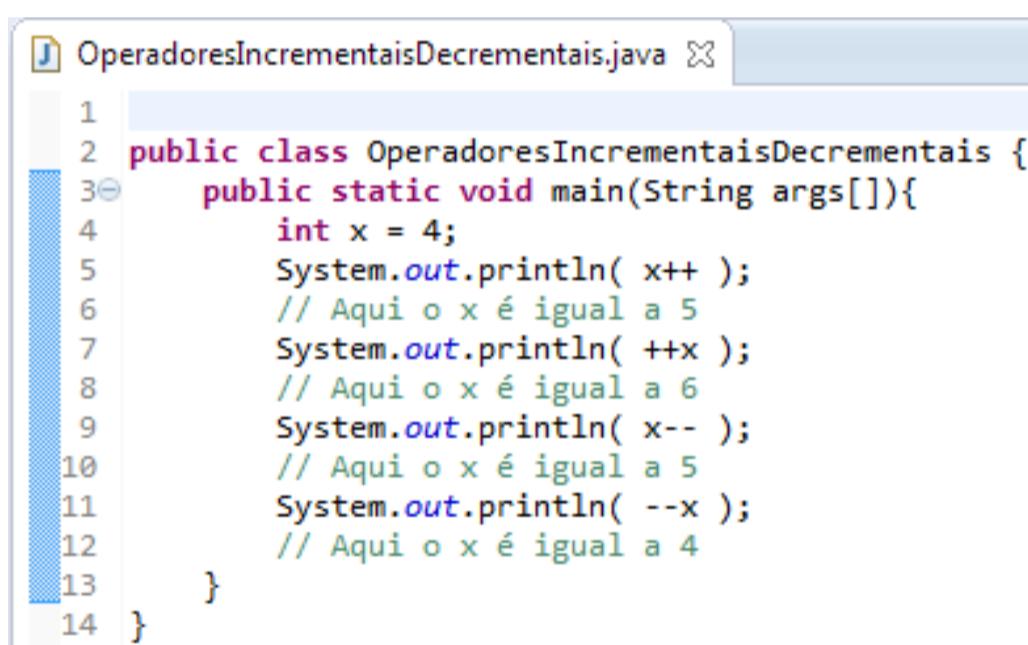
Os operadores incrementais e decrementais têm a função de aumentar ou diminuir o valor de uma variável em 1 (um). Eles podem ser pré-/pós-incrementais ou pré-/pós-decrementais. Veja:

- **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 (--)**

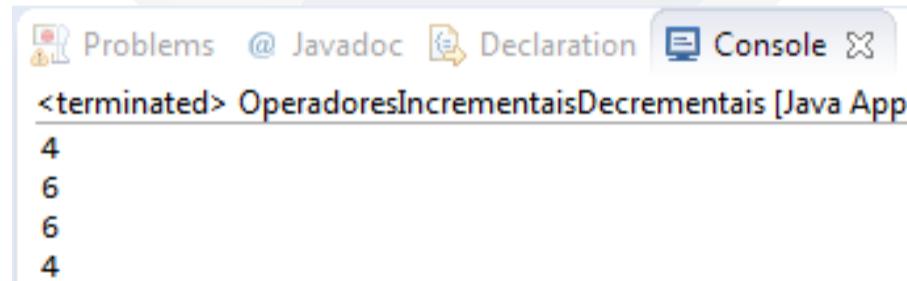
- **Pré-decremental ou prefixo:** Se o sinal for colocado antes da variável, seu valor será diminuído em um antes da expressão ser resolvida;
- **Pós-decremental ou sufixo:** Se o sinal for colocado após a variável, a expressão (adição, subtração, multiplicação etc.) será resolvida primeiro e, em seguida,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

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



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

O resultado é o seguinte:



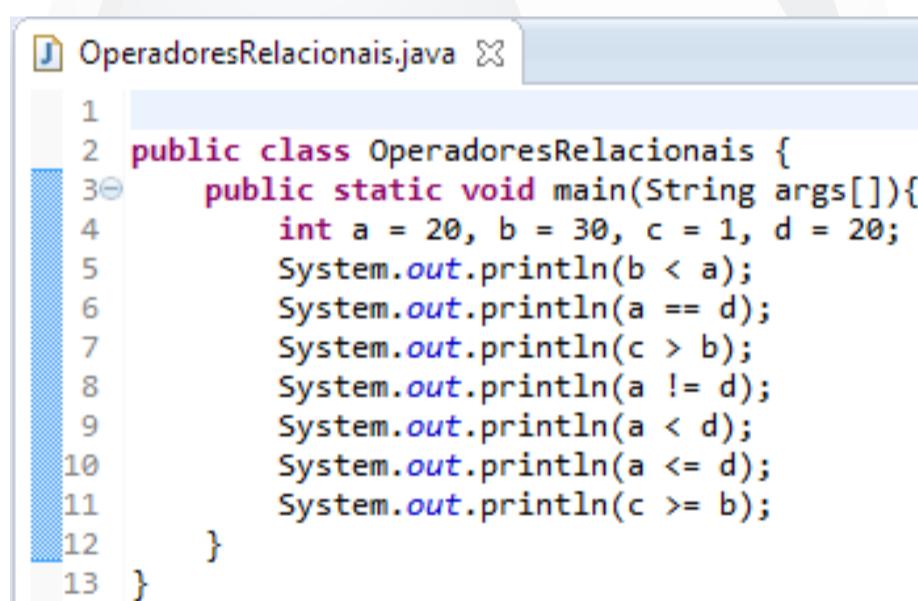
```
Problems @ Javadoc Declaration Console
<terminated> OperadoresIncrementaisDecrementais [Java App]
4
6
6
4
```

4.4. Operadores relacionais

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

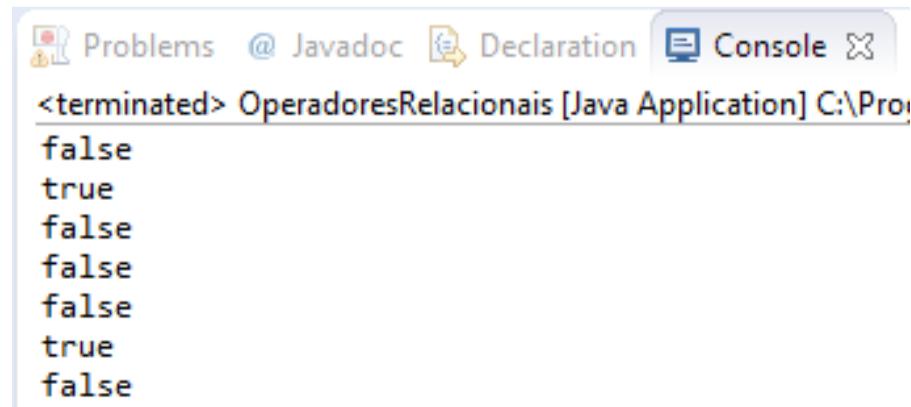
Operador relacional	Descrição
<code>==</code>	Igual a
<code>!=</code>	Diferente de
<code>></code>	Maior que
<code><</code>	Menor que
<code>>=</code>	Maior ou igual a
<code><=</code>	Menor ou igual a

Exemplo:



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

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



```
<terminated> OperadoresRelacionais [Java Application] C:\Pro
false
true
false
false
false
true
false
```

4.5. 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
&&	AND (conjunção)
 	OR (disjunção)
!	NOT (negação)

Em um teste lógico utilizando o operador **&& (AND)**, o resultado somente será verdadeiro (**true**) se todas as expressões lógicas forem avaliadas como verdadeiras. Porém, se o operador utilizado for **|| (OR)**, basta que uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro. 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 **&& (AND)** – Utiliza dois operandos:

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

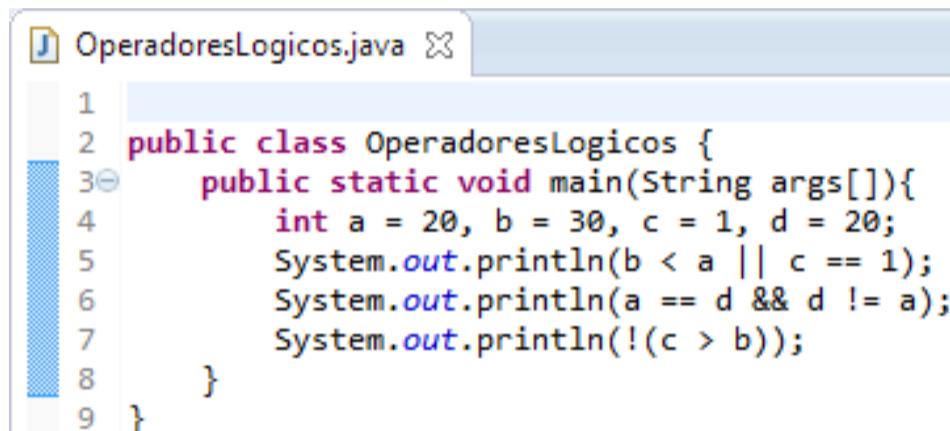
- Operador **|| (OR)** – Utiliza dois operandos:

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

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

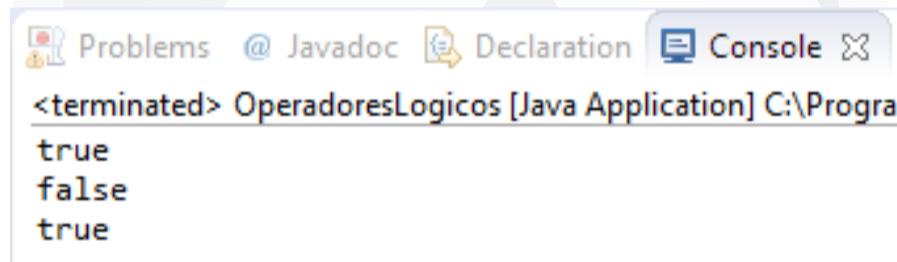
Operando	Resultado
true	false
false	true

A seguir temos um exemplo de como utilizar operadores lógicos:



```
1
2 public class OperadoresLogicos {
3     public static void main(String args[]){
4         int a = 20, b = 30, c = 1, d = 20;
5         System.out.println(b < a || c == 1);
6         System.out.println(a == d && d != a);
7         System.out.println(!(c > b));
8     }
9 }
```

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



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

4.6. Operador ternário

O operador ternário, ou operador condicional, é composto por três operandos separados pelos sinais ? e : 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>;
```

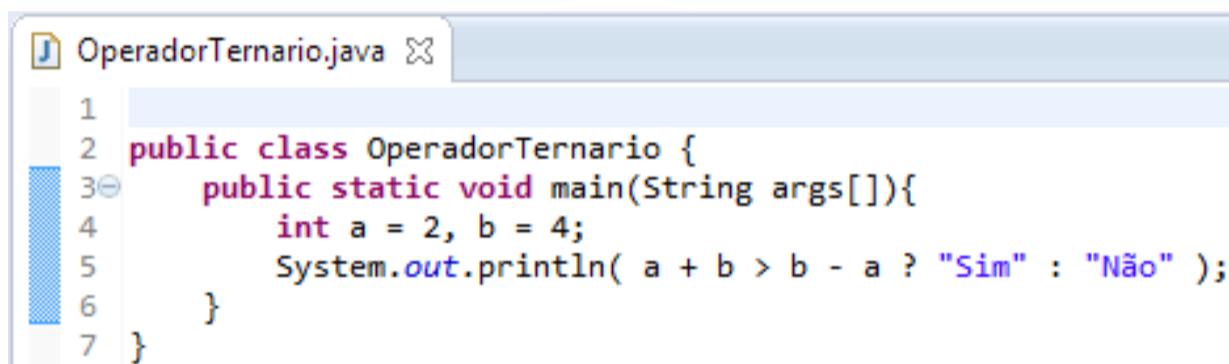
Esse operador simplifica uma operação de desvio condicional (assunto que será tratado posteriormente), que possui a seguinte estrutura:

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

Em que:

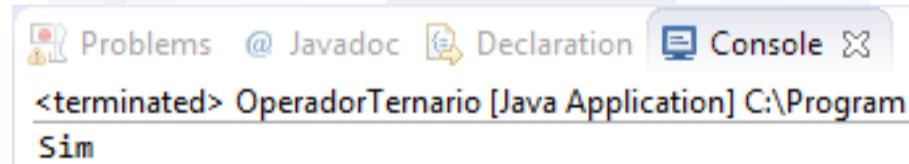
- **teste lógico** é qualquer valor ou expressão que pode ser avaliado como verdadeiro ou falso;
- **valor se verdadeiro** é o valor atribuído se o teste lógico for avaliado como verdadeiro;
- **valor se falso** é o valor atribuído se o teste lógico for avaliado como falso.

Veja um exemplo:



```
J OperadorTernario.java X
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 da execução do código será o seguinte:



```
Problems @ Javadoc Declaration Console X
<terminated> OperadorTernario [Java Application] C:\Program
Sim
```

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

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

```
Problems @ Javadoc Declaration Console
<terminated> OrdemPrioridade [Java Application] C:\Program
7
15
```



Teste seus conhecimentos Operadores

4

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. Quais são os operadores aritméticos disponíveis em Java?

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

3. Quais são 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.

Controle de fluxo

5

- ✓ Estruturas condicionais;
- ✓ Estruturas de repetição;
- ✓ Break;
- ✓ Instruções rotuladas;
- ✓ Continue.

5.1. Introdução

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

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**.

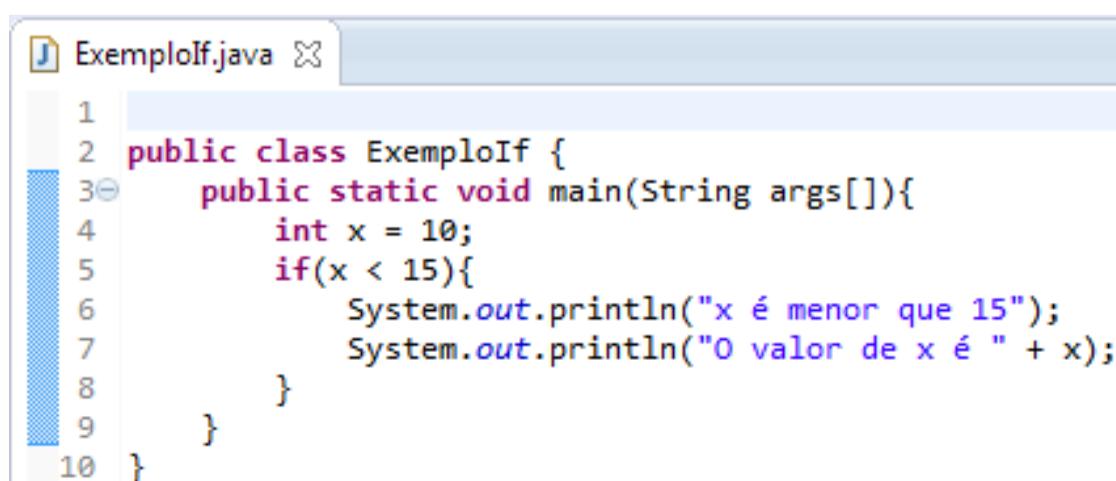
Veremos essas estruturas adiante.

5.1.1. 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:

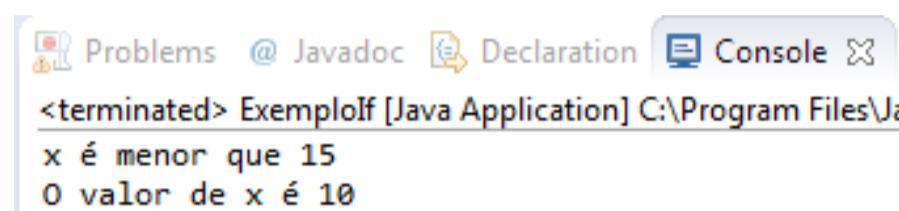
```
if (<teste condicional>){  
    Comandos;  
}
```

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:



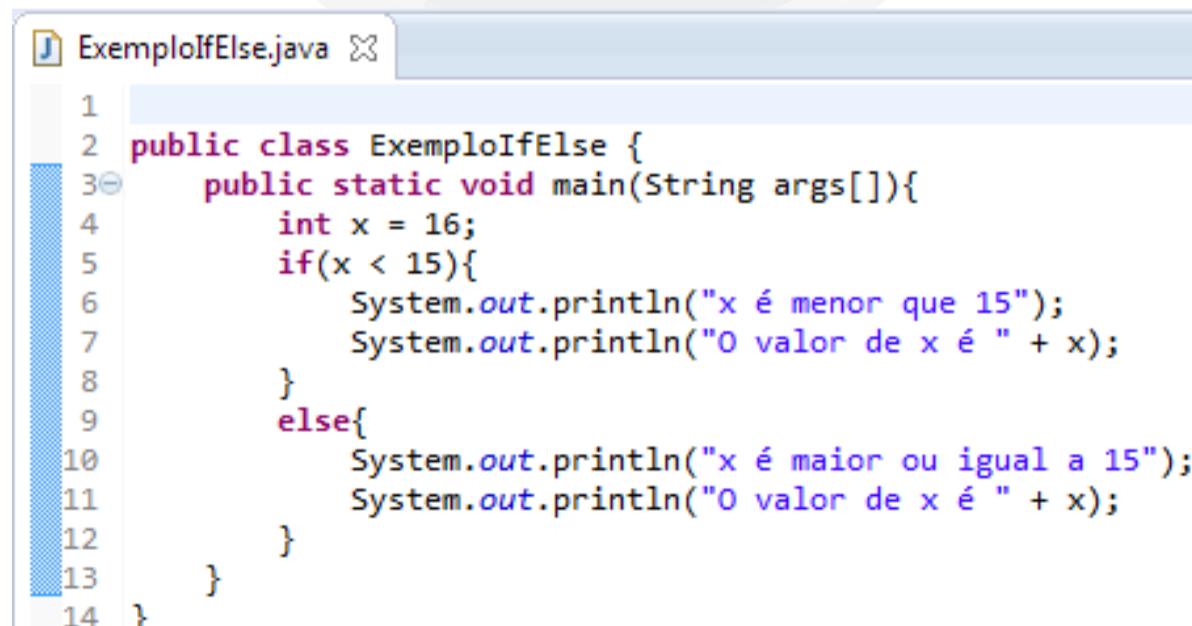
```
<terminated> ExemploIf [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\java.exe -jar ExemploIf.jar
x é menor que 15
O valor de x é 10
```

! 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 há uma 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:

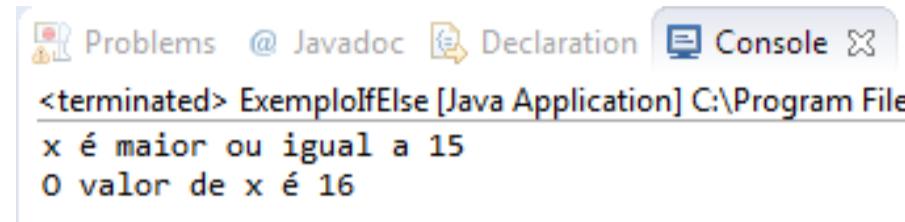
```
if (<teste condicional>) {
    comandos;
} else {
    comandos;
}
```

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



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

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

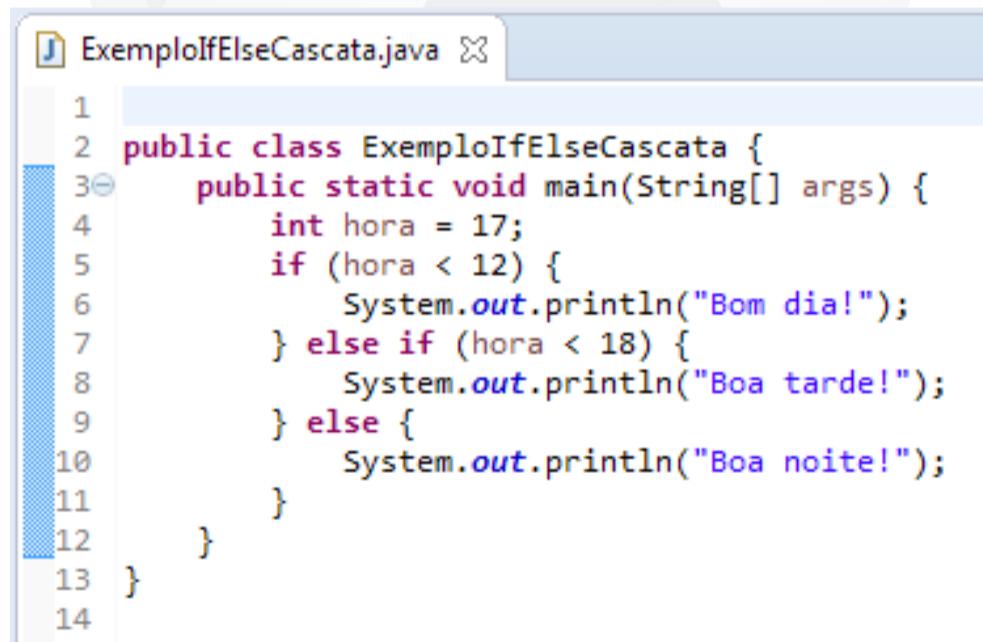


```
Problems @ Javadoc Declaration Console ×
<terminated> ExemploIfElse [Java Application] C:\Program File
x é maior ou igual a 15
O valor de x é 16
```

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:



```
ExemploIfElseCascata.java ×
1
2 public class ExemploIfElseCascata {
3     public static void main(String[] args) {
4         int hora = 17;
5         if (hora < 12) {
6             System.out.println("Bom dia!");
7         } else if (hora < 18) {
8             System.out.println("Boa tarde!");
9         } else {
10            System.out.println("Boa noite!");
11        }
12    }
13}
14
```

5.1.2. switch

Em algumas situações, pode haver uma sequência muito longa de desvios condicionais, o que dificulta a interpretação do programa. Para esse 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** 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 (**<=**).

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, executa 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 continua até que algum **break** esteja à frente ou se 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** ou **char**.

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

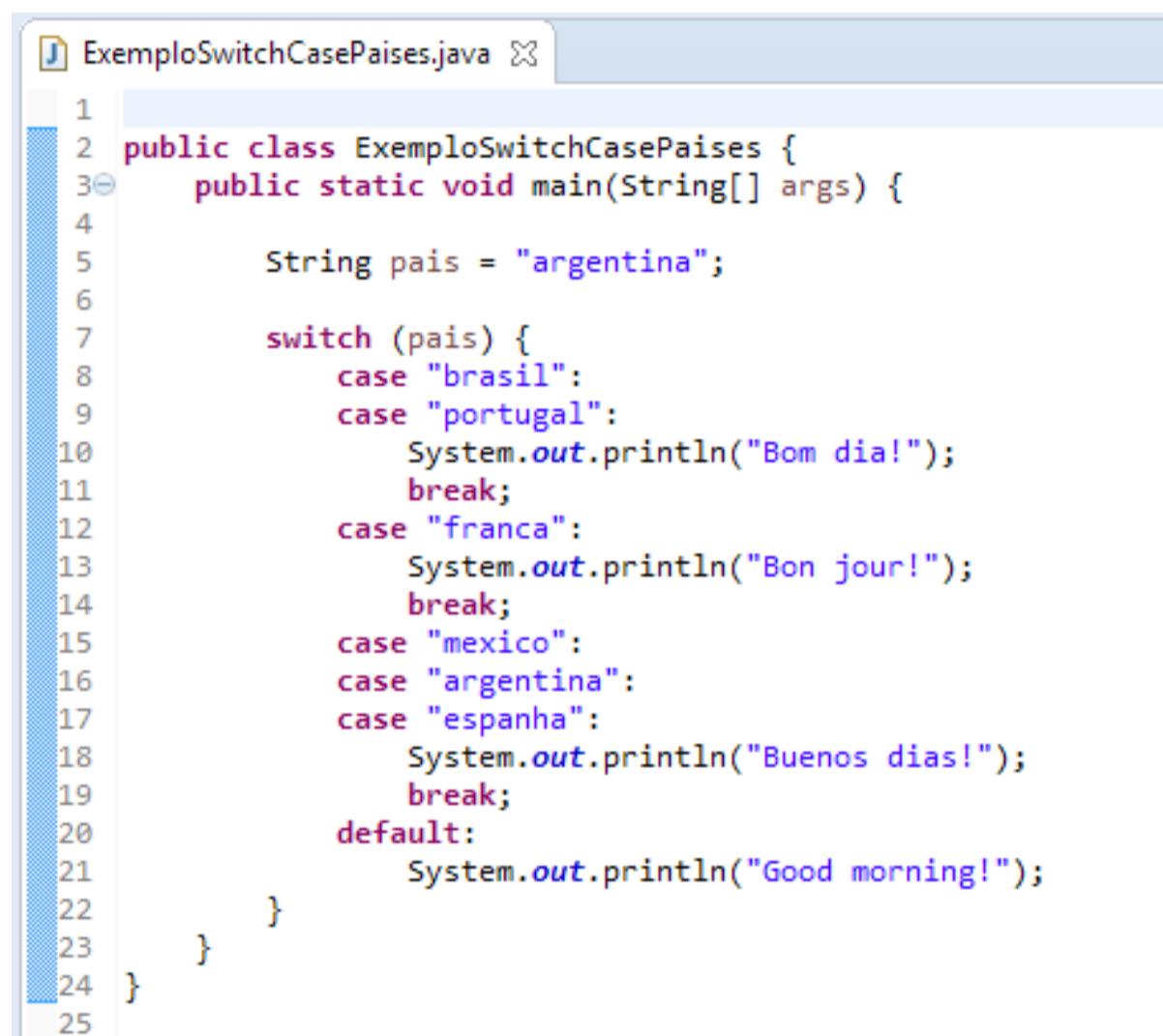
```
1  public class ExemploSwitchCase {
2      public static void main(String args[]){
3          int mes = 7;
4          switch(mes){
5              case 1: System.out.println("Janeiro"); break;
6              case 2: System.out.println("Fevereiro"); break;
7              case 3: System.out.println("Março");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

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



```
Problems @ Javadoc Declaration Console Task List
<terminated> ExemploSwitchCasePaises [Java Application] C:\Java\jdk\jdk1.8
Buenos dias!
```

5.2. 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 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.

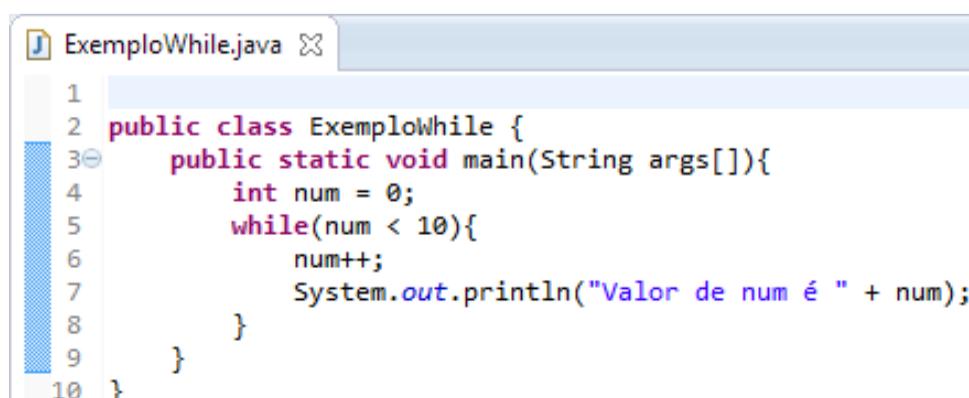
5.2.1. While

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:

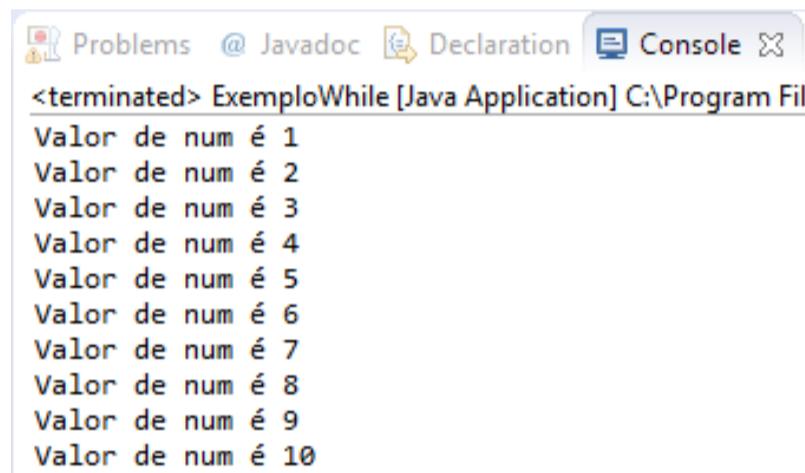


```
1  
2 public class ExemploWhile {  
3     public static void main(String args[]){  
4         int num = 0;  
5         while(num < 10){  
6             num++;  
7             System.out.println("Valor de num é " + num);  
8         }  
9     }  
10 }
```

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
```

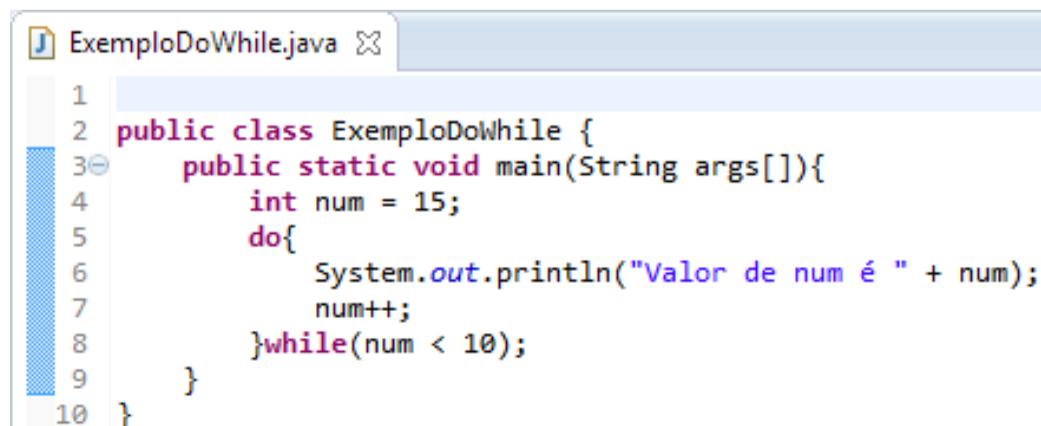
5.2.2. Do/while

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**:

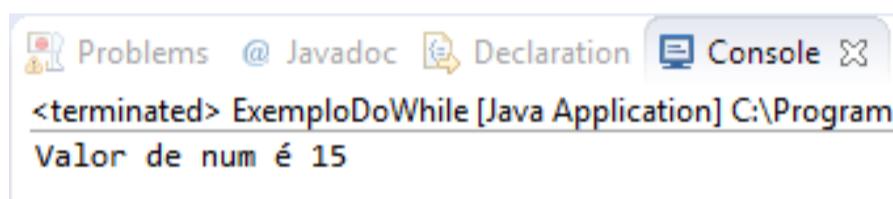


```
ExemploDoWhile.java X
1
2 public class ExemploDoWhile {
3     public static void main(String args[]){
4         int num = 15;
5         do{
6             System.out.println("Valor de num é " + num);
7             num++;
8         }while(num > 10);
9     }
10 }
```

Em que:

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

O resultado é o seguinte:



```
<terminated> ExemploDoWhile [Java Application] C:\Program  
Valor de num é 15
```

5.2.3. 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:

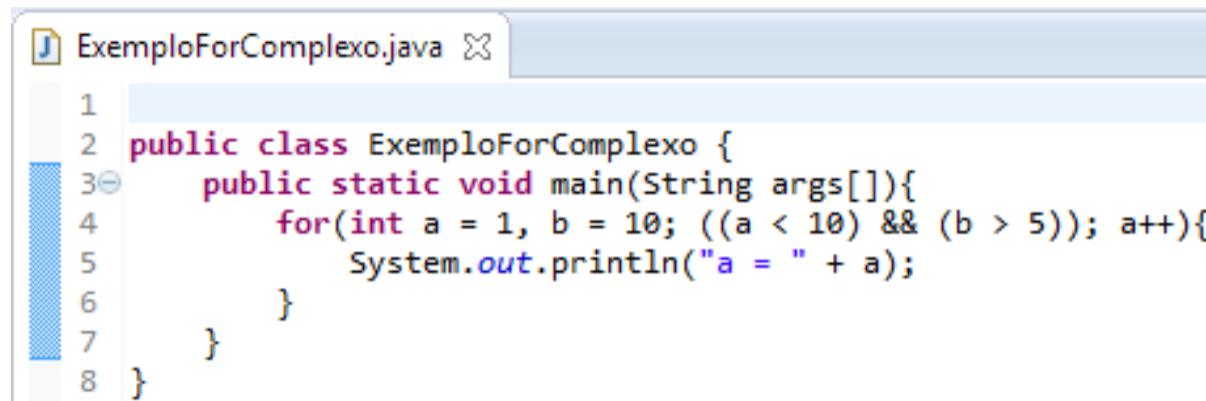
```
for (int a = 1, b = 1; ... { }
```

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**

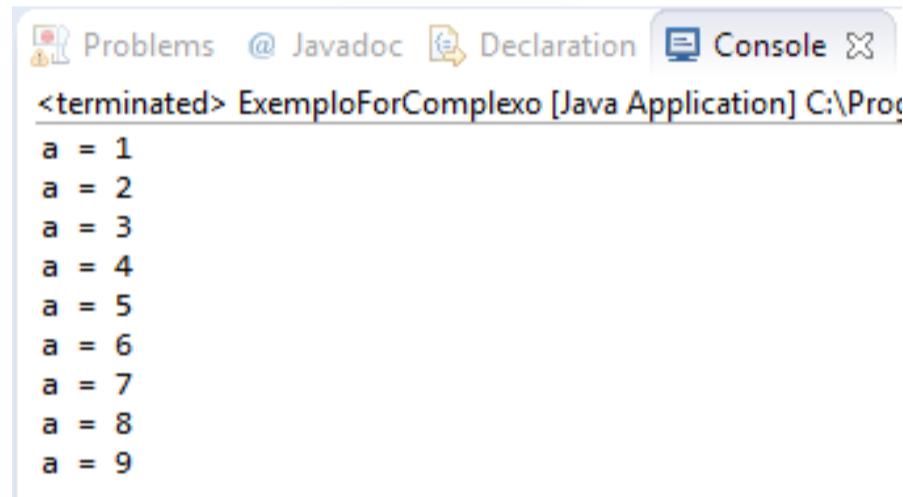
O elemento seguinte é a expressão condicional. Ela se refere a um teste que será executado e deverá retornar um valor booleano. Só podem ser especificadas expressões lógicas, que podem ser complexas. Por isso, é importante ter cuidado com códigos que utilizam diversas expressões lógicas. Equivale ao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

Veja um exemplo:



```
ExemploForComplexo.java
1
2 public class ExemploForComplexo {
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:



```
Problems @ Javadoc Declaration Console
<terminated> ExemploForComplexo [Java Application] C:\Pro
a = 1
a = 2
a = 3
a = 4
a = 5
a = 6
a = 7
a = 8
a = 9
```

Nesse 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 nesse 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**:

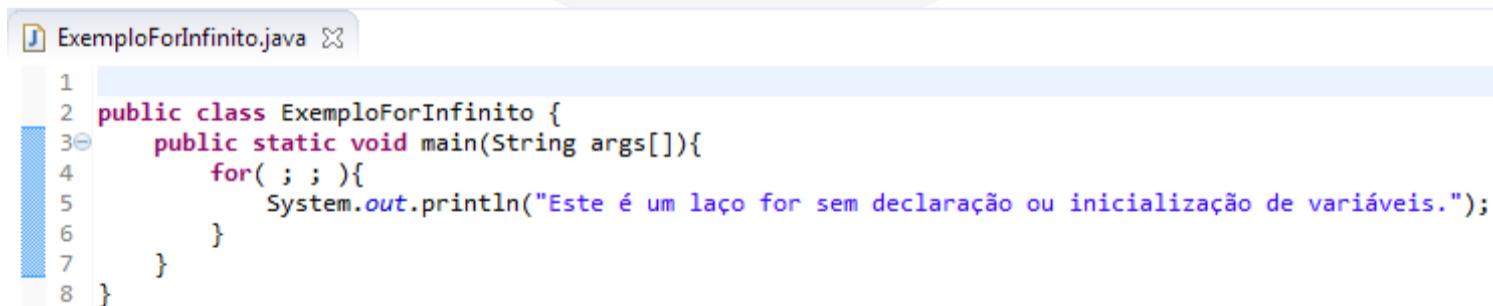
```
for(int a = 1; a < 10; a++){ }
```

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**.

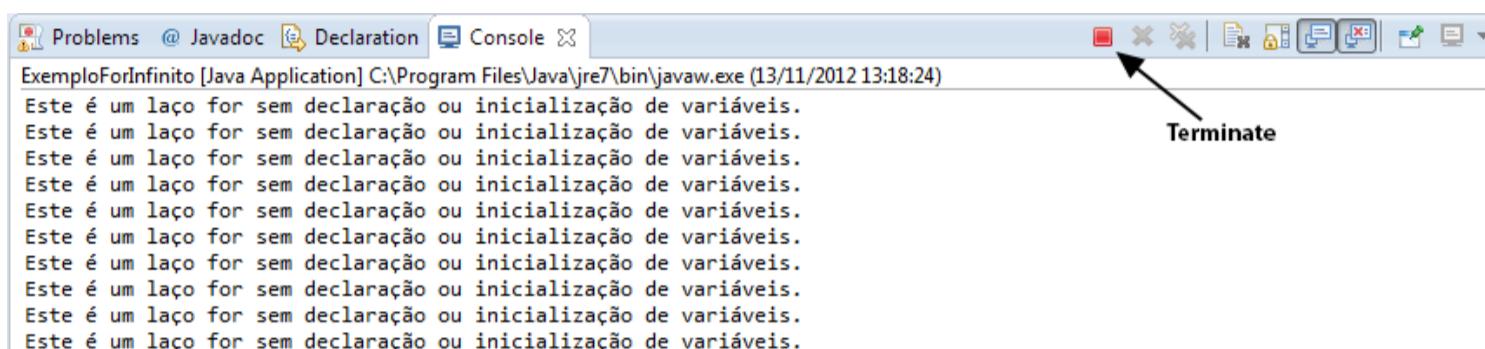
É possível, mas não recomendável, usar um **for** que não possua todas as três partes do laço descritas. Nenhuma das três partes é obrigatória. Nesse caso, o laço poderá ser infinito e, não havendo seções de declaração e inicialização, ele agirá como um laço de repetição **while**.

Veja um exemplo:



```
ExemploForInfinito.java  
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.

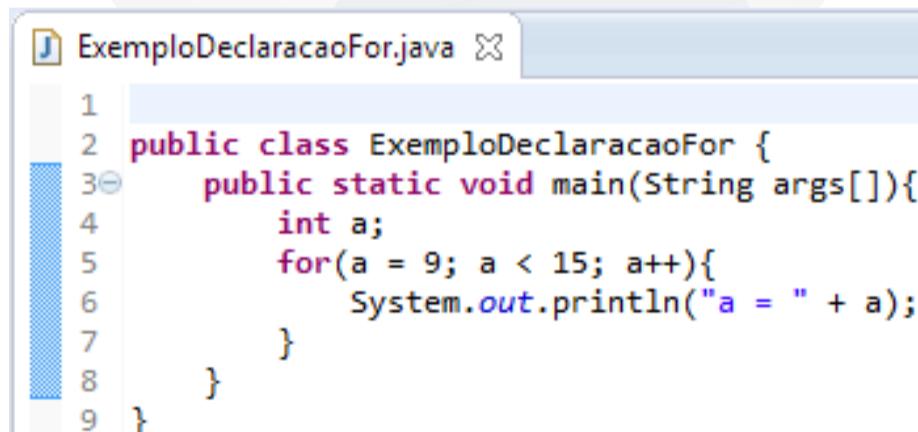


A screenshot of a Java application window titled "ExemploForInfinito [Java Application]". The window has tabs for "Problems", "@ Javadoc", "Declaration", and "Console". The "Console" tab is active, showing the output of a Java application. The output consists of ten identical lines of text: "Este é um laço for sem declaração ou inicialização de variáveis.". An arrow points from the text "Terminate" to a red square button with a white 'X' icon on the right side of the toolbar, indicating it is the button used to stop the infinite loop.

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**:



A screenshot of a Java code editor showing a file named "ExemploDeclaracaoFor.java". The code contains a single class definition: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:

```
<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:

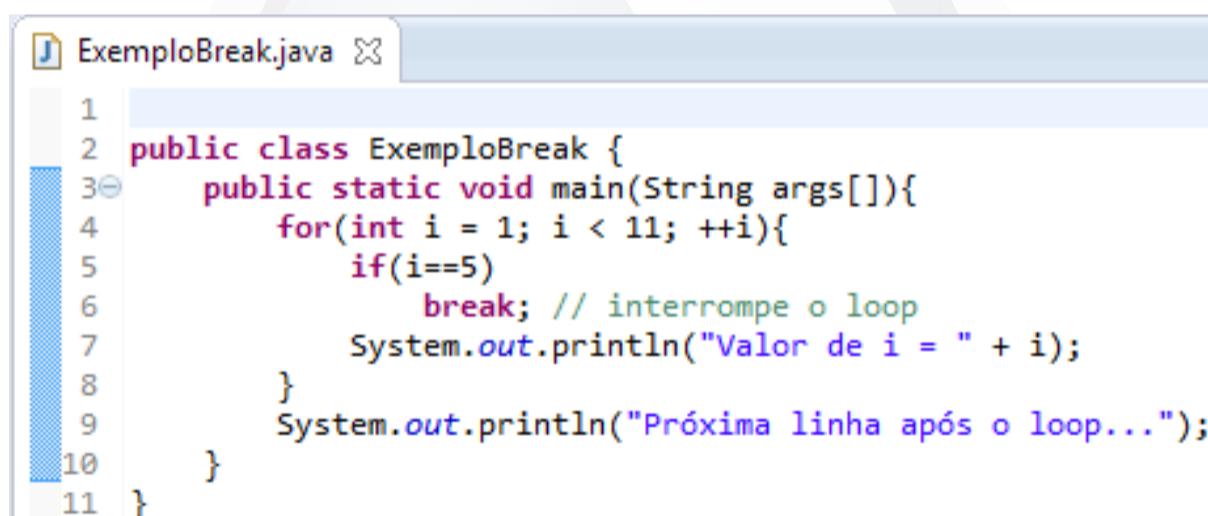
5.3. Outros comandos

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**.

5.3.1. 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:



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

O resultado é o seguinte:



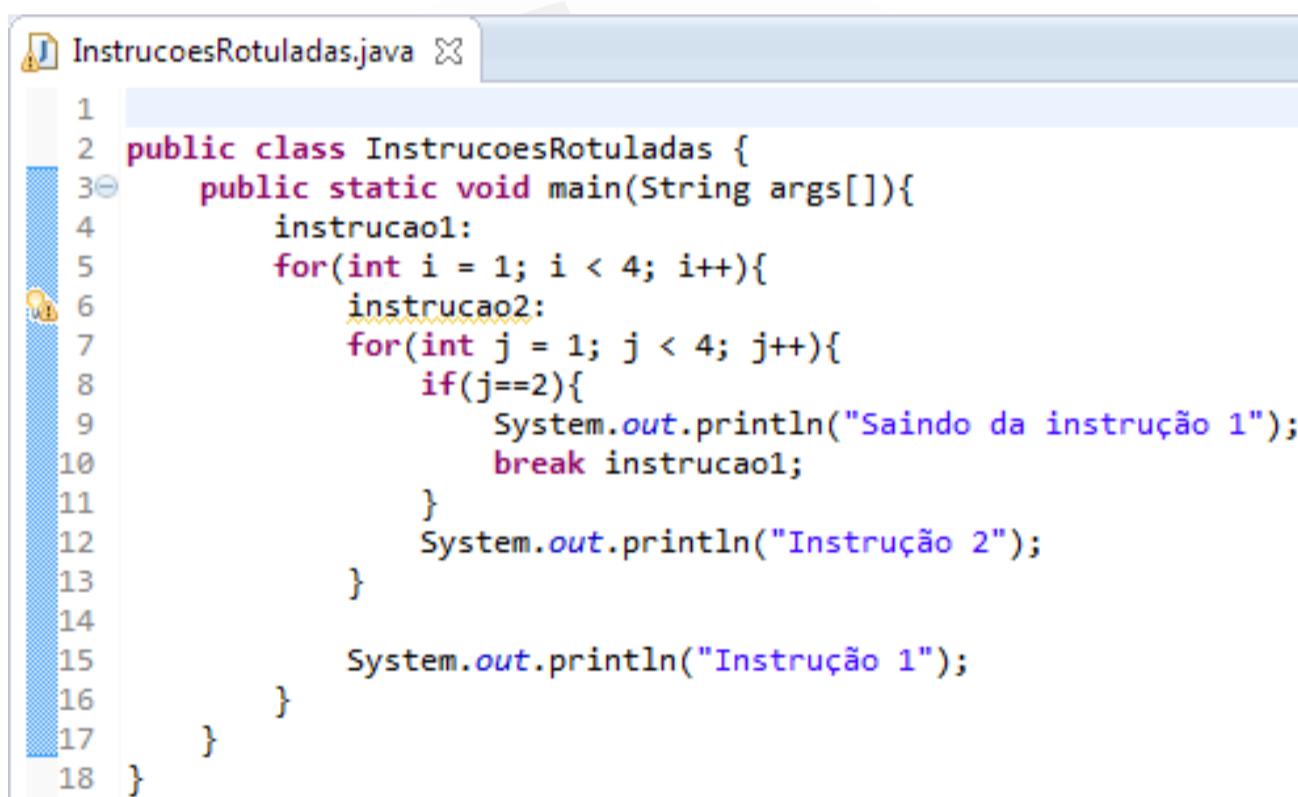
```
Problems @ Javadoc Declaration Console
<terminated> ExemploBreak [Java Application] C:\Program Fil
Valor de i = 1
Valor de i = 2
Valor de i = 3
Valor de i = 4
Próxima linha após o loop...
```

5.3.1.1.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.

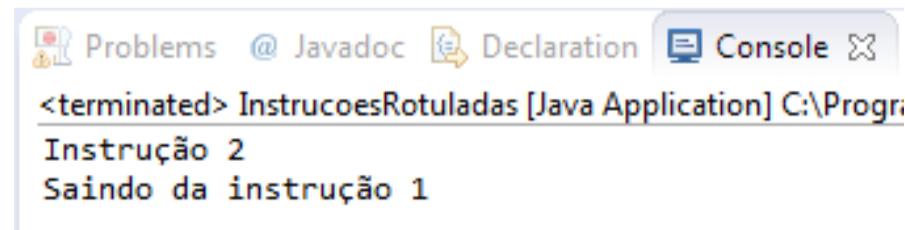
O rótulo atribuído à instrução deve seguir as regras de nomes de variável válidos para a linguagem Java. Além disso, devemos verificar se ele segue as convenções de nomeação para Java.

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:



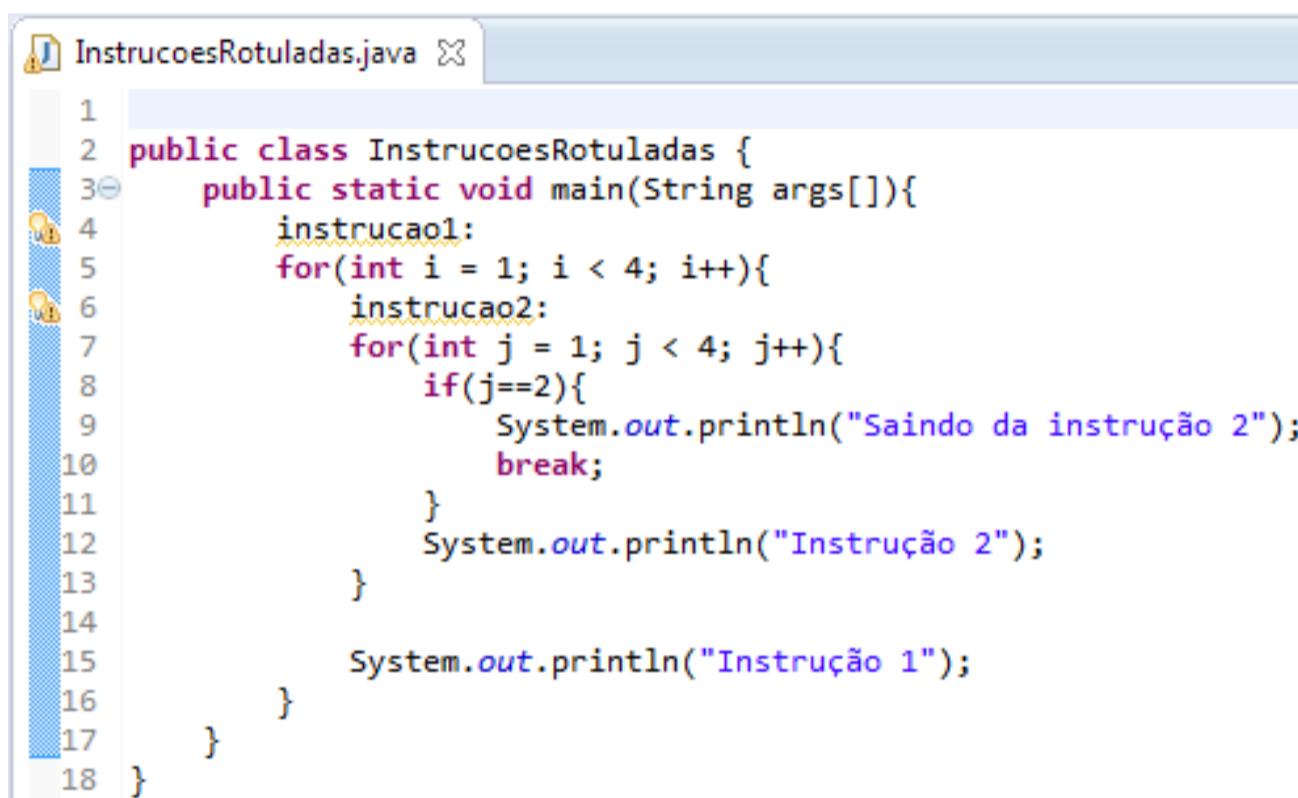
```
1  public class InstrucoesRotuladas {
2      public static void main(String args[]){
3          instrucao1:
4              for(int i = 1; i < 4; i++){
5                  instrucao2:
6                      for(int j = 1; j < 4; j++){
7                          if(j==2){
8                         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

O resultado será o seguinte:



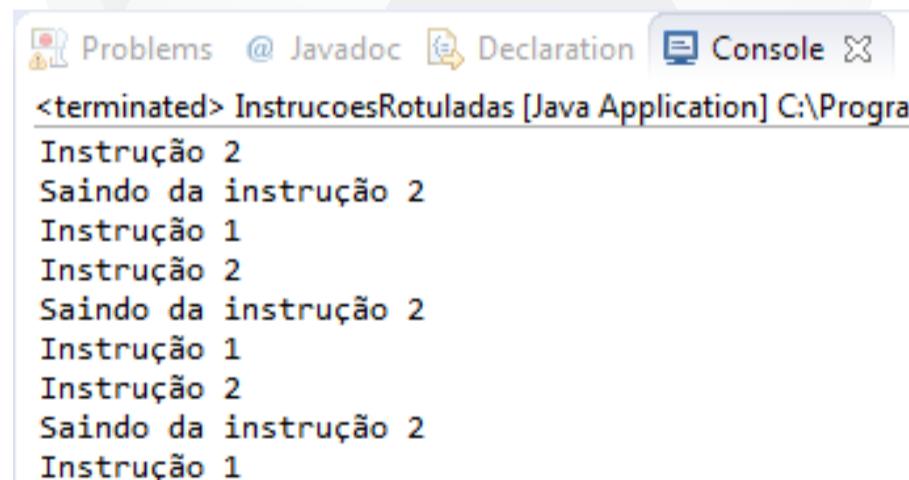
```
Problems @ Javadoc Declaration Console
<terminated> InstrucoesRotuladas [Java Application] C:\Progr
Instrução 2
Saindo da instrução 1
```

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:



```
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
15                     System.out.println("Instrução 1");
16                 }
17             }
18 }
```

O resultado será o seguinte:

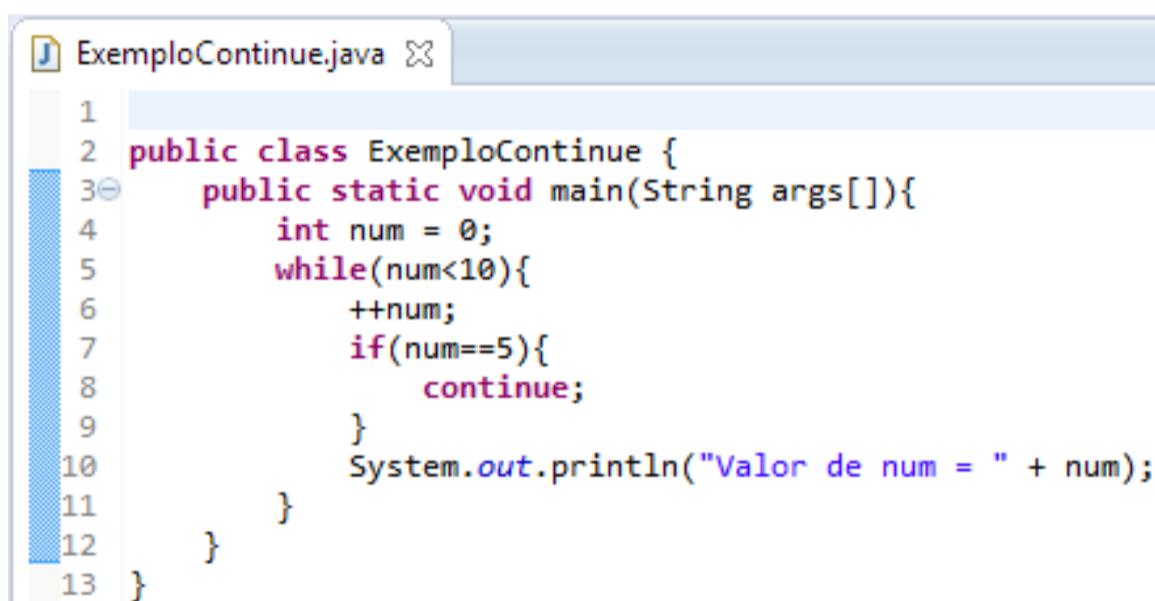


```
Problems @ Javadoc Declaration Console
<terminated> InstrucoesRotuladas [Java Application] C:\Progra
Instrução 2
Saindo da instrução 2
Instrução 1
Instrução 2
Saindo da instrução 2
Instrução 1
Instrução 2
Saindo da instrução 2
Instrução 1
```

5.3.2. Continue

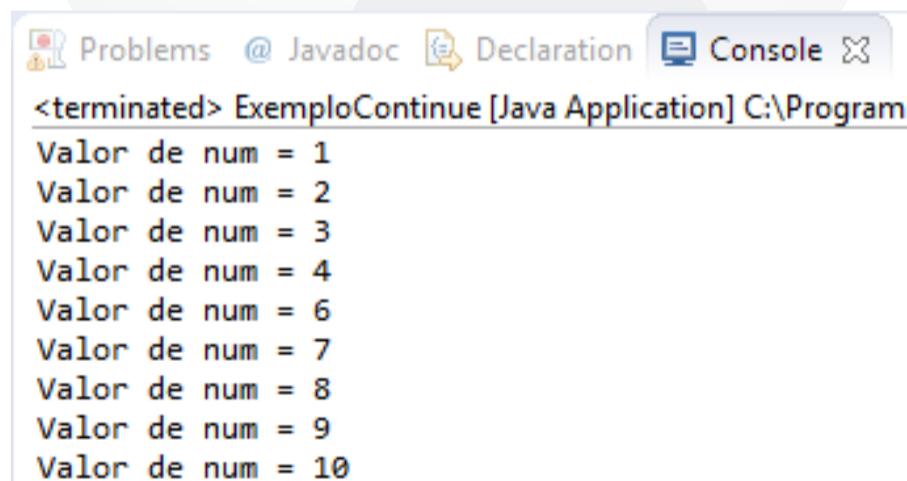
O comando **continue** pode ser utilizado somente em laços de repetição. Quando ele é executado, o laço volta imediatamente para o teste de condição. Diferentemente do comando **break**, o **continue** causa a saída da iteração atual do laço, dirigindo-se imediatamente à iteração seguinte.

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



```
ExemploContinue.java
1
2 public class ExemploContinue {
3     public static void main(String args[]){
4         int num = 0;
5         while(num<10){
6             ++num;
7             if(num==5){
8                 continue;
9             }
10            System.out.println("Valor de num = " + num);
11        }
12    }
13 }
```

O resultado é o seguinte:



```
Problems @ Javadoc Declaration Console
<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.



Teste seus conhecimentos Controle de fluxo

5

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.

Métodos

6

- ✓ Estrutura de um método;
- ✓ Chamando um método;
- ✓ Passagem de parâmetros.

6.1. Introdução

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. Os métodos em Java atuam como funções ou procedimentos, elementos estes encontrados em outras linguagens e tecnologias, porém, sempre são alocados em classes e fazem parte do contexto que formará o comportamento do objeto.

Para ilustrar como funcionam os métodos, imagine uma classe **Carro** que tem os seguintes métodos atribuídos aos seus objetos: **andar**, **abastecer** e **parar**. É possível implementar o método **abastecer**, por exemplo, de várias formas nos objetos da classe **Carro**, de acordo com o tipo de abastecimento que o carro pode ter (álcool, gasolina, diesel, gás ou mesmo eletricidade). O método terá sempre a mesma função, que é abastecer, mas cada tipo de abastecimento será implementado de uma maneira diferente.

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.

6.2.1. 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.

6.2.2. Um método na prática

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:

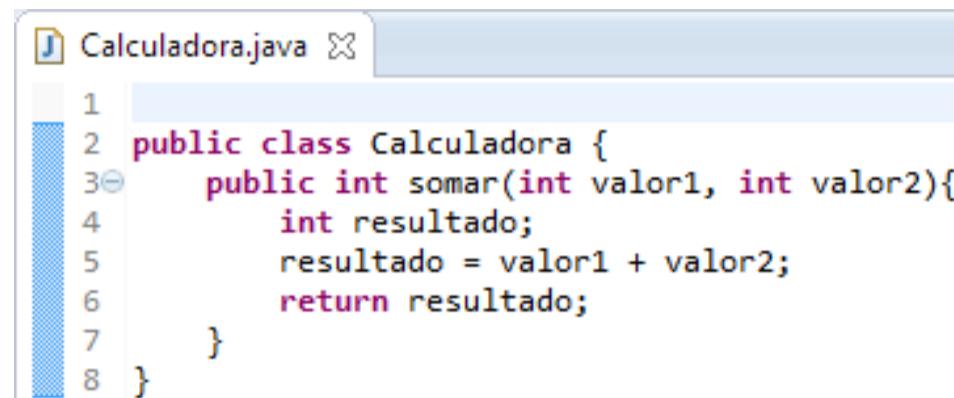
- **public** é um modificador de acesso opcional, declarado para esse método com o fim de torná-lo acessível a partir de qualquer local e por qualquer objeto do programa que também possua acesso à classe onde ele está inserido;
- **int** é o tipo de retorno do método, ou seja, define o tipo de dado **int** (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.

6.3. Chamando um 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.

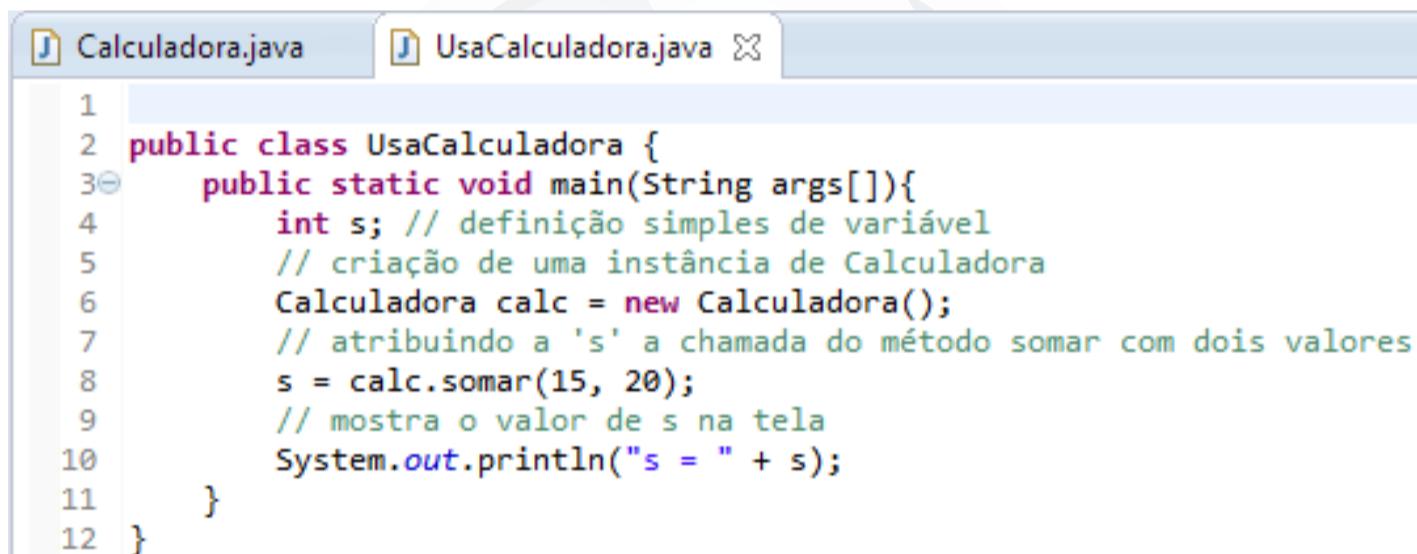
Veja um exemplo de como chamar um método:

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



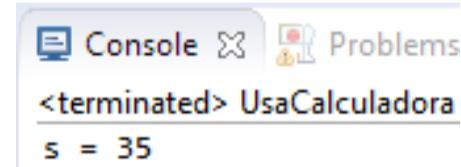
```
1
2 public class Calculadora {
3     public int somar(int valor1, int valor2){
4         int resultado;
5         resultado = valor1 + valor2;
6         return resultado;
7     }
8 }
```

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



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

O resultado será o seguinte:



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

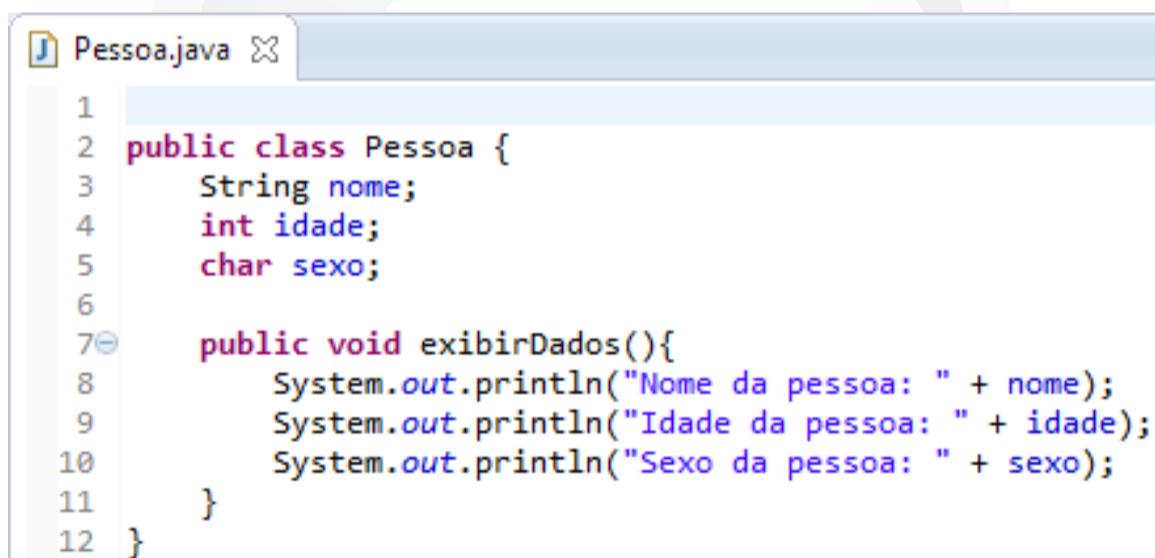
6.4. Passagem de parâmetros

Na linguagem Java, a passagem de argumentos, independente de se tratar 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. Isso 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 esse 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:



The screenshot shows a Java code editor window with the file 'Pessoa.java' open. The code defines a class 'Pessoa' with three fields: 'nome', 'idade', and 'sexo'. It also contains a method 'exibirDados()' that prints the values of these fields to the console using System.out.println().

```
1  public class Pessoa {
2      String nome;
3      int idade;
4      char sexo;
5
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 }
```

```
1  public class PassagemPorValor {
2      public static void main(String args[]){
3          int valor = 1500;
4          // tentativa de alterar o valor
5          System.out.println("Antes da chamada do método alterarValor o valor é " + valor);
6          alterarValor(valor);
7          System.out.println("Depois da chamada do método alterarValor o valor é " + valor);
8          System.out.println();
9
10         Pessoa maria = new Pessoa();
11         maria.sex = 'f';
12         maria.nome = "Maria";
13         maria.idade = 50;
14
15         // tentativa de alterar a referencia do objeto
16         System.out.println("Valores antes da chamada do método alterarReferenciaObjeto:");
17         maria.exibirDados();
18         alterarReferenciaObjeto(maria);
19         System.out.println("Valores depois da chamada do método alterarReferenciaObjeto:");
20         maria.exibirDados();
21         System.out.println();
22
23         // alterando o CONTEUDO do objeto
24         System.out.println("Valores antes da chamada do método alterarConteudoObjeto:");
25         maria.exibirDados();
26         alterarConteudoObjeto(maria);
27         System.out.println("Valores depois da chamada do método alterarConteudoObjeto:");
28         maria.exibirDados();
29     }
30
31     static void alterarValor(int valor){
32         //alterando o valor
33         valor = 137;
34     }
35
36     static void alterarReferenciaObjeto(Pessoa p){
37         Pessoa ana = new Pessoa();
38         ana.sex = 'f';
39         ana.nome = "Ana";
40         ana.idade = 18;
41         p = ana; // alterando o valor de p
42     }
43
44     static void alterarConteudoObjeto(Pessoa p){
45         // alterando o conteudo do objeto referenciado por p
46         p.sex = 'm';
47         p.nome = "João";
48         p.idade = 33;
49     }
50 }
51 }
```

A imagem a seguir ilustra a saída:

The screenshot shows a Java application running in an IDE's console window. The title bar indicates it's a Java Application named 'PassagemPorValor'. The console output shows three distinct sections of code execution:

- Section 1:** Prints the value of 'valor' before and after calling the method 'alterarValor'. Both outputs show the value as 1500.
- Section 2:** Prints the values of 'p.nome', 'p.idade', and 'p.sexo' both before and after calling 'alterarReferenciaObjeto'. The values remain the same (Maria, 50, f).
- Section 3:** Prints the values of 'p.nome', 'p.idade', and 'p.sexo' both before and after calling 'alterarConteudoObjeto'. The values change (João, 33, m), demonstrating that the method has modified the original object reference.

```
<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.



Teste seus conhecimentos Métodos

6

1. 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.

2. 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) 03
- d) 12
- e) 45

3. Qual das seguintes afirmações a respeito do comando Return está incorreta?

- a) Se um método retornar um valor, este será devolvido para a rotina que fez a chamada do método pelo comando return.
- b) O comando return basicamente encerra o método e retorna a execução do programa para o chamador do método.
- c) O comando return não pode ser utilizado sem nenhum retorno declarado no método.
- d) return resultado; indica que qualquer chamada ao método somar receberá o resultado da variável resultado como valor de retorno.
- e) O comando return pode ser utilizado mesmo sem nenhum retorno declarado no método.

4. Qual das afirmativas a seguir sobre passagem de parâmetro está incorreta?

- a) Na linguagem Java, a passagem de argumentos é feita apenas por valor, isto é, o valor do argumento não pode ser alterado dentro do método.
- b) Ao passar um objeto como argumento de um método, na realidade você está passando o valor da referência do objeto, e não o objeto em si.
- c) 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.
- d) Na linguagem Java, a passagem de argumentos não é feita por valor, ou seja, o valor do argumento pode ser alterado dentro do método.
- e) Parâmetro refere-se a uma variável definida por um método. Quando este método é chamado, a variável recebe um valor.

5. Em que situação ocorre uma sobrecarga de métodos?

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

Classes e orientação a objetos 7

- ✓ Objetos;
- ✓ Classes;
- ✓ Métodos.

7.1. Introdução

Nas leituras 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. Nesta leitura complementar, você conhecerá esses e outros conceitos fundamentais da orientação a objetos, indispensáveis para uma boa compreensão da estrutura e do funcionamento da linguagem Java.

7.2. Objetos

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 (POO) 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 e o conjunto de métodos do objeto.

Nas linguagens orientadas a objetos, é possível classificar e determinar os tipos de objetos que serão usados em uma aplicação. Isso é feito a partir do conceito de classe, que veremos mais à frente.

7.2.1. Atributo

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 relativos 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.

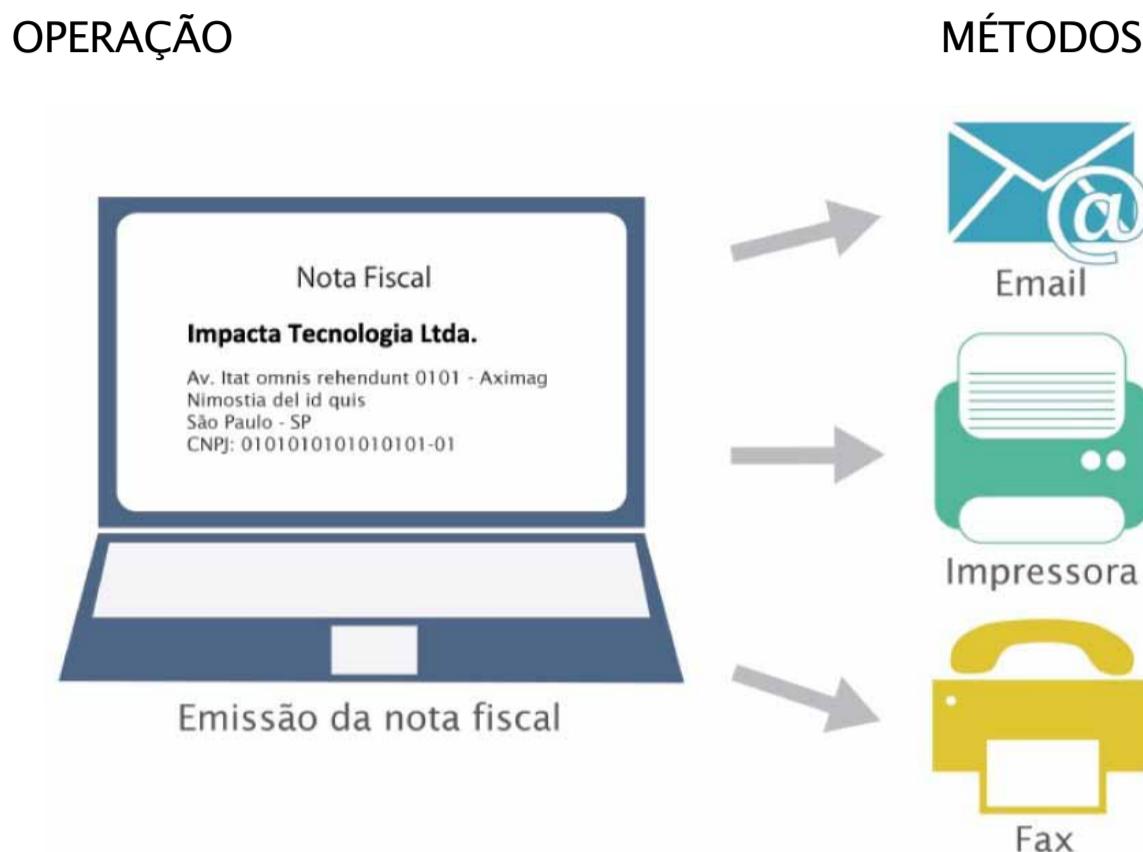
É importante destacar que cada objeto é responsável pelo controle da alteração de seus próprios atributos e não deve interferir nos atributos de outro objeto, salvo em caso de solicitação de serviços. Ou seja, um objeto deve ser projetado para ser independente em relação aos outros.

7.2.2. Método

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

Dentro do paradigma da programação orientada a 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.



7.2.3. Mensagem

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 a fim de invocar um de seus métodos. A mensagem já inclui os argumentos necessários para a execução da tarefa solicitada.

Veja um exemplo:

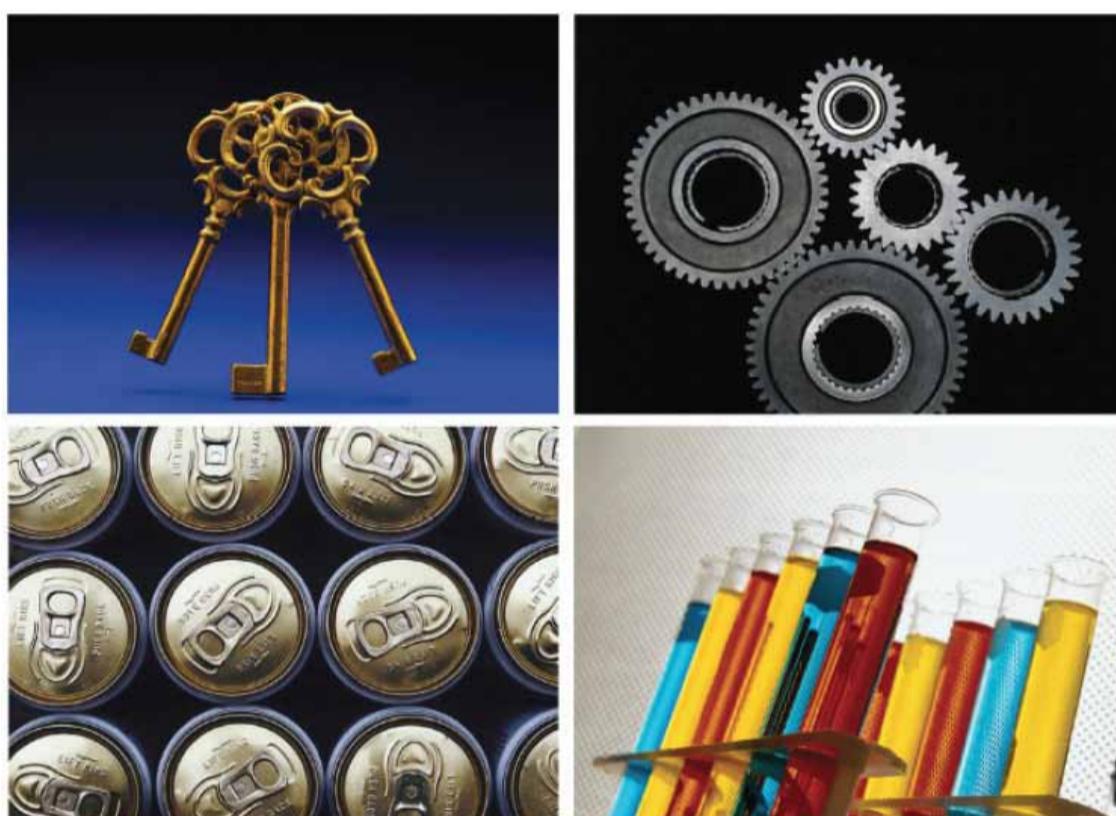
```
pessoa.falar("Bom dia");
```

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”).

7.2.4. Classe

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

Dentro do paradigma da programação orientada a objetos, 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. Veja, a seguir, o conceito de instanciação.

- **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 equivale a criar uma cópia de uma classe na memória, para ser usada no programa.

- **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 os métodos podem ser acessados por 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.

- **Herança**

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.

- **Polimorfismo**

O polimorfismo é um mecanismo que confere a um mesmo objeto a capacidade de poder se comportar de diferentes maneiras, dependendo do contexto em que ele esteja sendo empregado.

Assim, se um objeto A for criado a partir de uma classe X, ele se comportará de uma determinada maneira. Se esse mesmo objeto A for criado a partir de uma classe Y, ele se comportará de uma maneira diferente. E quando for necessário, o mesmo objeto A poderá converter-se em X ou Y, conforme a necessidade do desenvolvedor, para atender a uma demanda específica.

7.3. Classes

No tópico anterior, definimos brevemente o conceito de classes. No decorrer deste tópico, abordaremos o assunto de forma mais ampla. Mas, antes de começar com as classes, você deve conhecer o conceito de pacotes e saber como criá-los.

7.3.1. 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.

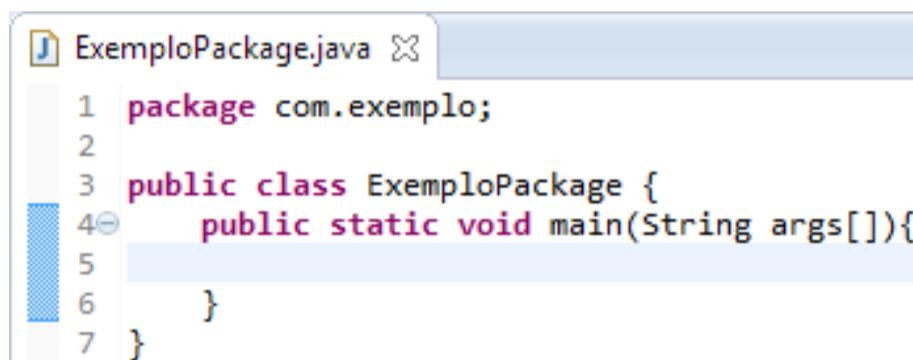
Na tabela a seguir, você encontra alguns dos principais pacotes de Java:

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;
2
3 public class ExemploPackage {
4     public static void main(String args[]){
5
6     }
7 }
```

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.

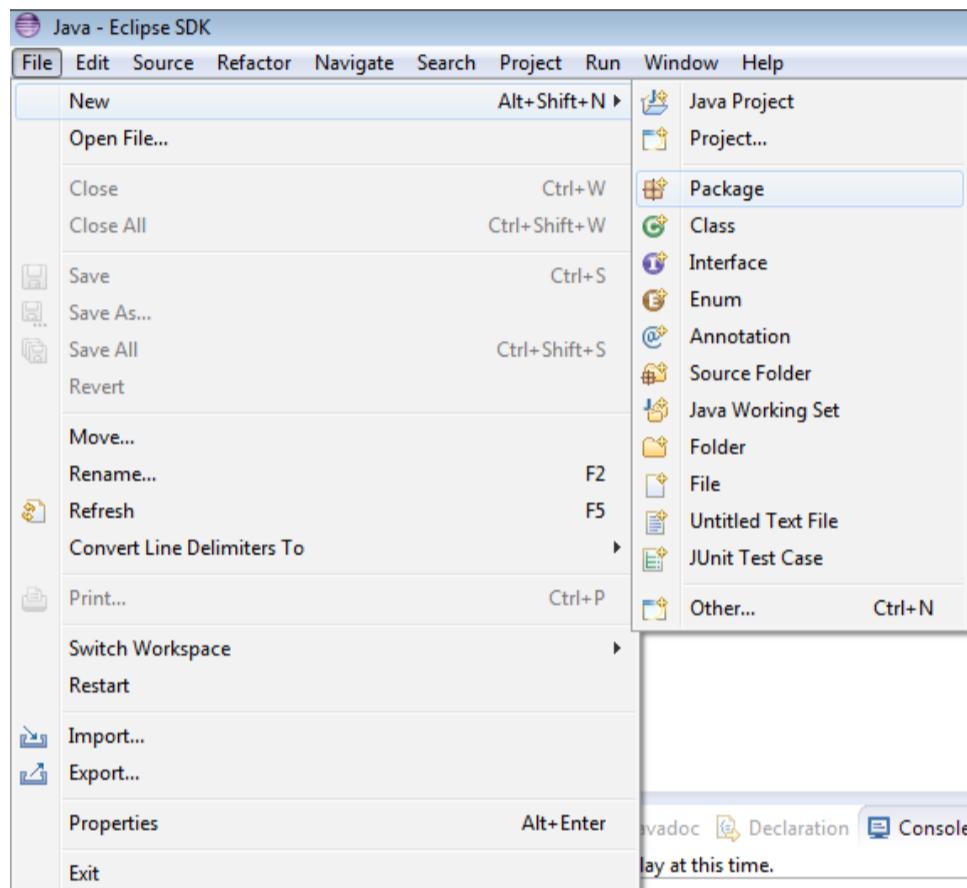
Pacotes representam namespaces em Java. Um namespace é um contexto de nomeação que pode ser usado para prover segurança e encapsulamento às suas classes. Não é possível ter duas classes com o nome igual no mesmo namespace. Logo, em pacotes diferentes, é possível criar classes de mesmo nome. Dessa forma, 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.
```

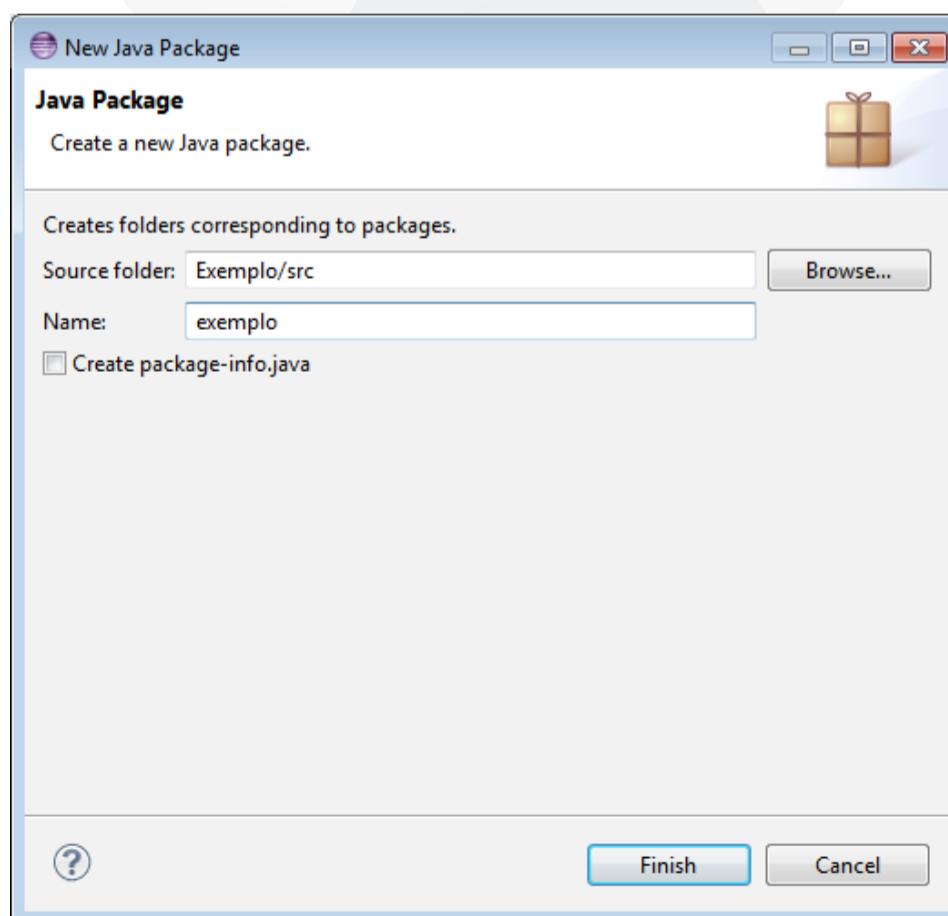
7.3.1.1.Criando um pacote

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

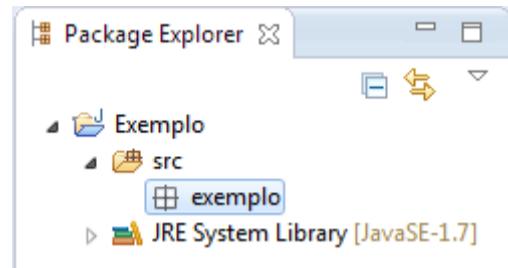
1. Acesse **File / New / Package** para que a janela **New Java Package** seja exibida:



2. Defina a pasta do projeto na qual será criado o pacote e um nome:



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



7.3.1.2. Acessando uma classe em outro pacote

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

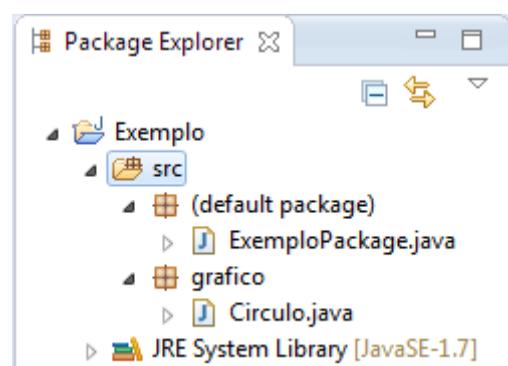
Quando for necessário utilizar uma classe que está em outro pacote, deve-se importar esse pacote no início da definição da classe. Para isso, basta usar o comando **import** e o nome do pacote, ou melhor, o nome completamente qualificado da classe. Observe, no exemplo a seguir, como utilizar esse comando:

```
Circulo.java X ExemploPackage.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 }
```



```
Circulo.java ExemploPackage.java X
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, na imagem a seguir, 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.

7.3.2. Considerações ao declarar uma classe

Como vimos, uma classe pode ser considerada uma abstração de um conjunto de objetos, que são agrupados por comportamento e características semelhantes. Em termos gerais, as classes são capazes de auxiliar na organização de um conjunto de dados, bem como de auxiliar na definição dos métodos mais adequados para a alteração e a utilização desses dados.

Em Java, a criação de classes deve ter início apenas depois de estabelecidas as classes que formarão a aplicação. Você já viu, anteriormente, como é a estrutura de uma classe e como declará-la. Para que a declaração de classes seja feita sempre de maneira adequada, é preciso analisar algumas regras, descritas a seguir:

- **Classe pública**

Cada arquivo de código-fonte pode conter somente uma classe pública e seu nome deve ser igual ao da classe em questão.

- **Instruções**

Todas as classes pertencentes ao arquivo de código-fonte apresentam instruções, sejam elas de pacote ou de importação. Caso a classe faça parte de um pacote, a instrução apresentada na primeira linha do arquivo deve ser uma instrução de pacote.

As instruções de importação devem ser inseridas entre a declaração da classe e a instrução de pacote. Caso não haja uma instrução de pacote, elas devem ser situadas na primeira linha do arquivo.

Se não houver instrução de pacote nem de importação, a primeira linha do arquivo de código-fonte deve conter a declaração da classe.

- **Modificadores**

Antes que uma classe seja declarada, podemos adicionar modificadores a ela. Eles podem ser ou não de acesso. Entre os modificadores de acesso, as únicas opções disponíveis para classes são o **public** e o **default** (modificador padrão). Entre os modificadores que não são de acesso, estão o **abstract** e o **final**.

7.3.3. Encapsulamento

Em Java, a classe é o alicerce para o encapsulamento, que permite ocultar os seus atributos, de modo que eles só possam ser lidos ou alterados pelos métodos da própria classe. Como vimos, isso torna desnecessário o conhecimento a respeito da implementação interna da classe, responsabilidade dos seus métodos internos.

Quando uma classe é criada, são definidos os seus membros: o código, responsável por formar os métodos (métodos membro); e os dados, chamados variáveis de membro ou variáveis de instância.

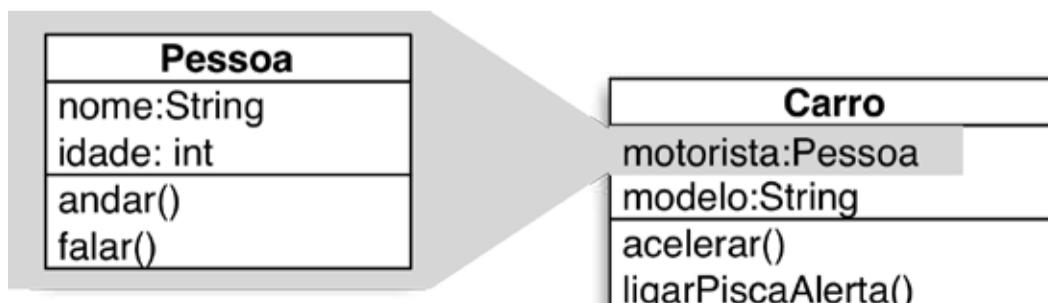
Os métodos presentes em um programa em Java são responsáveis por determinar a maneira como serão utilizadas as variáveis membro, de modo que são eles que determinam qual será a interface e o comportamento apresentados pela classe.

Como os métodos e variáveis de uma classe podem ser definidos como públicos ou privados, tudo o que o usuário externo precisa conhecer a respeito de uma classe encontra-se naquilo que chamaremos de interface pública. Além disso, apenas os códigos membros da classe são capazes de acessar seus métodos e variáveis privados. Isso garante que não ocorram ações inadequadas, mas também exige que a interface pública seja planejada com cautela para que o funcionamento interno da classe não seja muito exposto.

A vantagem do uso de encapsulamento em Java está no fato de o código ser facilmente acessível, podendo ser usado de maneira segura, sem que haja a necessidade do conhecimento dos detalhes de programação envolvidos.

7.3.4. 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:

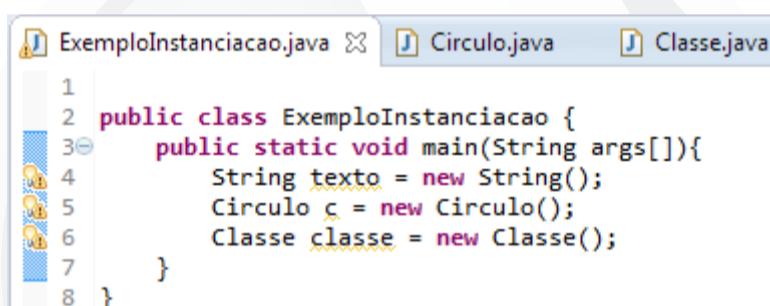


O processo de criação de variáveis por meio de tipos construídos recebe o nome de instanciação. Este é um processo diferente da criação de variáveis de tipo primitivo, tais como **int**, **char**, **boolean**, **long**, **double**, entre outras.

7.3.5. Instanciação

A instanciação é um processo por meio do qual se cria uma cópia, em memória, de um objeto a partir de uma classe existente. Uma classe, que tem a função de determinar um tipo de dado, deve ter objetos instanciados para que possamos utilizá-la. Ou seja, devemos criar uma instância dela, um objeto referente ao tipo de dado que foi definido pela classe. Ressaltamos que, com exceção feita às classes **abstract** (que serão vistas mais adiante em nosso curso), qualquer outra classe pode ter objetos instanciados como um tipo de dado de Java.

Para compreender de forma mais clara o processo de instanciação, veja o exemplo a seguir:



```
1  public class ExemploInstanciacao {
2      public static void main(String args[]){
3          String texto = new String();
4          Circulo c = new Circulo();
5          Classe classe = new Classe();
6      }
7  }
```

7.3.6. 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**

```
1  public class Exemplo1Atribuicao {
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     }
17 }
18 }
```

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
```

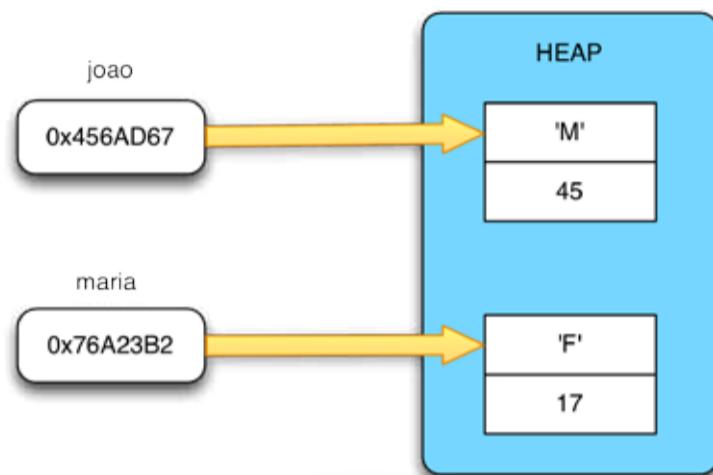
- Exemplo 2

```
1  public class Exemplo2Atribuicao {
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
25 }
```

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

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

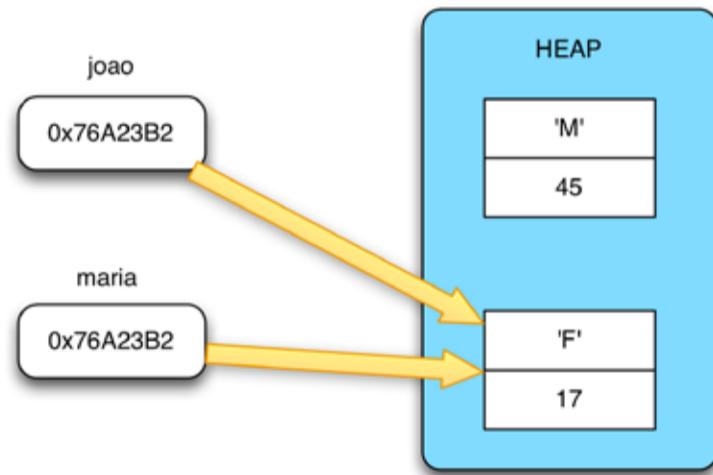
Esquematicamente, veja o que ocorre com a declaração e a instanciação dos objetos **joao** e **maria**, ambos do tipo **Pessoa**:



Após a atribuição realizada, temos:

```
joao = maria;
```

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



7.3.7. Variáveis não inicializadas

Conforme já vimos, antes de utilizar um objeto, devemos instanciá-lo, alocando espaço em memória para preenchimento de todos os seus atributos.

A não instanciação de um objeto pode acarretar problemas em sua aplicação, gerando erros de compilação ou em tempo de execução.

7.3.7.1. Variáveis locais 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 main method with a local variable 'Pessoa pes'. The line 'pes.nome = "Manuel";' is highlighted in red, and the line 'pes.idade = 25;' is also highlighted in red. A tooltip window appears over the second line, containing the message 'The local variable pes may not have been initialized' and '1 quick fix available: Initialize variable'. The cursor is positioned at the end of the line 'pes.idade = 25;'. The status bar at the bottom right says 'Press 'F2' for focus'.

```
1
2 public class CadastroDePessoas {
3
4     public static void main(String[] args) {
5
6         Pessoa pes;
7
8         pes.nome = "Manuel";
9         pes.idade = 25;
10    }
11
12 }
```

7.3.7.2. Variáveis de classe não inicializadas

As variáveis de classe (também chamadas de atributos) são aquelas declaradas dentro da classe, mas do lado de fora do método ou da função.

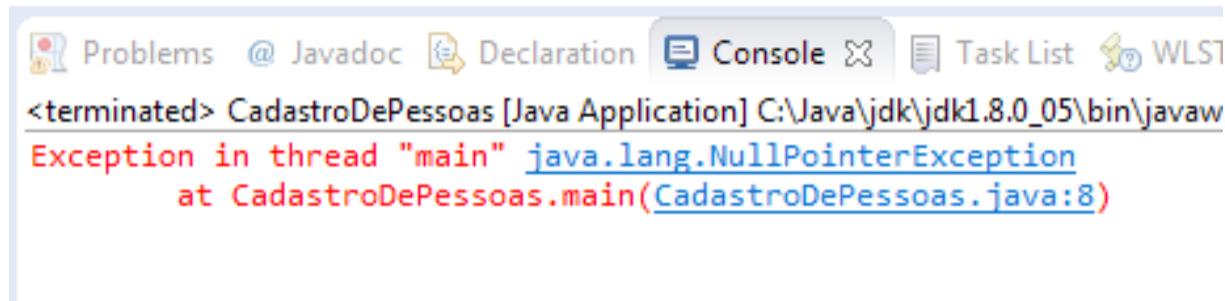
Quando estamos utilizando uma variável de classe em nossa aplicação, o Java não consegue detectar se ela foi inicializada (instanciada) ou não em tempo de compilação, o que pode provocar problemas durante a execução da aplicação.

No exemplo adiante, temos o mesmo código do exemplo anterior, com a diferença de que a variável **pes**, desta vez, foi declarada fora da função/método:

A screenshot of an IDE showing a Java code editor. The file is named 'CadastroDePessoas.java'. The code contains a static variable 'static Pessoa pes;' declared outside the main method. The line 'pes.nome = "Manuel";' is highlighted in red, and the line 'pes.idade = 25;' is also highlighted in red. A tooltip window appears over the second line, containing the message 'The local variable pes may not have been initialized' and '1 quick fix available: Initialize variable'. The cursor is positioned at the end of the line 'pes.idade = 25;'. The status bar at the bottom right says 'Press 'F2' for focus'.

```
1
2 public class CadastroDePessoas {
3
4     static Pessoa pes;
5
6     public static void main(String[] args) {
7
8         pes.nome = "Manuel";
9         pes.idade = 25;
10    }
11
12 }
```

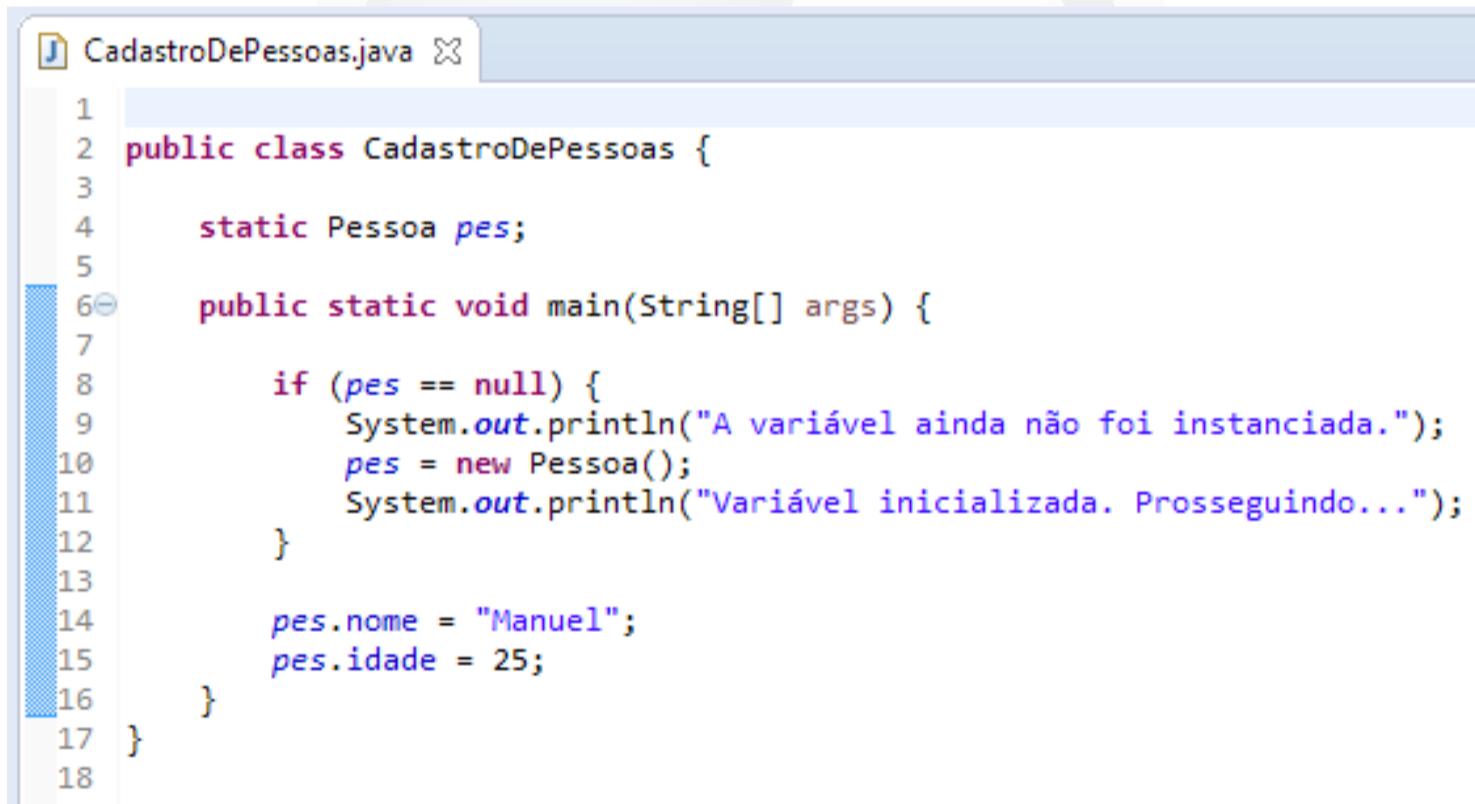
O código anterior compila com sucesso. Porém, se você tentar executá-lo, verá o seguinte erro:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:
<terminated> CadastroDePessoas [Java Application] C:\Java\jdk\jdk1.8.0_05\bin\javaw
Exception in thread "main" java.lang.NullPointerException
at CadastroDePessoas.main(CadastroDePessoas.java:8)

 No Java, os problemas que ocorrem em tempo de execução, como no exemplo anterior, são chamados de exceções (exceptions). Este é um tema que será abordado com detalhes posteriormente.

Desta forma, caso haja necessidade, podemos verificar se aquela variável já foi instanciada. Para isso, basta comparar o conteúdo da variável com o valor **null**, antes de tentar utilizá-la:



```
1 public class CadastroDePessoas {  
2     static Pessoa pes;  
3  
4     public static void main(String[] args) {  
5  
6         if (pes == null) {  
7             System.out.println("A variável ainda não foi instanciada.");  
8             pes = new Pessoa();  
9             System.out.println("Variável inicializada. Prosseguindo...");  
10        }  
11  
12        pes.nome = "Manuel";  
13        pes.idade = 25;  
14    }  
15}  
16}
```

7.3.8. Acesso

Quando falamos de acesso a classes, estamos nos referindo a um conceito de visibilidade, ou seja, uma determinada classe deve ser capaz de visualizar a outra para poder acessar seus métodos e variáveis.

Para compreendermos esse conceito de maneira adequada, destacamos duas classes distintas: a classe **X** e a classe **Y**. A classe **X** é capaz de criar uma instância da classe **Y**, bem como pode tornar-se uma subclasse de **Y** e acessar seus métodos ou variáveis de acordo com o tipo de controle de acesso de ambos.

Veja, adiante, os dois tipos de acesso às classes disponíveis na linguagem Java: padrão (default) e público.

7.3.8.1. Padrão (Default)

Uma classe com acesso padrão não é precedida por um modificador em sua declaração. O acesso padrão é considerado o acesso ao nível do pacote, uma vez que classes com esse tipo de acesso são encontradas somente pelas classes que fazem parte do mesmo pacote.

Caso duas classes estejam em pacotes distintos e uma delas possua acesso padrão, a outra deverá desconsiderar sua existência para que não haja problema durante a compilação.

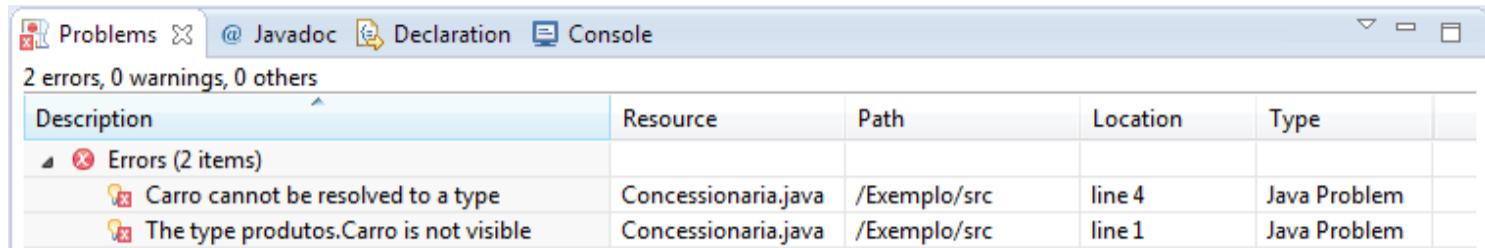
Observe os exemplos a seguir, nos quais temos dois arquivos de código-fonte:

```
Carro.java
1 package produtos;
2
3 class Carro {
4
5 }
```

```
Carro.java Concessionaria.java
1 import produtos.Carro;
2
3 public class Concessionaria {
4     Carro c;
5 }
```

Tendo em vista que há uma instrução de importação no começo do arquivo da classe **Concessionaria**, e que esta realiza uma tentativa de importar **Carro**, a compilação do arquivo **Concessionaria.java** apresentará problemas, ao contrário da compilação do arquivo **Carro.java**.

Veja a mensagem resultante do processo de compilação da classe **Concessionaria**:



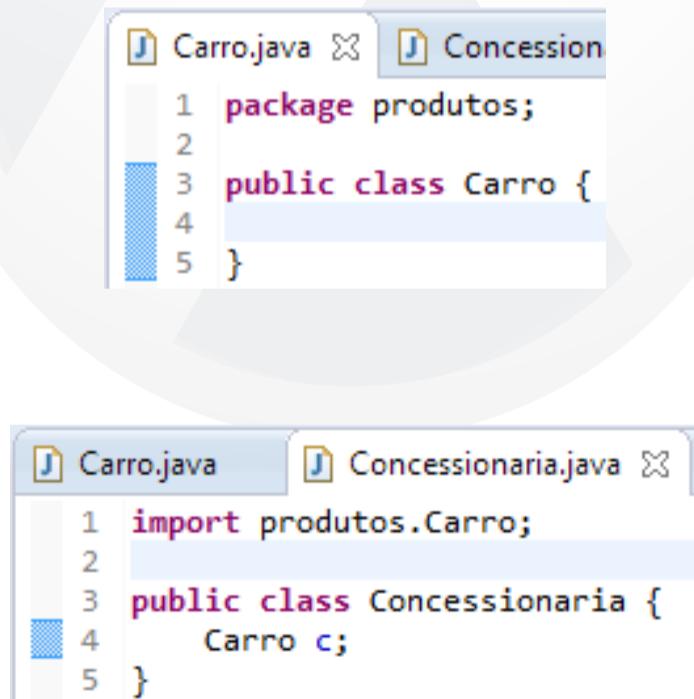
Problems				
2 errors, 0 warnings, 0 others				
Description	Resource	Path	Location	Type
Errors (2 items)				
Carro cannot be resolved to a type	Concessionaria.java	/Exemplo/src	line 4	Java Problem
The type produtos.Carro is not visible	Concessionaria.java	/Exemplo/src	line 1	Java Problem

O processo de compilação do arquivo **Concessionaria.java** apresenta problemas porque a classe **Carro** encontra-se em um pacote distinto do seu e possui acesso **default**. Para solucionar esse problema, podemos colocar as duas classes em um único pacote ou, ainda, declarar a classe **Carro** com o modificador de acesso **public**.

7.3.8.2. PÚBLICO (Public)

Como já vimos, quando você usar duas classes de pacotes diferentes, deve usar o comando **public** para que uma classe possa acessar classes de outros pacotes.

Observe o exemplo a seguir:



```
Carro.java
1 package produtos;
2
3 public class Carro {
```

```
Concessionaria.java
1 import produtos.Carro;
2
3 public class Concessionaria {
4     Carro c;
5 }
```

Nesse exemplo, a classe **Carro** pode ser visualizada por todas as classes de Java e, portanto, seus objetos podem ser instanciados a partir delas.

Todas as classes de Java têm acesso a uma classe pública. Mas, para utilizar uma classe pública pertencente a um pacote distinto da classe criada, é preciso que a classe pública seja importada.

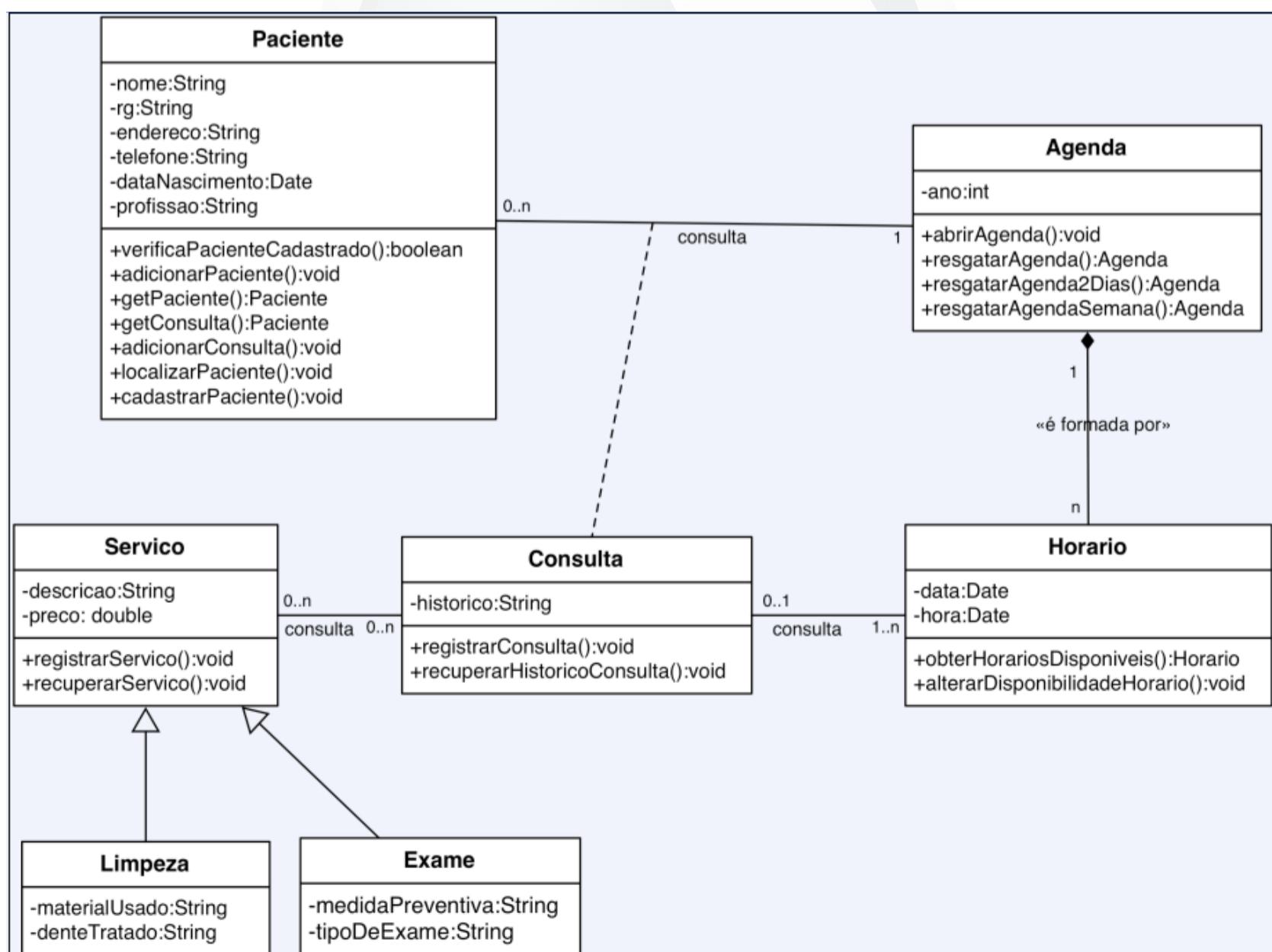
7.3.9. UML – Diagrama de classes

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.

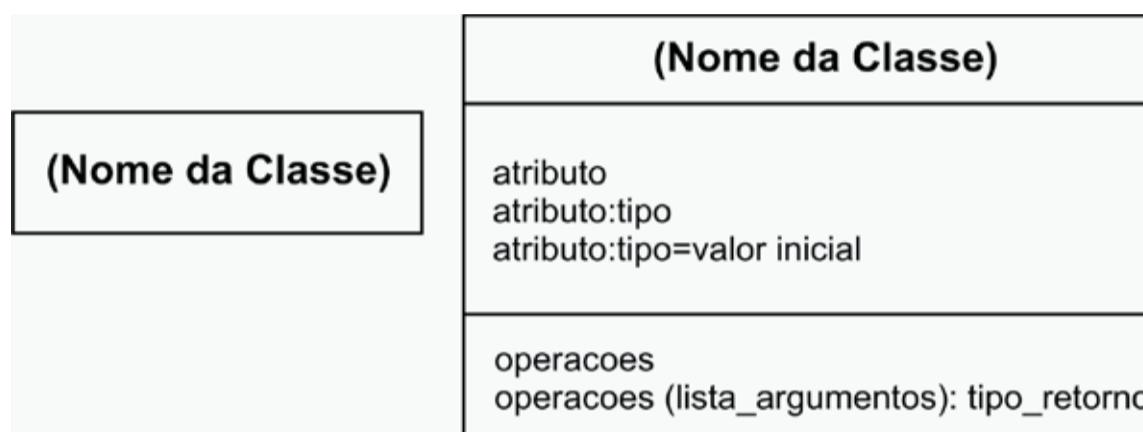
O diagrama de classes descreve a estrutura estática das classes de um sistema, o que inclui o nome da classe, seus atributos e operações e seus relacionamentos com outras classes.

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



7.3.9.1. Diagrama de classes em detalhes

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):



7.4. Métodos

Nesta leitura, já definimos brevemente o conceito de métodos. No decorrer deste tópico, abordaremos o assunto de forma mais ampla.

7.4.1. Acesso a métodos

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.

A seguir, descrevemos os modificadores de acesso disponíveis na linguagem Java, todos aplicáveis às definições de métodos:

- **public**: Quando você utiliza o modificador **public**, significa que o método pode ser chamado a partir de outros métodos de qualquer classe. Isto ocorre porque o método é público;
- **protected**: Quando você utiliza o modificador **protected**, significa que o método pode ser chamado pelas classes derivadas da classe que possui o método **protected**, independentemente de a classe derivada pertencer ou não ao mesmo pacote, e por todas as classes que pertencem ao mesmo pacote da classe onde o método foi declarado;

Este assunto será explicado com mais detalhes na leitura que trata sobre herança.

- **private**: Utilizando este modificador, você indica que o método é particular da classe em que ele se encontra. Sendo assim, ele somente pode ser utilizado por membros internos desta mesma classe;
- **default**: Este modificador de acesso é o padrão. Você o aplica sempre que não for definido outro modificador explicitamente. Com ele, o método pode ser utilizado apenas pelas classes que pertencem ao mesmo pacote.

7.4.2. 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 uma leitura futura).

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 mais clareza e detalhes na leitura complementar 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 em uma leitura futura.

7.4.3. 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:

The screenshot shows an IDE interface with two code editors. The left editor contains the code for the **Empregado** class, and the right editor contains the code for the **TesteEmpregado** class.

Empregado.java:

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

TesteEmpregado.java:

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

Nesse exemplo, a palavra **this** refere-se aos atributos definidos na classe, e não aos parâmetros dos métodos. Observando o método **mostrar()**, você verifica 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.

7.4.4. Métodos recursivos

Há duas versões de rotinas, quando se trata de realizar chamadas encadeadas e repetitivas: as recursivas e as iterativas.

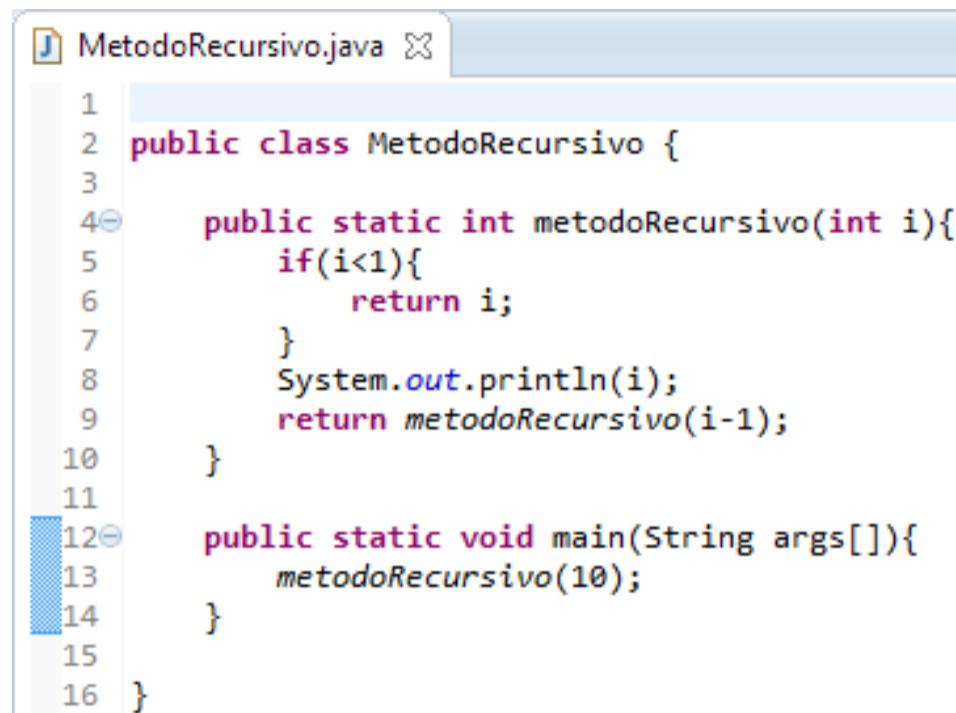
Quando um método realiza uma chamada a si mesmo, tanto as variáveis locais quanto os parâmetros criados por cada chamada ficam armazenados nas pilhas. Ao realizar a chamada a um método recursivo, apenas os argumentos são novos, ou seja, não são criadas novas cópias do método.

Sendo assim, as variáveis locais e os parâmetros antigos são removidos da pilha quando cada uma das chamadas recursivas retorna, e o código do método é executado com as novas variáveis desde o começo (ponto de chamada dentro do método).

Em razão de as variáveis locais e os parâmetros serem armazenados na pilha, a cada chamada realizada ao método são criadas cópias dessas variáveis e desses parâmetros, e isso pode fazer com que a pilha se esgote. Quando isso acontece, temos o que conhecemos por “estouro de pilha”, que é uma grande quantidade de chamadas recursivas a um método. Neste caso, o sistema **runtime**, de Java, causará uma exceção.

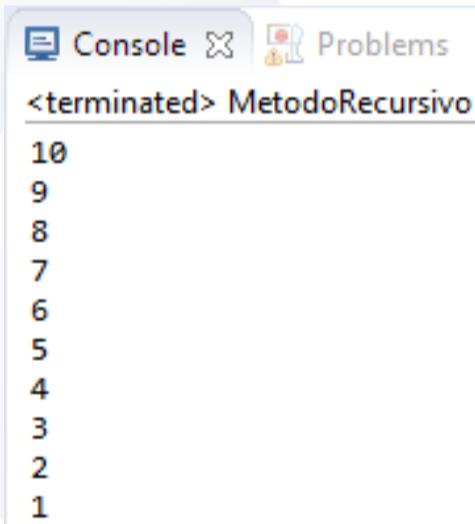
É importante ressaltar que, para forçar o retorno de um método sem que a chamada recursiva seja executada, você deve utilizar uma condição **if** no código. Caso contrário, o método não será retornado quando for chamado. Ainda, para que você possa acompanhar o algoritmo recursivo enquanto ele está sendo implementado, você pode utilizar o método **println()**. Com ele, é possível visualizar, passo a passo, cada uma das partes do código que está sendo executado.

Observe o exemplo a seguir:



```
1
2  public class MetodoRecursivo {
3
4      public static int metodoRecursivo(int i){
5          if(i<1){
6              return i;
7          }
8          System.out.println(i);
9          return metodoRecursivo(i-1);
10     }
11
12     public static void main(String args[]){
13         metodoRecursivo(10);
14     }
15
16 }
```

O resultado será o seguinte:



Console Problems
<terminated> MetodoRecursivo

```
10
9
8
7
6
5
4
3
2
1
```

7.4.5. Métodos acessores

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 acessores 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 essas 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**.

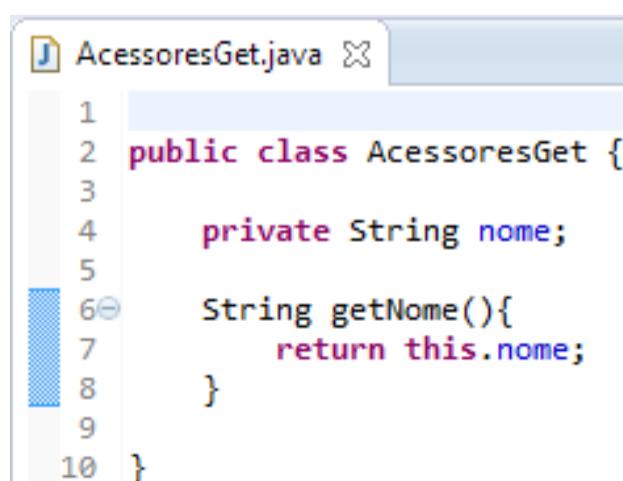
7.4.5.1. Método getter

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:



```
J AcessoresGet.java X
1
2 public class AcessoresGet {
3
4     private String nome;
5
6     String getNome(){
7         return this.nome;
8     }
9
10 }
```

7.4.5.2. Método setter

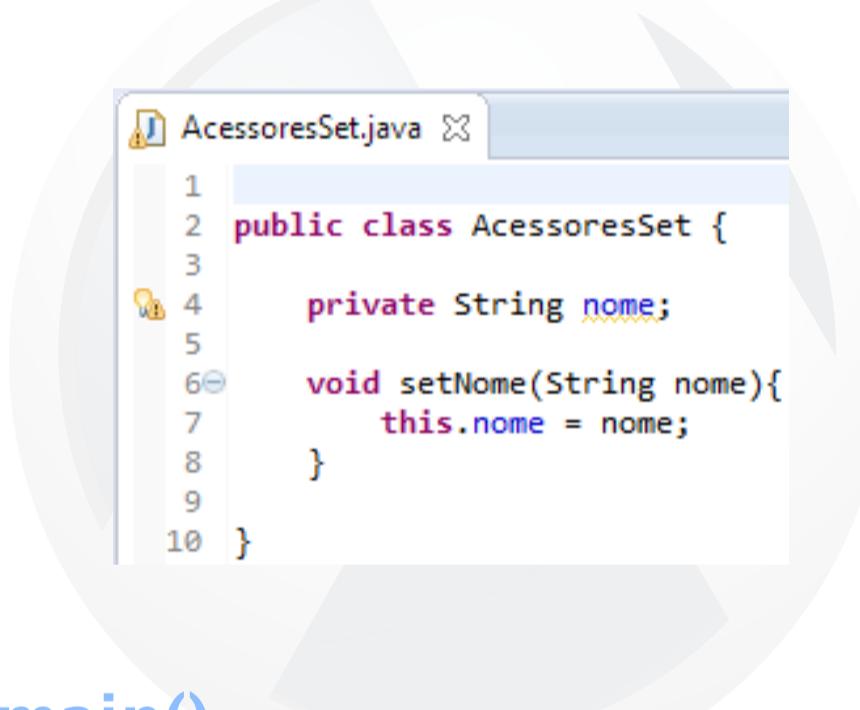
Para configurar o valor apresentado pelos atributos em uma classe, você dispõe do método **setter**, cuja sintaxe é a seguinte:

```
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  
3  
4     private String nome;  
5  
6     void setNome(String nome){  
7         this.nome = nome;  
8     }  
9  
10 }
```

7.4.6. Método main()

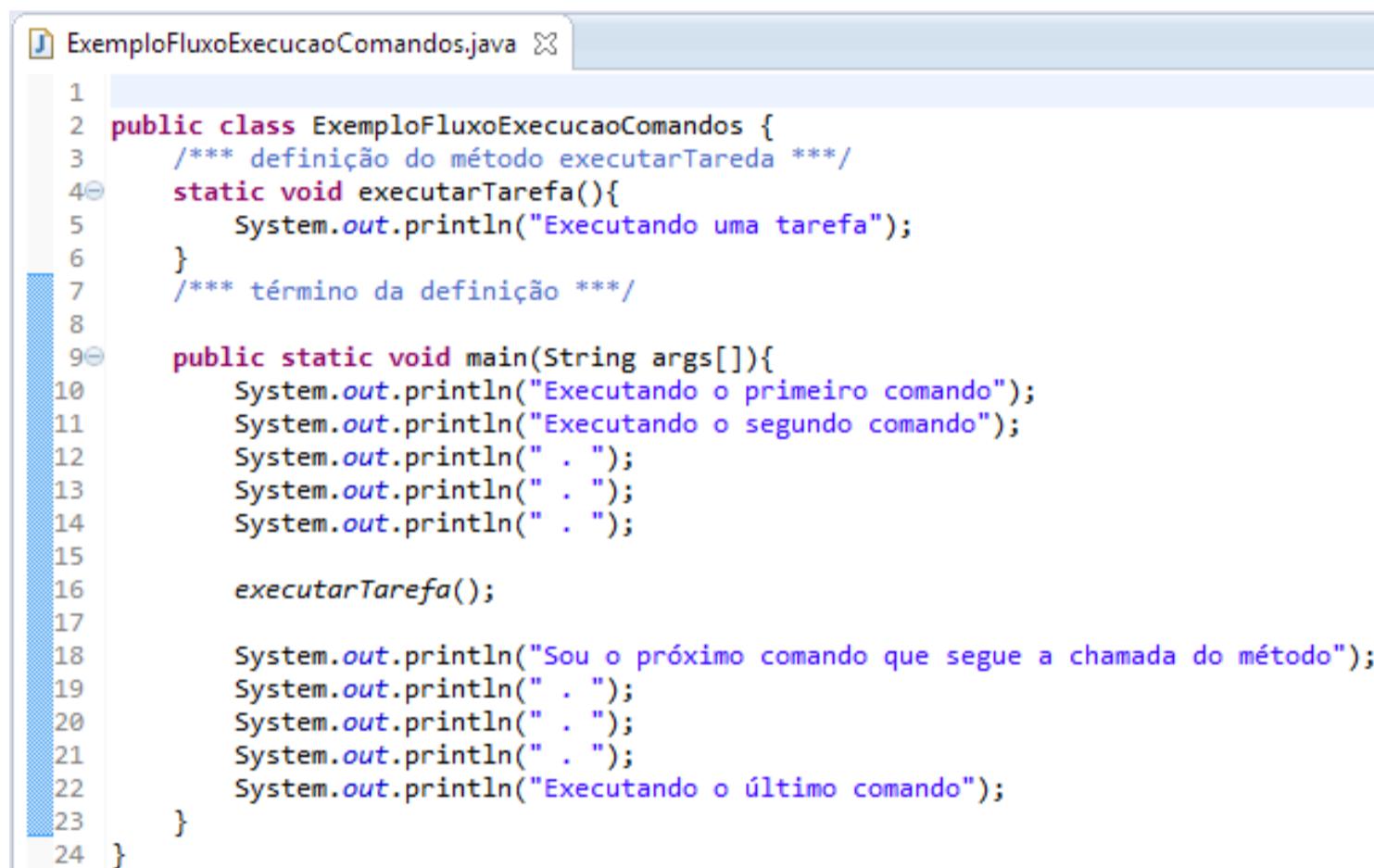
O **main()** é o método static mais famoso na linguagem Java. É por meio dele que a aplicação passa a ser executada. Portanto, quando você deseja utilizar algum método ou realizar alguma operação no programa, faça isso dentro do método **main()**.

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**;

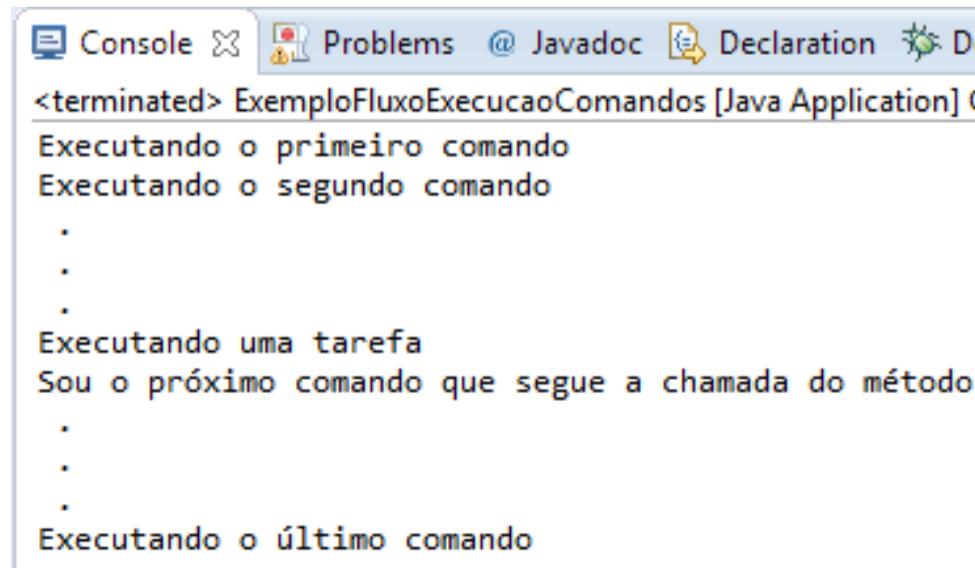
- Deve ser **void**;
- Deve receber como argumento um array de tipos String (**String args[]**).

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



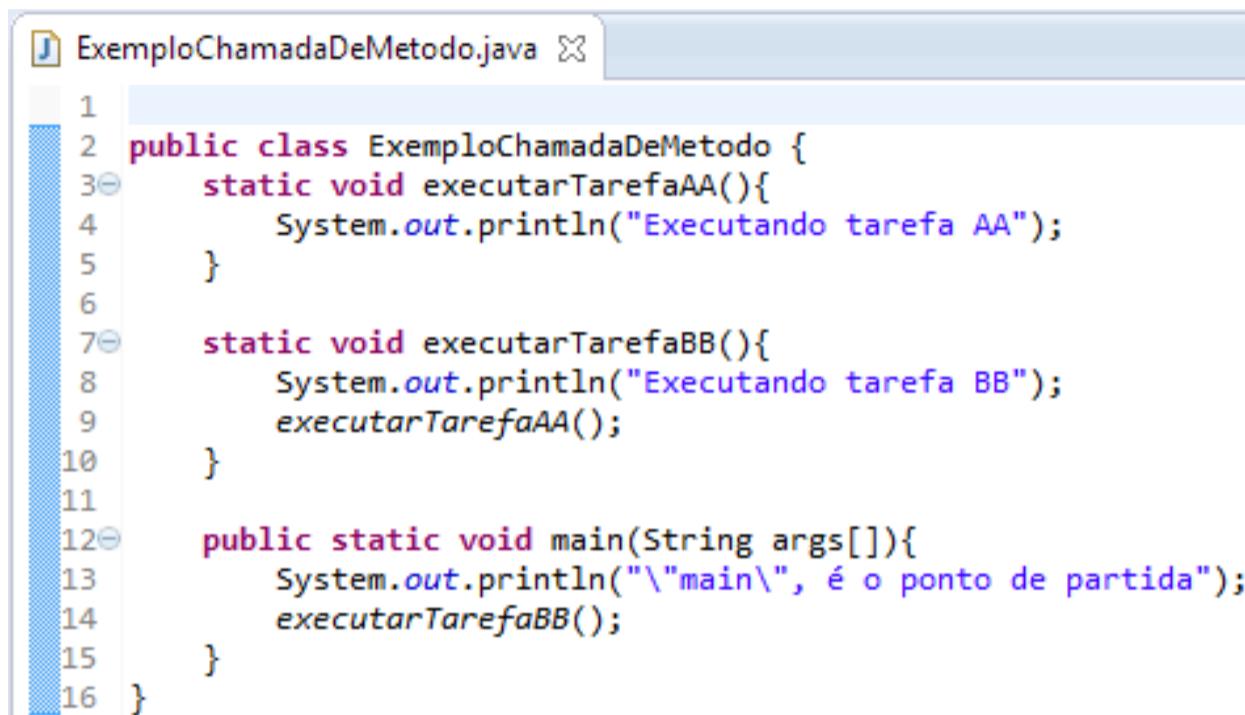
```
1  public class ExemploFluxoExecucaoComandos {
2      /** definição do método executarTarefa **/
3      static void executarTarefa(){
4          System.out.println("Executando uma tarefa");
5      }
6      /** término da definição **/
7
8
9      public static void main(String args[]){
10         System.out.println("Executando o primeiro comando");
11         System.out.println("Executando o segundo comando");
12         System.out.println(" . ");
13         System.out.println(" . ");
14         System.out.println(" . ");
15
16         executarTarefa();
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 imagem a seguir:



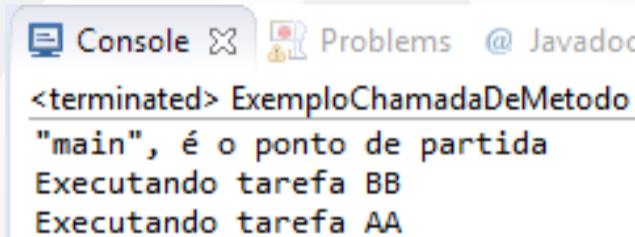
```
Console Problems @ Javadoc Declaration De
<terminated> ExemploFluxoExecucaoComandos [Java Application] C
Executando o primeiro comando
Executando o segundo comando
.
.
.
Executando uma tarefa
Sou o próximo comando que segue a chamada do método
.
.
.
Executando o último comando
```

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



```
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 imagem a seguir:



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

7.4.7. 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. Esse 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:

The screenshot shows an IDE interface with two code editors. The top editor, titled 'SobreCarga.java', contains the following code:

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

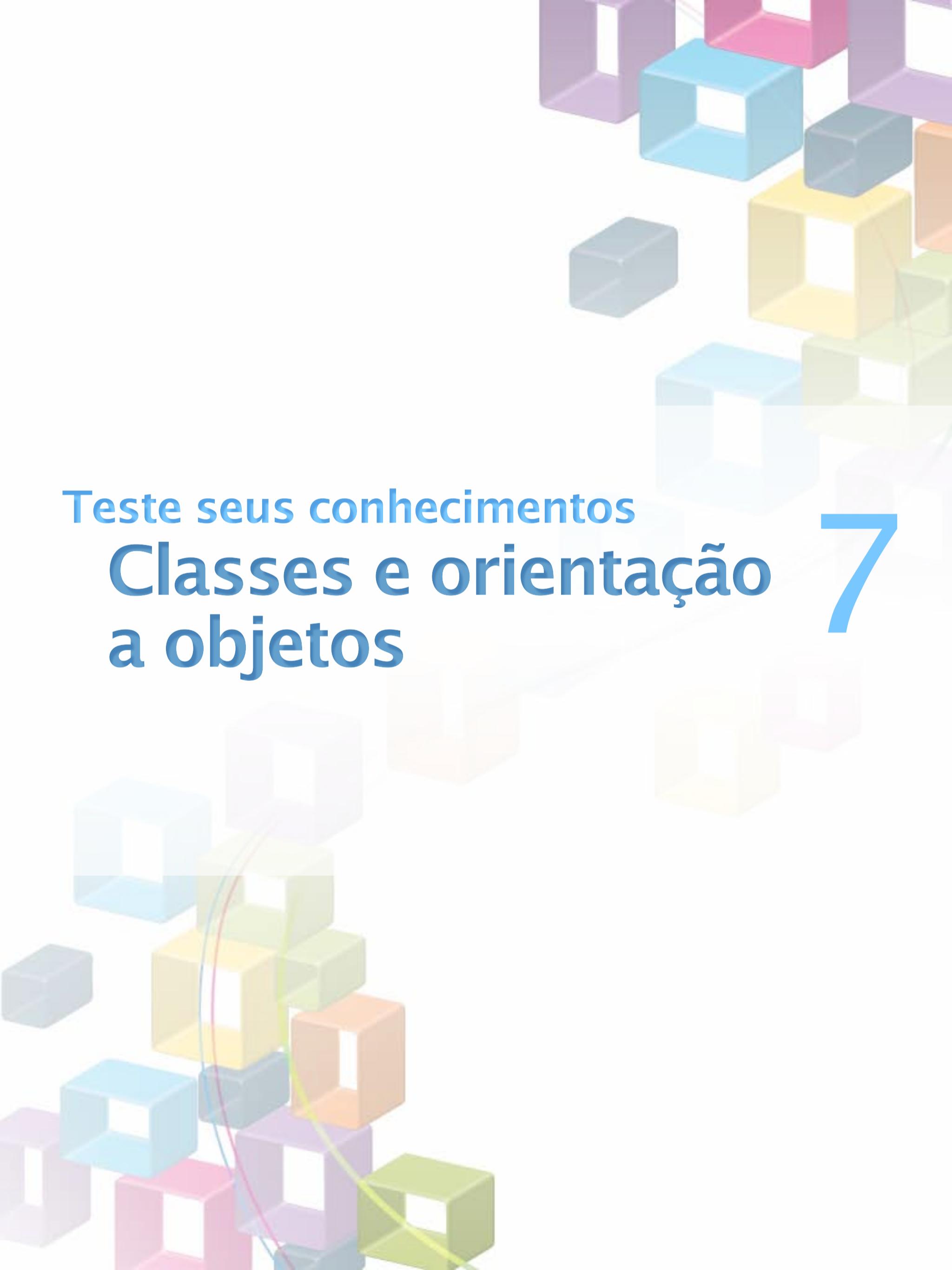
The bottom editor, titled 'ExemploSobreCarga.java', contains the following code:

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

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

The screenshot shows the 'Console' tab of the IDE displaying the following output:

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

Teste seus conhecimentos Classes e orientação a objetos 7

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 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.

5. Qual é o mecanismo que possibilita a uma classe herdar atributos e métodos de outra?

- a) Polimorfismo
- b) Herança
- c) Encapsulamento
- d) Instanciação
- e) Nenhuma das alternativas anteriores está correta.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

Construtores

8

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

8.1. 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 isso, 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 sobrecarregados 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, algumas passagens de parâmetros podem ser recebidas pelo construtor. Além disso, você deve reparar que os construtores podem estar no início da classe, depois da declaração de atributos, bem como podem ser adicionados em outro local, antes da chave de encerramento da classe, por exemplo.

8.2. 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, como no 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 método 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 }
```

The constructor Calcado() is undefined

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 método construtor:

The screenshot shows an IDE interface with two tabs open. The top tab is titled "Carro.java" and contains the following code:

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

The bottom tab is titled "ExemploUsoConstrutor.java" and contains the following code:

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

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

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

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

Implicit super constructor Calcado() is undefined for default constructor.

8.3. 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. Caso a classe do construtor privado permita o uso de uma parte de sua instância, é fundamental que ela disponha de uma variável ou método estático para que uma instância gerada dentro da classe possa ser acessada. Essa técnica é usada em diversos padrões de projeto aplicáveis em Java;
- 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;

- A chamada a um construtor sobreescrito (**this()**) ou ao construtor da superclasse (**super()**) deve ser a primeira instrução de todo construtor;
- Se você não digitar a chamada a **super()** quando inserir um construtor, uma chamada a **super()** sem argumentos será incluída automaticamente pelo compilador do Java. Lembre-se que uma chamada a **super()** pode tanto ter os argumentos que foram passados para o construtor da superclasse como pode não conter nenhum argumento. O conceito de superclasse ficará mais claro no momento em que for abordado o assunto da herança. Retorne posteriormente a este ponto para repassar o conteúdo de construtores;
- Uma chamada ao método de uma instância ou acesso a uma variável de instância não pode ser feita até que o construtor da superclasse tenha sido executado;
- As classes abstratas possuem construtores. Toda vez que uma subclasse concreta é instanciada, estes construtores são chamados;
- Por não fazerem parte da árvore de herança de um objeto, as interfaces não possuem construtores.



**Teste seus conhecimentos
Construtores**

8

1. O que é um construtor?

- a) É o método responsável por construir objetos com valores aleatórios.
- b) É o método responsável por construir objetos com valores determinados.
- c) É o método responsável por inicializar os atributos de uma classe.
- d) É o método 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.

Membros estáticos 9

- ✓ Modificador static;
- ✓ Atributos estáticos;
- ✓ Métodos estáticos;
- ✓ Exemplos práticos de membros estáticos.

9.1. 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 dessa classe para acessar os membros estáticos fora dela. Por esse motivo, você pode se referenciar a eles como membros de classe.

9.2. 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's console window. The window title is "Console". Below it, the text "<terminated> AtributoEstatico" is displayed. Underneath, two lines of text are shown: "2012" and "2013".

9.3. Métodos estáticos

Ao colocar o modificador **static** juntamente com a definição de um método, este se torna estático. Isso não significa que o seu valor não possa mudar, mas que haverá somente 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 file named "Classe.java". The code contains the following:

```
1 public class Classe {
2
3     String nome;
4
5     public static void mostraNome(){
6         System.out.println(nome); // Erro pois a variável nome não é estática
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.

Isso 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.

Java Programmer – Módulo I (online)

154

Veja o exemplo a seguir:

The screenshot shows an IDE interface with two tabs open. The first tab, 'Classe2.java', contains the following code:

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

The second tab, 'MetodoEstatico.java', contains the following code:

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

O resultado será o seguinte:

The screenshot shows the 'Console' tab of the IDE displaying the following output:

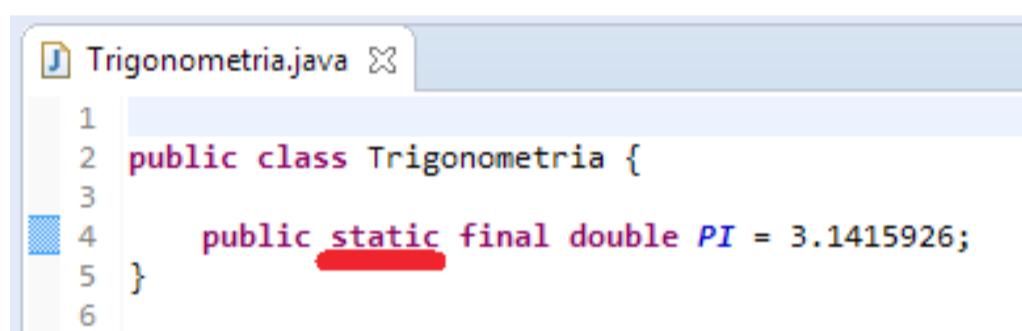
```
Console Problems  
<terminated> MetodoEstatico  
Método Estático  
Método Não Estático
```

9.4. 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:



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

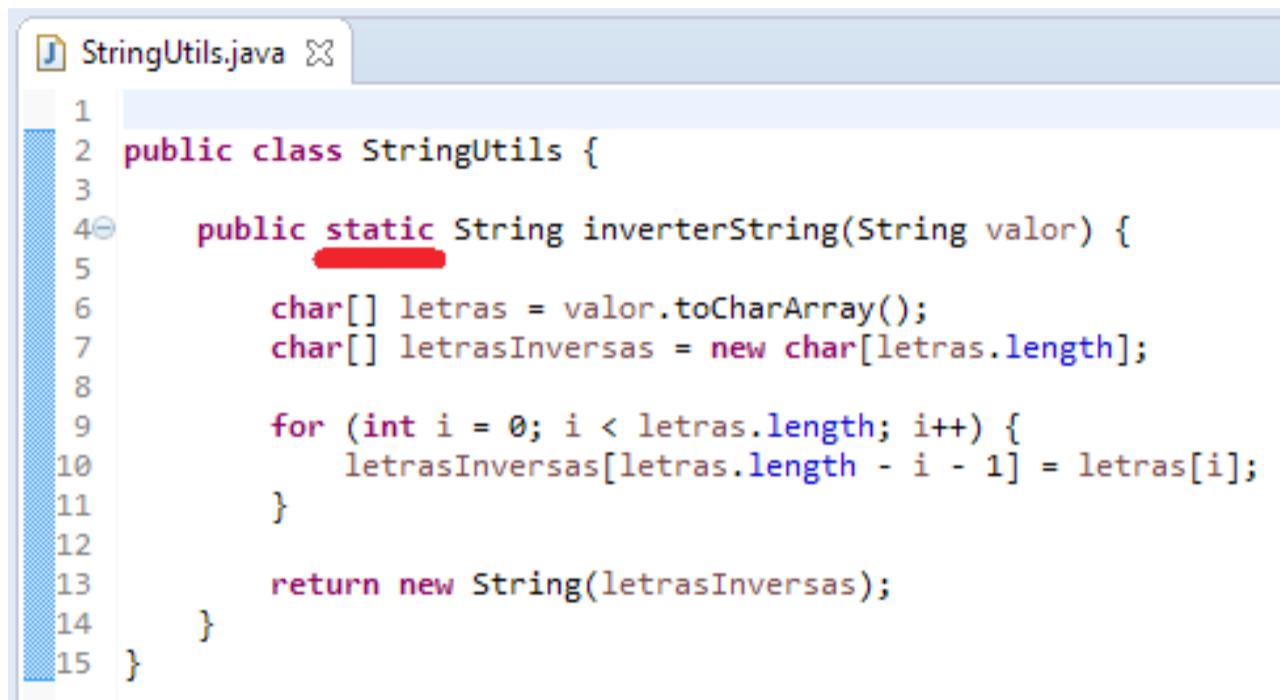
Por ser estático, podemos acessar esse valor sem precisar instanciar a classe:

```
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.

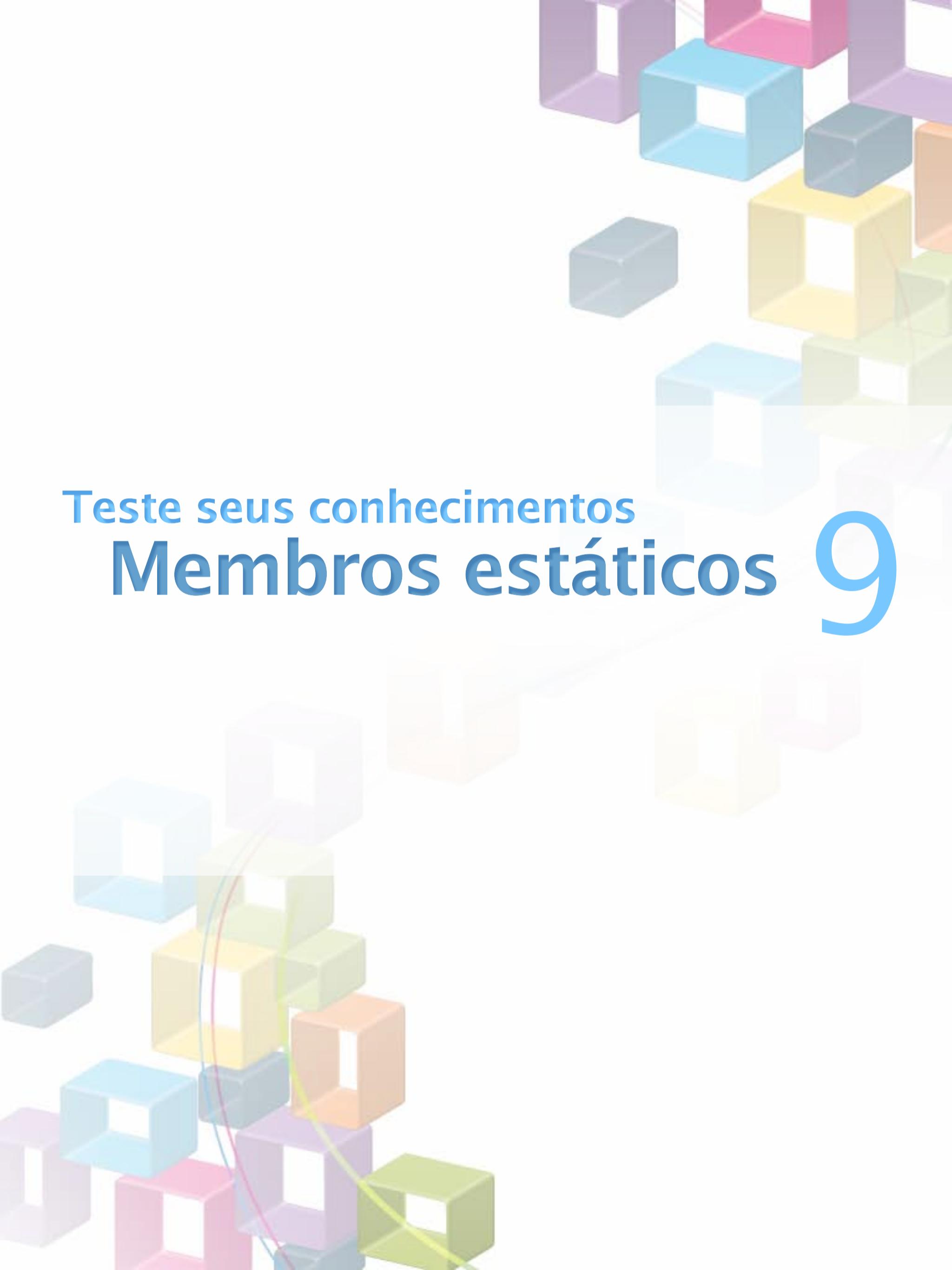
Os métodos utilitários são pequenas rotinas utilizadas no cálculo ou no processamento geral e que podem ser chamados por diversos pontos de sua aplicação:



```
1 public class StringUtils {
2
3     public static String inverterString(String valor) {
4         // Redacted code
5
6         char[] letras = valor.toCharArray();
7         char[] letrasInversas = new char[letras.length];
8
9         for (int i = 0; i < letras.length; i++) {
10             letrasInversas[letras.length - i - 1] = letras[i];
11         }
12
13         return new String(letrasInversas);
14     }
15 }
```

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

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



Teste seus conhecimentos Membros estáticos 9

1. Qual das afirmações sobre o modificador static está correta?

- a) Um atributo se torna estático quando o modificador static é colocado juntamente com sua definição.
- b) Não é necessário criar objetos da classe em que o membro estático foi definido para acessar os membros estáticos fora dela.
- c) Membros estáticos podem ser referenciados pelo próprio nome da classe em que foram definidos.
- d) Para especificar que um membro de uma classe é estático, você deve prefixar à sua declaração o modificador static.
- e) Todas as alternativas anteriores estão corretas.

2. O que é correto afirmar sobre o modificador static?

- a) Se colocado junto ao método, indica que, dentro dele, só é possível acessar outros métodos ou atributos que também sejam declarados como static.
- b) Se colocado junto ao método, indica que, dentro dele, é possível acessar outros métodos ou atributos que não sejam declarados como static.
- c) Ao colocar o modificador static juntamente com a definição de um atributo, este se torna estático, ou seja, o atributo passa a pertencer a um contexto global em sua aplicação e nenhuma instância desse atributo existirá durante o programa.
- d) Ao colocar o modificador static juntamente com a definição de um atributo, este se torna estático, ou seja, o atributo passa a pertencer a um contexto global em sua aplicação e várias instâncias desse atributo existirão durante o programa.
- e) Nenhuma das alternativas anteriores está correta.

3. Qual das seguintes afirmações está correta?

- a) Um membro estático representa uma característica da classe, não do objeto.
- b) Para utilizar um membro estático, não é necessário instanciar sua classe.
- c) Métodos estáticos não podem utilizar diretamente membros não estáticos da própria classe.
- d) Membros estáticos são tipicamente utilizados em duas situações: criação de constantes e de rotinas de utilidade geral.
- e) Todas as alternativas anteriores estão corretas.

Herança

10

- ✓ Herança e generalização;
- ✓ Ligação;
- ✓ Associação;
- ✓ Herança e composição;
- ✓ Estabelecendo herança entre classes;
- ✓ Relacionamentos;
- ✓ Herança e classes;
- ✓ Polimorfismo.

10.1. Introdução

Até aqui, você viu que classes podem herdar atributos e métodos de outras classes. Esse é o conceito básico de herança. Nesta leitura, vamos ampliar esse conceito e abordar, ainda, diversos aspectos que estão relacionados à herança, incluindo associação, relacionamentos e polimorfismo, entre outros.

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 a outra classe;
- **Subclasse:** A classe que está no nível mais baixo na hierarquia e que herda as características da superclasse.

Se uma classe tem alguns atributos encapsulados, a herança atuará de maneira interativa com esse encapsulamento. Assim, uma subclasse dessa classe herdará esses atributos encapsulados, desde que o modificador de acesso assim o permita, além de outros atributos adicionais que fazem parte da especialização da subclasse.

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.

Com o uso da herança, um objeto pode se tornar uma instância específica de um caso mais geral. Isso quer dizer que um objeto pode herdar os atributos gerais da respectiva superclasse. É preciso apenas definir as características que o tornam único na classe à qual pertence.

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**.

Para ilustrar, apresentamos a hierarquia (na biologia, o termo usado é **taxonomia**) de classes do cachorro:

- **Reino:** Animal;
- **Filo:** Cordata (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): dálmata, pastor, dobermann etc.

Considere como exemplo um cão da raça dálmata, pertencente à **espécie** “Canis familiaris”. Por herança, o dálmata 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.

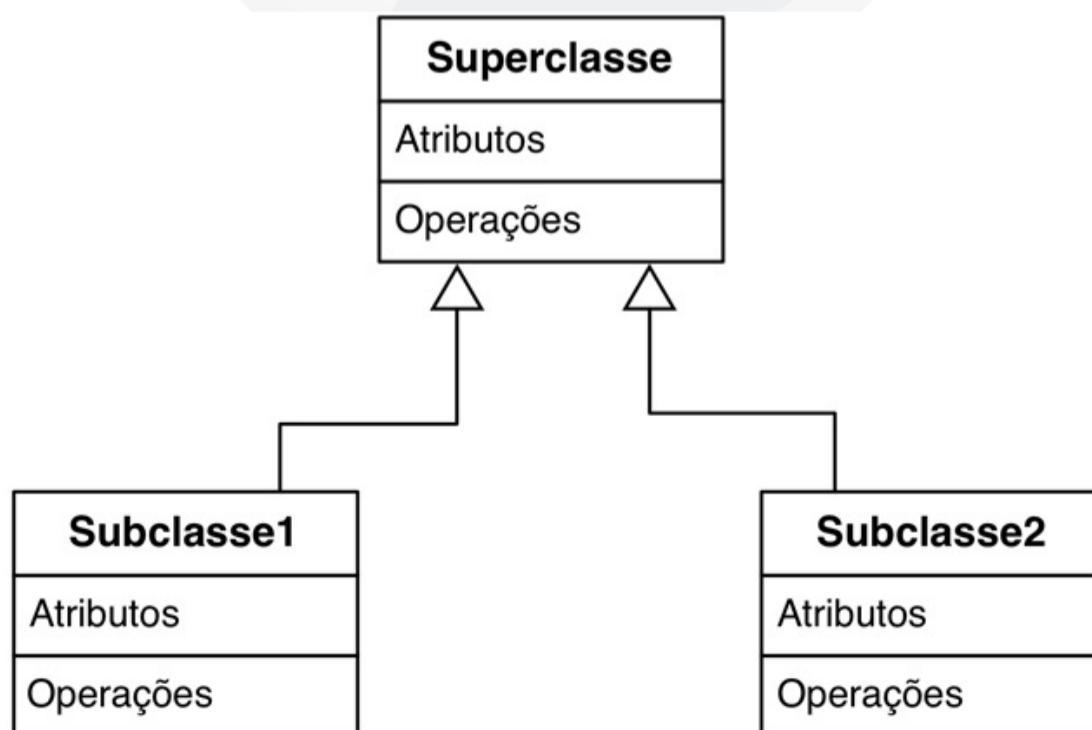
10.2. Herança e generalização

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:



Cada associação do tipo generalização pode possuir um discriminador associado. O discriminador é um atributo (do tipo enumeração) utilizado para indicar qual é a propriedade do objeto que é abstraída pelo relacionamento de generalização. Trata-se de um nome para a base de generalização.

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 chamamos de sobrescrição.

10.3. Ligação

Uma ligação 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. Veja:



10.4. Associação

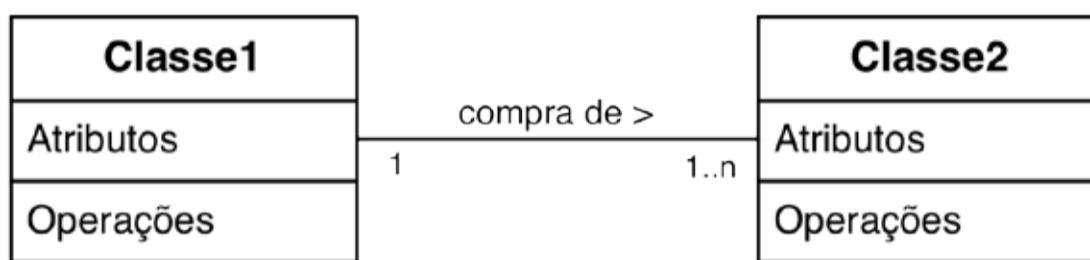
Uma associação define um conjunto de ligações entre instâncias de duas ou mais classes, ou um conjunto de ligações que compartilham a mesma semântica e estrutura. As associações descrevem grupos de ligações potenciais. Como exemplo de associação, podemos dizer: **Uma pessoa é aluna de uma faculdade**.

Na notação UML, uma associação é representada por uma linha conectando duas classes, como mostrado a seguir:



- **Atributo de ligação**

Em diversos casos, é preferível 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:



10.4.1. Tipos de associação

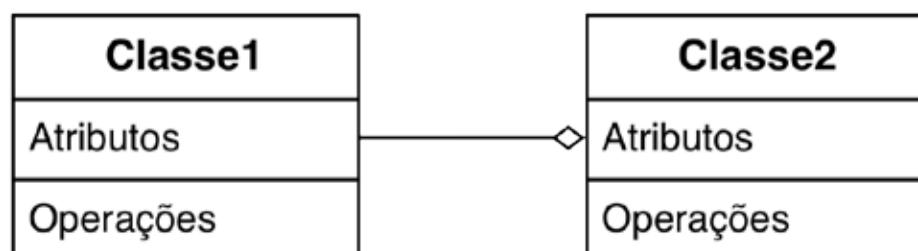
Há dois tipos de associação: a agregação e a composição.

10.4.1.1. 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. Isso 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:



10.4.1.2. 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 de contenção, 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 produto comprado. Esse 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:



10.5. Herança e 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**

Na composição, uma classe é estendida e o trabalho é delegado para um objeto dessa classe. 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 }
```

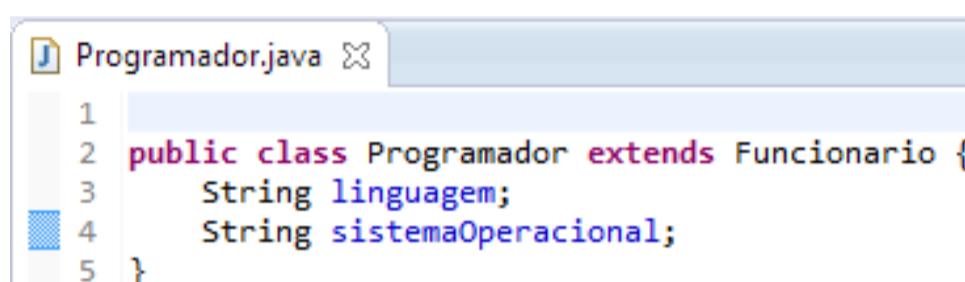
10.6. Estabelecendo herança entre classes

Para demonstrar a criação de herança entre classes, considere o exemplo apresentado a seguir, que inicia com a criação da classe **Funcionario**:

```
Funcionario.java
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 }
```

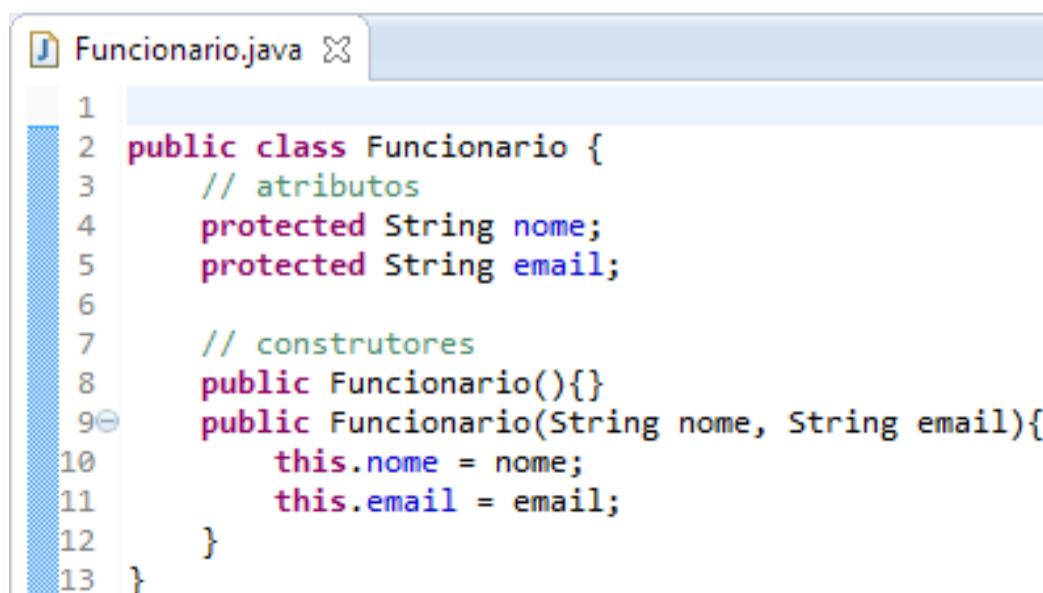
Repare que a classe **Programador** é uma extensão (também chamada de classe filha, derivada e subclasse) da classe **Funcionario** (também chamada de classe pai ou superclasse).

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**.

Lembre-se de que o 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 Applets, criação de componentes gráficos, multiprocessadores e outras mais.

A seguir, temos um exemplo de **Funcionario** e **Programador** utilizando o recurso de herança:



```
1
2 public class Funcionario {
3     // atributos
4     protected String nome;
5     protected 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 }
```

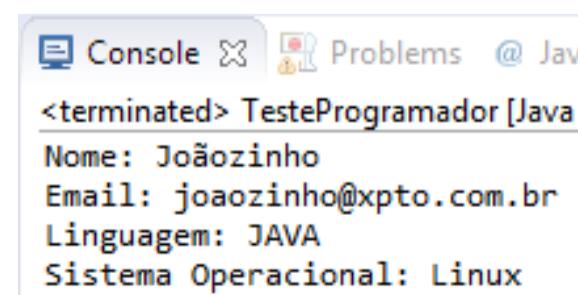
Java Programmer – Módulo I (online)

170

```
J Programador.java ✘
1
2 public class Programador extends Funcionario {
3     // atributos privados
4     private String linguagem;
5     private String sistemaOperacional;
6
7     // construtores
8     public Programador(){}
9     public Programador(String n, String e, String linguagem, String sistemaOperacional){
10         // Os atributos nome e email são HERDADOS da superclasse Funcionário
11         nome = n;
12         email = e;
13         this.linguagem = linguagem;
14         this.sistemaOperacional = sistemaOperacional;
15     }
16
17     // métodos getters e setters
18     public String getLinguagem() {
19         return linguagem;
20     }
21     public void setLinguagem(String linguagem) {
22         this.linguagem = linguagem;
23     }
24     public String getSistemaOperacional() {
25         return sistemaOperacional;
26     }
27     public void setSistemaOperacional(String sistemaOperacional) {
28         this.sistemaOperacional = sistemaOperacional;
29     }
30
31     public void imprimirDados(){
32         // Os atributos nome e email são HERDADOS da superclasse Funcionário
33         System.out.println("Nome: " + nome);
34         System.out.println("Email: " + email);
35         System.out.println("Linguagem: " + linguagem);
36         System.out.println("Sistema Operacional: " + sistemaOperacional);
37     }
38 }
```

```
J TesteProgramador.java ✘
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 }
```

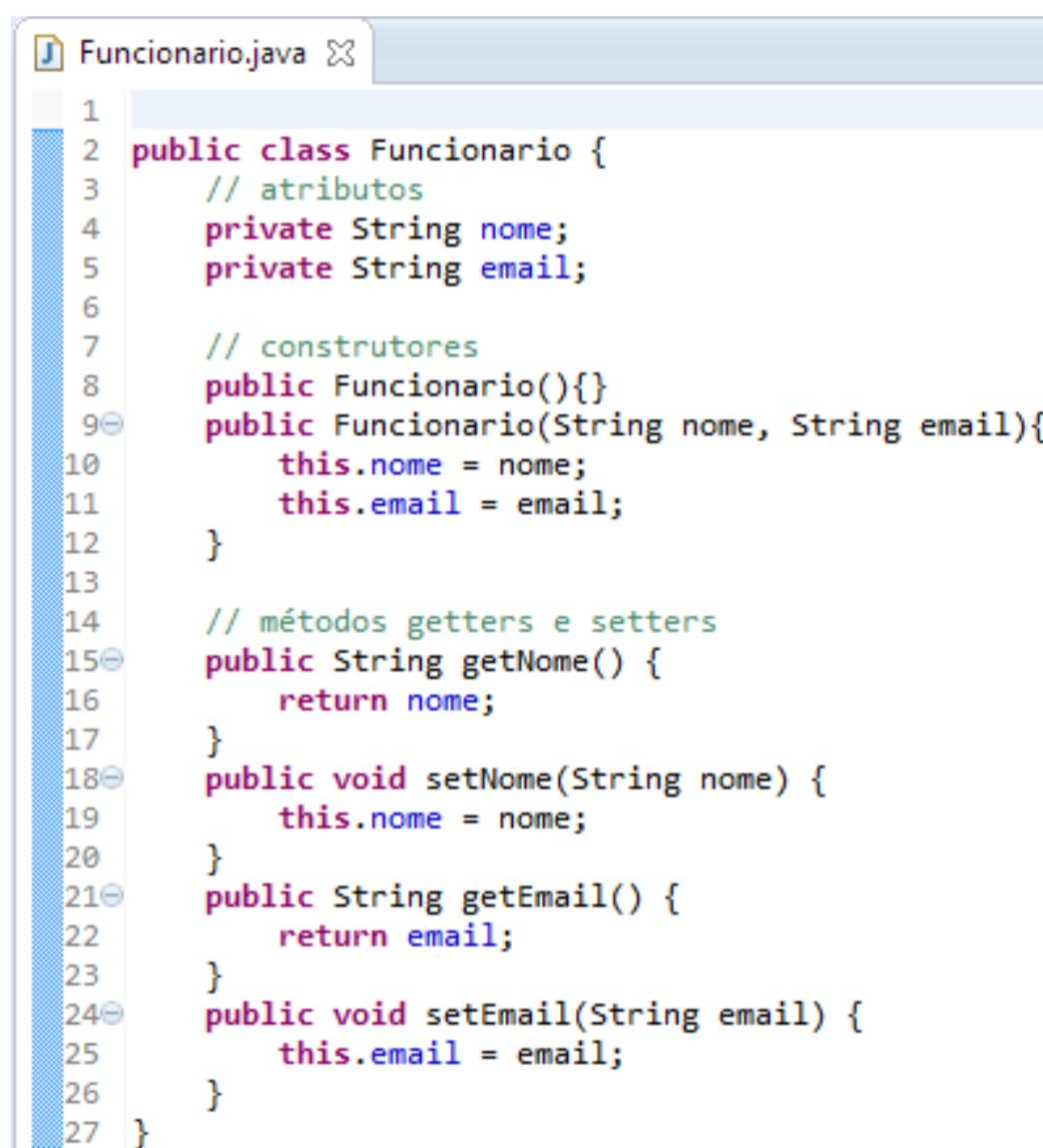
Após a compilação e a execução, o resultado será o seguinte:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:
<terminated> TesteProgramador [Java]
Nome: Joãozinho
Email: joaozinho@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Linux

10.6.1. 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**. Na verdade, membros privados sequer são herdados, uma vez que esse modificador impede a herança. Veja o acesso via métodos acessores, mostrado no exemplo a seguir:



The screenshot shows a Java code editor window titled "Funcionario.java". The code defines a class "Funcionario" with private attributes "nome" and "email", a no-argument constructor, and four accessor methods (getters and setters) for these attributes. The code is numbered from 1 to 27.

```
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          this.nome = nome;
10         this.email = email;
11     }
12
13     // métodos getters e setters
14     public String getNome() {
15         return nome;
16     }
17     public void setNome(String nome) {
18         this.nome = nome;
19     }
20     public String getEmail() {
21         return email;
22     }
23     public void setEmail(String email) {
24         this.email = email;
25     }
26 }
27 }
```

Java Programmer – Módulo I (online)

172

```
J Programador.java ✘
1
2 public class Programador extends Funcionario {
3     // atributos privados
4     private String linguagem;
5     private String sistemaOperacional;
6
7     // construtores
8     public Programador(){}
9     public Programador(String nome, String email, String linguagem, String sistemaOperacional){
10         // Os métodos setNome e setEmail são HERDADOS da superclasse Funcionário
11         setNome(nome);
12         setEmail(email);
13         this.linguagem = linguagem;
14         this.sistemaOperacional = sistemaOperacional;
15     }
16
17     // métodos getters e setters
18     public String getLinguagem() {
19         return linguagem;
20     }
21     public void setLinguagem(String linguagem) {
22         this.linguagem = linguagem;
23     }
24     public String getSistemaOperacional() {
25         return sistemaOperacional;
26     }
27     public void setSistemaOperacional(String sistemaOperacional) {
28         this.sistemaOperacional = sistemaOperacional;
29     }
30
31     public void imprimirDados(){
32         // Os métodos getNome e getEmail são HERDADOS da superclasse Funcionário
33         System.out.println("Nome: " + getNome());
34         System.out.println("Email: " + getEmail());
35         System.out.println("Linguagem: " + linguagem);
36         System.out.println("Sistema Operacional: " + sistemaOperacional);
37     }
38 }
```

```
J TesteProgramador.java ✘
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 }
```

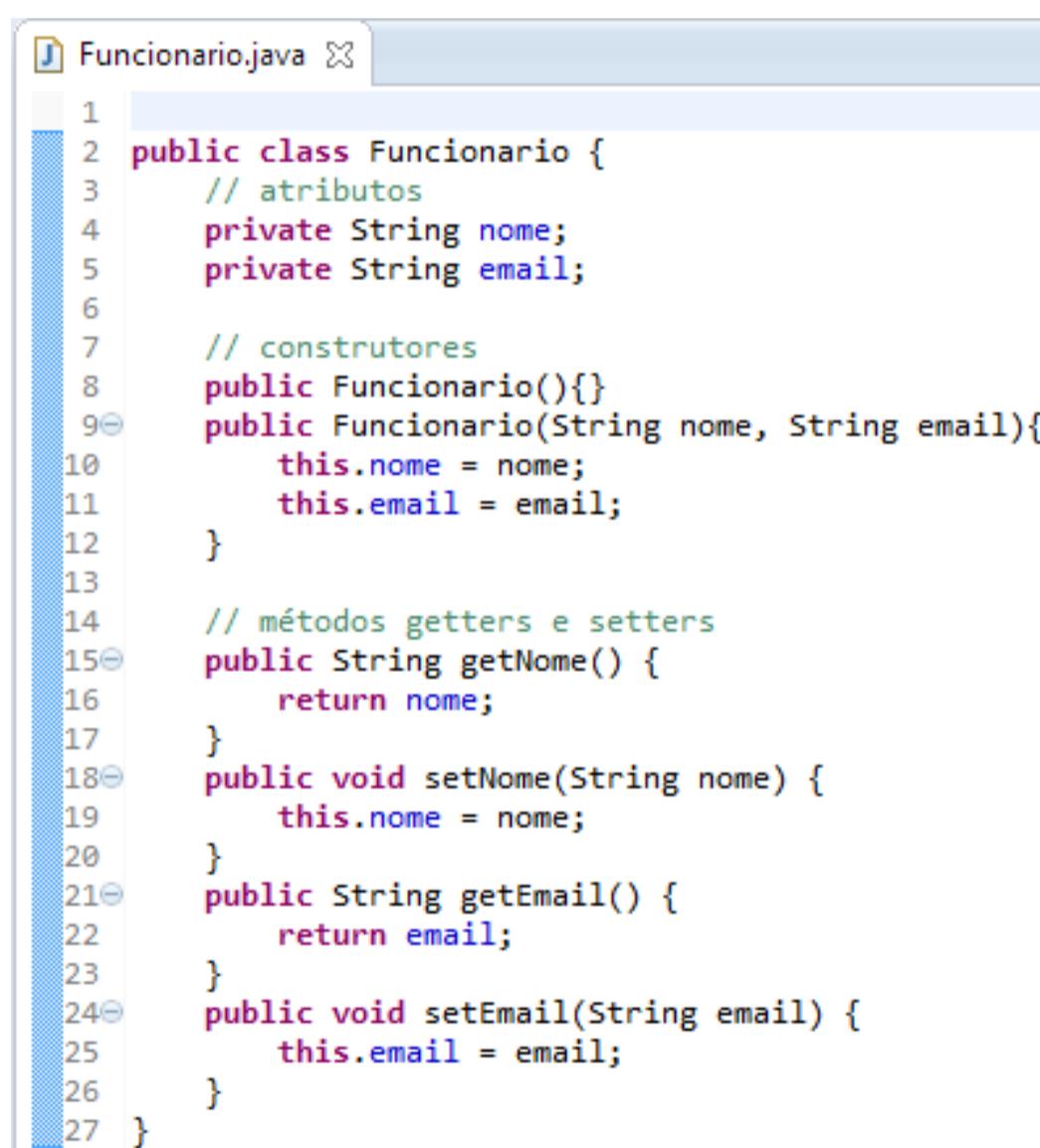
Após a compilação e a execução do código, o resultado será o seguinte:

```
Console ✘ Problems @ Jav
<terminated> TesteProgramador [Java]
Nome: Joãozinho
Email: joaozinho@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Linux
```

10.6.2. 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 **Funcionario**:



The screenshot shows a Java code editor window titled "Funcionario.java". The code defines a class "Funcionario" with private attributes "nome" and "email", two constructors, and four getter/setter methods for "nome" and "email". Lines 1 through 27 are numbered on the left. The code uses standard Java syntax with color-coded keywords and comments.

```
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          this.nome = nome;
10         this.email = email;
11     }
12
13     // métodos getters e setters
14     public String getNome() {
15         return nome;
16     }
17     public void setNome(String nome) {
18         this.nome = nome;
19     }
20     public String getEmail() {
21         return email;
22     }
23     public void setEmail(String email) {
24         this.email = email;
25     }
26 }
27 }
```

Java Programmer – Módulo I (online)

174

```
J Programador.java ✘
1
2 public class Programador extends Funcionario {
3     // atributos privados
4     private String linguagem;
5     private String sistemaOperacional;
6
7     // construtores
8     public Programador(){}
9     public Programador(String nome, String email, String linguagem, String sistemaOperacional){
10         // acessando os métodos setNome e setEmail com o super
11         super.setNome(nome);
12         super.setEmail(email);
13         this.linguagem = linguagem;
14         this.sistemaOperacional = sistemaOperacional;
15     }
16
17     // métodos
18     public String getLinguagem() {
19         return linguagem;
20     }
21     public void setLinguagem(String linguagem) {
22         this.linguagem = linguagem;
23     }
24     public String getSistemaOperacional() {
25         return sistemaOperacional;
26     }
27     public void setSistemaOperacional(String sistemaOperacional) {
28         this.sistemaOperacional = sistemaOperacional;
29     }
30
31     public void imprimirDados(){
32         // Os métodos getNome e getEmail são HERDADOS da superclasse Funcionário
33         System.out.println("Nome: " + super.getNome());
34         System.out.println("Email: " + super.getEmail());
35         System.out.println("Linguagem: " + linguagem);
36         System.out.println("Sistema Operacional: " + sistemaOperacional);
37     }
38 }
```

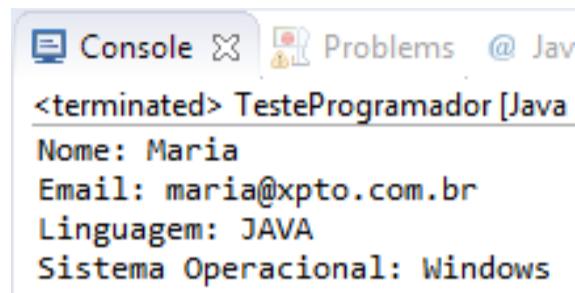
```
J TesteProgramador.java ✘
1
2 public class TesteProgramador {
3     public static void main(String args[]){
4         Programador junior = new Programador("Maria", "maria@xpto.com.br", "JAVA", "Windows");
5         junior.imprimirDados();
6     }
7 }
```

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 a 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
```

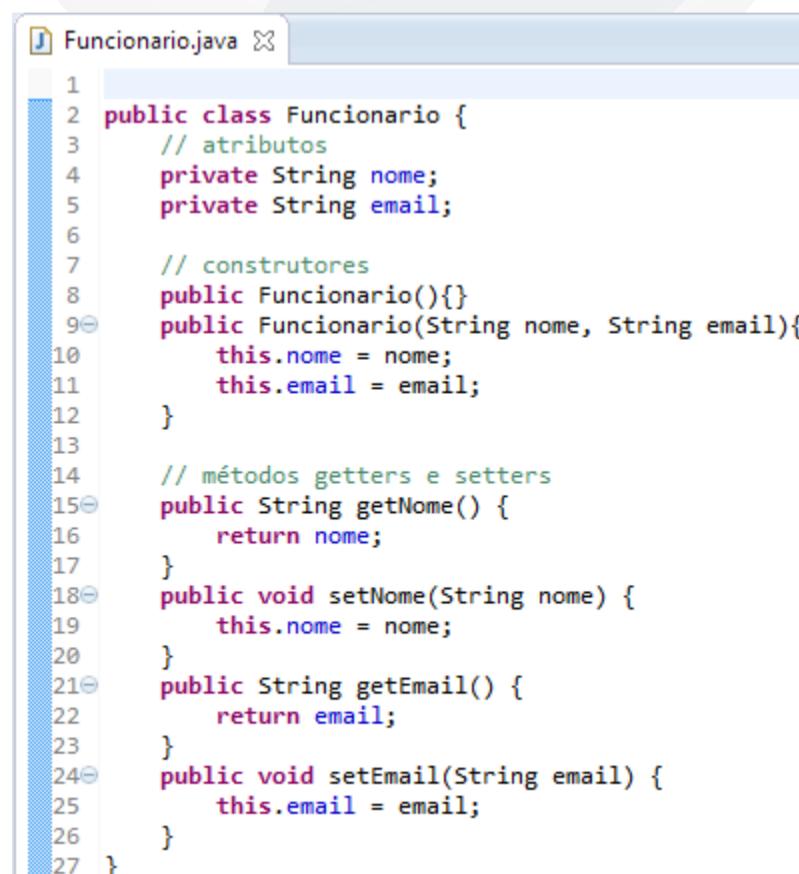
10.6.3. Chamada ao construtor da superclasse

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.

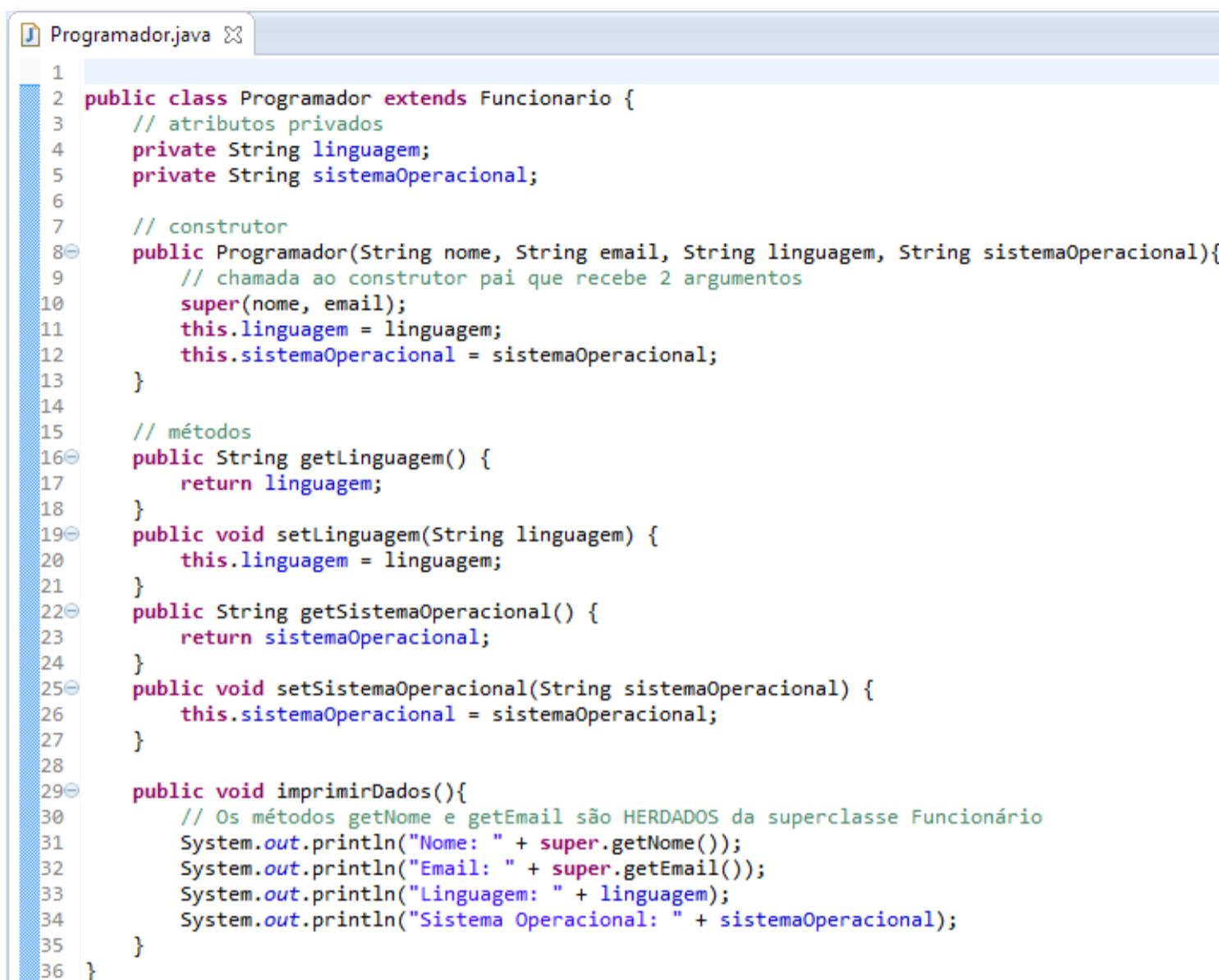
Veja, a seguir, como fazer a chamada ao construtor pai por meio do comando **super** dentro do construtor da subclasse:



```
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          this.nome = nome;
10         this.email = email;
11     }
12
13     // métodos getters e setters
14     public String getNome() {
15         return nome;
16     }
17     public void setNome(String nome) {
18         this.nome = nome;
19     }
20     public String getEmail() {
21         return email;
22     }
23     public void setEmail(String email) {
24         this.email = email;
25     }
26 }
27 }
```

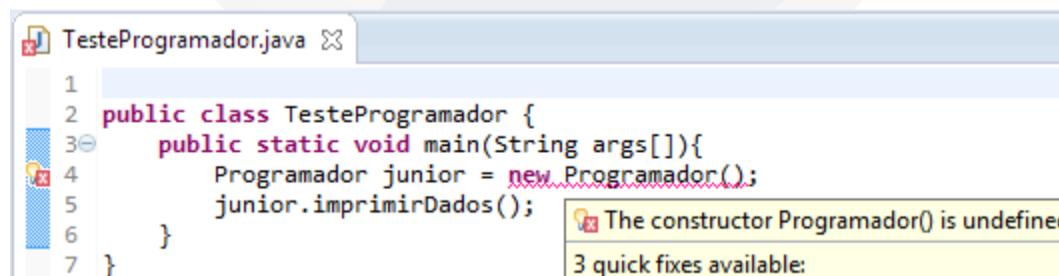
Java Programmer – Módulo I (online)

176



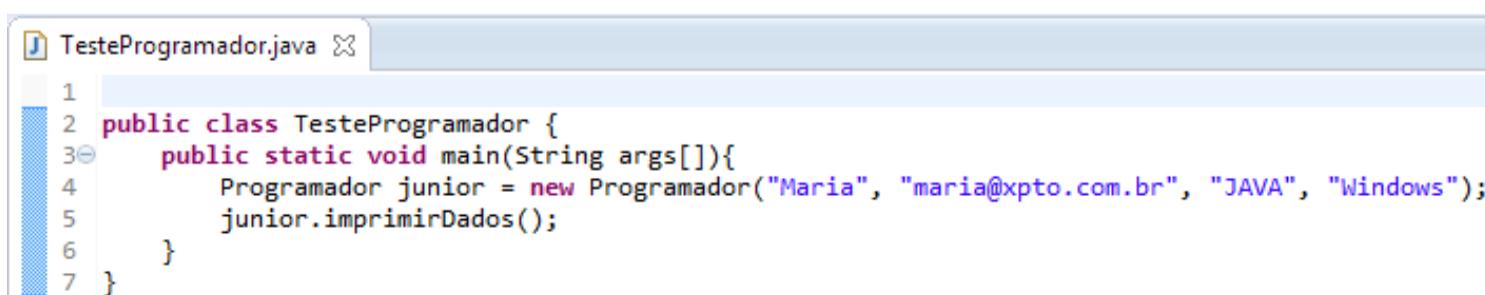
```
1
2 public class Programador extends Funcionario {
3     // atributos privados
4     private String linguagem;
5     private String sistemaOperacional;
6
7     // construtor
8     public Programador(String nome, String email, String linguagem, String sistemaOperacional){
9         // chamada ao construtor 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**:



```
1
2 public class TesteProgramador {
3     public static void main(String args[]){
4         Programador junior = new Programador();
5         junior.imprimirDados();
6     }
7 }
```

Agora, veja a forma correta de criar a instância:



```
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 a execução do código, o resultado será o seguinte:



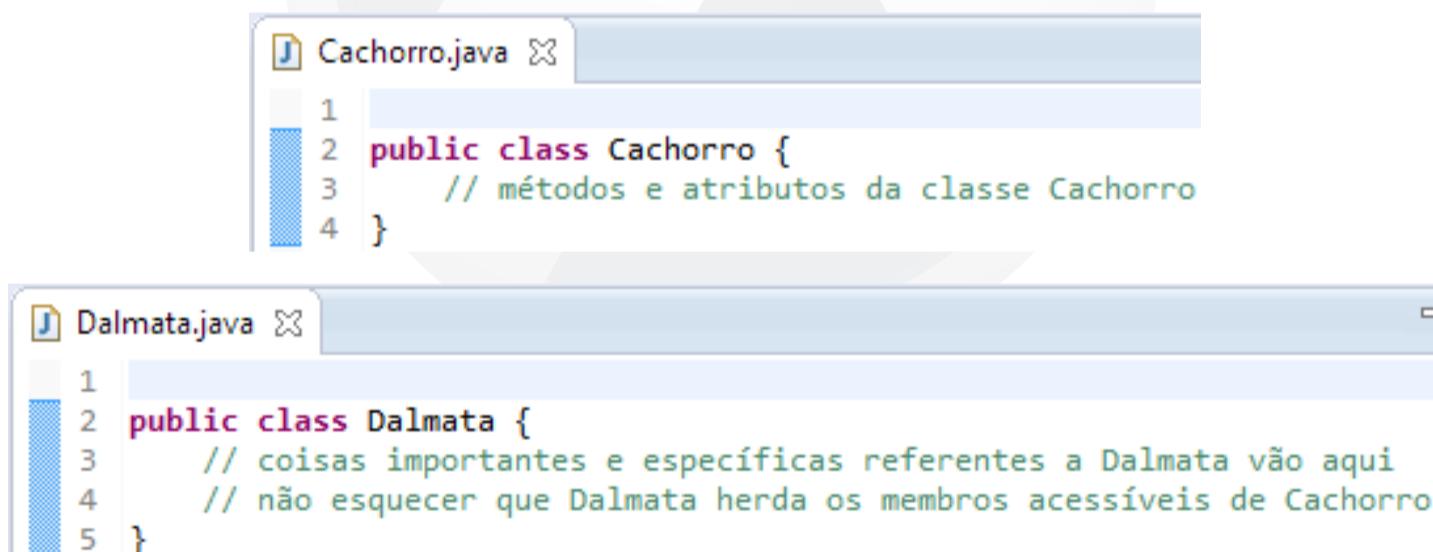
```
Console Problems @ Jav
<terminated> TesteProgramador [Java]
Nome: Maria
Email: maria@xpto.com.br
Linguagem: JAVA
Sistema Operacional: Windows
```

10.7. Relacionamentos

Veja, a seguir, dois tipos de relacionamentos que podemos usar na linguagem Java: o relacionamento baseado na herança e o baseado na utilização.

10.7.1. Relacionamento baseado na herança

Na programação orientada a objetos, utiliza-se o **é-um**, um tipo de relacionamento baseado na herança. Esse relacionamento é uma forma de classificar um item, por exemplo, “banana é-um fruto” ou “cachorro é-um animal”. Utilize a palavra-chave **extends** para expressar o relacionamento **é-um**:



```
Cachorro.java
1
2 public class Cachorro {
3     // métodos e atributos da classe Cachorro
4 }
```

```
Dalmata.java
1
2 public class Dalmata {
3     // coisas importantes e específicas referentes a Dalmata vão aqui
4     // não esquecer que Dalmata herda os membros acessíveis de Cachorro
5 }
```

Veja, a seguir, a ordem da árvore das classes:

```
public class Animais { ... }

public class Cachorro extends Animais { ... }

public class Dalmata extends Cachorro { ... }
```

Assim, de acordo com a terminologia utilizada pela programação orientada a objetos, é correto dizer que:

- **Animais** é a superclasse de **Cachorro**;
- **Cachorro** é a subclasse de **Animais**;
- **Cachorro** é a superclasse de **Dalmata**;
- **Dalmata** é a subclasse de **Animais**;
- **Cachorro** herda de **Animais**;
- **Dalmata** herda de **Cachorro**;
- **Dalmata** herda de **Animais**;
- **Dalmata** é derivado de **Cachorro**;
- **Cachorro** é derivado de **Animais**.

10.7.2. Relacionamento baseado na utilização (Composição)

Diferentemente do **é-um**, baseado na herança, o relacionamento **tem-um** baseia-se 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 public class Veiculo {
2     // métodos e atributos da classe Veiculo
3 }
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 a ela dentro da própria classe **Carro**.

Uma vantagem no uso de relacionamentos **tem-um** é que você pode projetar classes utilizando as técnicas de programação orientada a objetos sem o uso de classes monolíticas que realizam diversos tipos de tarefas.

Ao utilizar o relacionamento **tem-um**, é recomendável especializar ao máximo as classes, já que, quanto mais especializadas forem, maiores serão as chances de serem reaproveitadas por outros aplicativos. Na especialização, o código de **ConjuntoDePneus** é mantido em uma classe reservada e especializada.

Se você inserir todos os códigos ligados à classe **ConjuntoDePneus** dentro da classe **Carro**, o código **ConjuntoDePneus** será duplicado por todas as outras classes que podem utilizar seu comportamento.

A classe **Carro** contém uma referência à classe **ConjuntoDePneus** pelo fato de a primeira declarar uma variável de instância do tipo **ConjuntoDePneus**. Assim, toda vez que o código chamar **rodar()** em uma instância de **Carro**, esta chamará o método na variável de instância **ConjuntoDePneus** do objeto **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  public class Carro {
2      private ConjuntoDePneus meuConjuntoDePneus;
3
4      public void rodar(){
5          // delega o comportamento de rodar para o objeto ConjuntoDePneus
6          meuConjuntoDePneus.rodar();
7      }
8  }
```

```
ConjuntoDePneus.java
1  public class ConjuntoDePneus {
2      public void rodar(){
3          // Realiza a tarefa atual de rodar aqui
4      }
5  }
```

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.

10.8. 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.

10.8.1. 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 da classe **Object**, os objetos de todas as classes têm as definições da classe em questão disponíveis.

10.8.2. Classes abstratas

Classes abstratas são aquelas a partir das quais não é possível realizar qualquer tipo de instância. Além disso, classes abstratas podem conter membros que não são abstratos, como métodos e variáveis normais. Entretanto, em sua declaração, essas classes deverão ter, preferencialmente, um ou mais métodos abstratos. De forma inversa, caso uma classe contenha ao menos um método abstrato, deverá ser declarada obrigatoriamente abstrata.

Uma vez que os métodos abstratos não contêm corpo e, tampouco, implementação, a classe que tem a função de estender a classe abstrata é responsável pela implementação desses métodos abstratos.

Como medida de segurança, as classes abstratas somente podem ser estendidas 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.

A funcionalidade dos métodos abstratos herdados pelas subclasses normalmente é atribuída de acordo com o objetivo ou o propósito dessas classes. Porém, é possível não atribuir uma funcionalidade a esses métodos abstratos. Nesse caso, faz-se necessário, pelo menos, declarar tais métodos, ainda que com corpo vazio.

10.8.2.1. 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 desse tipo de método é a seguinte:

```
<modificadores> abstract <tipoRetorno> <nomeMetodo>(listaParametros);
```

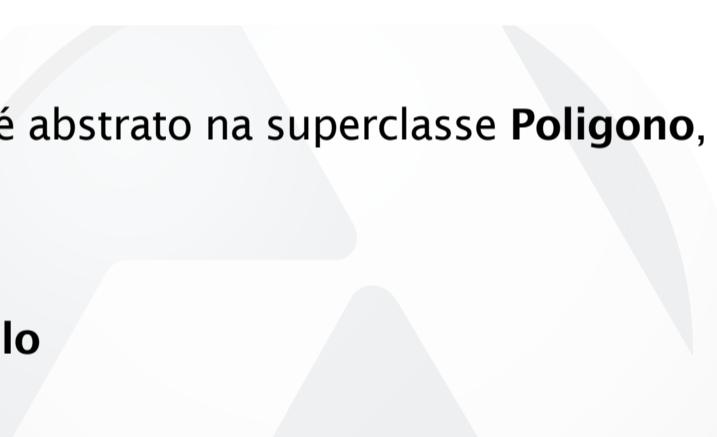
No exemplo a seguir, a classe **Polígono** será abstrata e terá duas subclasses concretas, **Quadrado** e **Triângulo**, as quais são obrigadas a implementar o método abstrato **calcularArea()**. Observe:

```
1  public abstract class Poligono {  
2      public abstract double calcularArea();  
3  }
```

! A palavra **abstract** como prefixo da declaração da classe **Polígono** indica que essa 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 **Triângulo**. Veja, então, como definir as classes **Quadrado** e **Triângulo**:

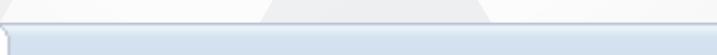
- Definindo a classe Quadrado



```
J Quadrado.java ✘
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}
```

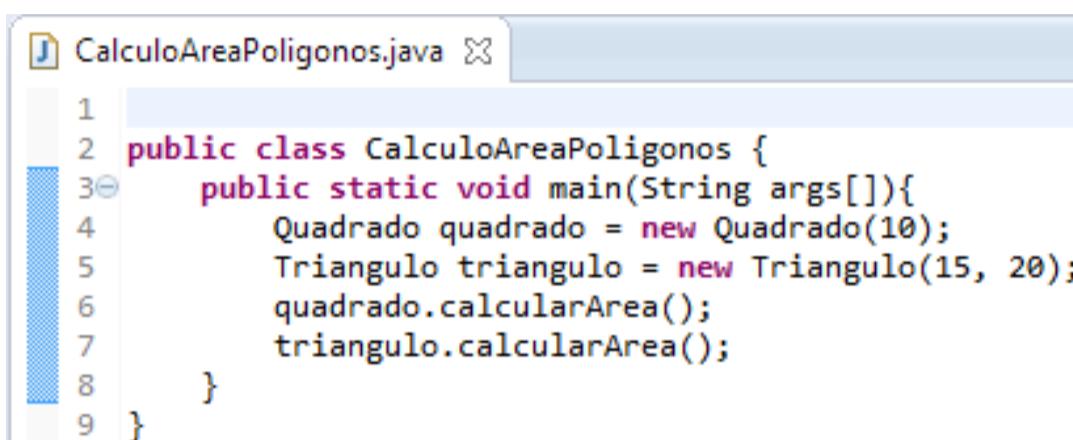
O método **calcularArea()**, que é abstrato na superclasse **Poligono**, é implementado na classe **Quadrado**.

- Definindo a classe Triangulo



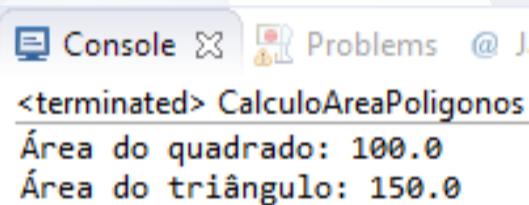
```
J Triangulo.java ✘
1
2 public class Triangulo extends Poligono {
3     private double base;
4     private double altura;
5
6     public Triangulo(double base, double altura){
7         this.base = base;
8         this.altura = altura;
9     }
10
11    public double calcularArea(){
12        double 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 }
9 }
```

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.

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.

10.8.3. Classes finais

As classes que são declaradas com a palavra-chave **final**, as chamadas classes finais, não podem ser estendidas por outras classes. Sendo assim, as classes finais devem ser criadas somente quando você deseja assegurar que nenhum de seus métodos seja sobreescrito. Além disso, a utilização da palavra-chave **final** assegura que a implementação de determinados métodos não será alterada.

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.

Semanticamente, o modificador **final** se opõe ao modificador **abstract**. Podem ser considerados logicamente opostos e, por esse motivo, jamais podem ser usados em conjunto.

Os exemplos descritos a seguir mostram como declarar a classe final **Animal** e, em seguida, o que acontece se você tentar compilar a subclasse **Gato**. Veja:

```
1 public final class Animal {  
2     public void comer(){  
3         System.out.println("Animal comendo...");  
4     }  
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**.

```
1  
2 public class Gato extends Animal {  
3  
4 }
```

The type Gato cannot subclass the final class Animal

Console Problems Javadoc Declaration
1 error, 0 warnings, 0 others
Description
Errors (1 item)
The type Gato cannot subclass the final class Animal

Por não poder estender classes finais, caso fosse necessário alterar o comportamento em um de seus métodos e você não tivesse o código-fonte, este não poderia ser sobrescrito. Entretanto, em situações como esta, as classes que não são finais podem ser estendidas, fazendo com que o método possa ser redefinido em uma subclasse, o que solucionaria os problemas apresentados.

Um grande exemplo de classe final em Java é a classe **String**, já vista em exercícios anteriores e amplamente utilizada por desenvolvedores Java.

10.9. 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 sobreSCRIÇÃO 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.

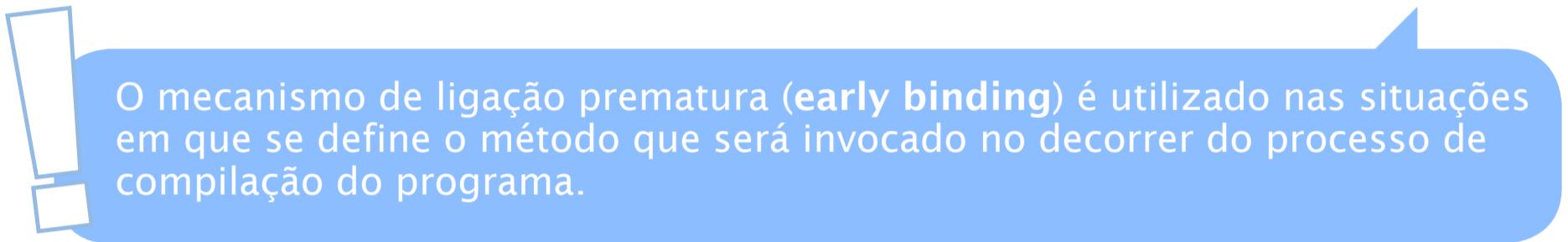
Apesar de o mecanismo em questão ser capaz de simplificar não somente a compreensão, mas também o desenvolvimento do código orientado a objetos, em algumas circunstâncias, é possível que o resultado da execução seja não intuitivo.

10.9.1. 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.

10.9.2. 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, pois, caso contrário, haverá uma sobrecarga.

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 sobreSCRIÇÃO 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

```
graph TD; Animal --> Zebra; Animal --> Leao; Zebra --> Zoologico; Leao --> Zoologico;
```

The diagram illustrates the inheritance hierarchy. The `Animal` class is the base class, with `Zebra` and `Leao` as derived classes. Both `Zebra` and `Leao` inherit from `Animal`. The `Zoologico` class is a client that uses objects of type `Animal`, `Zebra`, and `Leao`.

Animal.java

```
1
2 public class Animal {
3     public void comer(){
4         System.out.println("Animal comendo...");
5     }
6 }
```

Zebra.java

```
1
2 public class Zebra extends Animal {
3     public void comer(){
4         System.out.println("Zebra comendo...");
5     }
6 }
```

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 a 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 passa 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 a execução do exemplo anterior, o resultado será como o indicado na imagem a seguir:

```
Console X Proj
<terminated> Zoologico
Animal comendo...
Zebra comendo...
Leão comendo...
```

- **Exemplo 2**

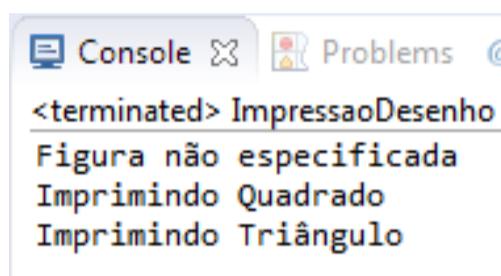
```
Desenho.java
1
2 public class Desenho {
3     public void imprimir(){
4         System.out.println("Figura não especificada");
5     }
6 }

Quadrado.java
1
2 public class Quadrado extends Desenho {
3     public void imprimir(){
4         System.out.println("Imprimindo Quadrado");
5     }
6 }

Triangulo.java
1
2 public class Triangulo extends Desenho {
3     public void imprimir(){
4         System.out.println("Imprimindo Triângulo");
5     }
6 }

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 a execução desse exemplo, o resultado será como o mostrado na imagem a seguir:



```
Console X Problems 
<terminated> ImpressaoDesenho
Figura não especificada
Imprimindo Quadrado
Imprimindo Triângulo
```

10.9.3. 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.

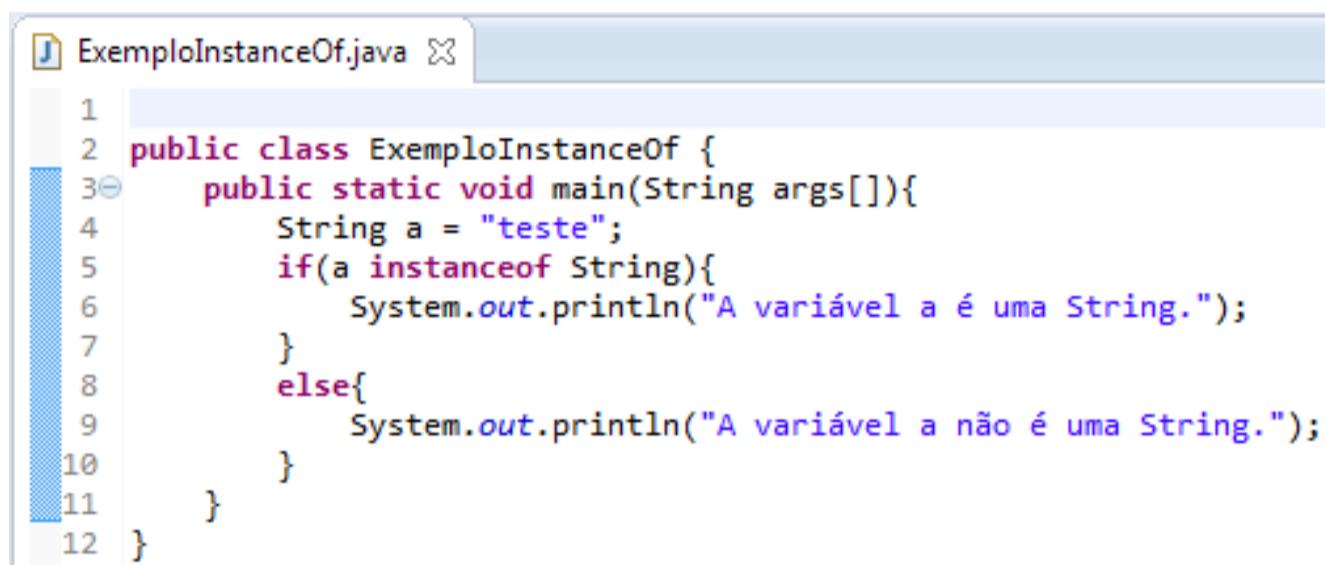
Para testarmos a referência a um objeto, podemos confrontá-la com o tipo de classe ou, então, com uma das superclasses. A partir disso, utilizando o **instanceof** com o tipo **Object**, temos que qualquer referência a um objeto é avaliada como verdadeira, pois a classe **Object** é a superclasse de todas as classes em Java.

Caso uma das superclasses implemente a interface, o objeto é considerado um tipo de interface específico.

Quando uma superclasse de objeto implementa uma interface, porém, a classe da instância não o implementa, temos uma implementação indireta. Se isso ocorrer, recomendamos que sejam realizadas perguntas com o operador **instanceof** para testar se o objeto é realmente instância de uma classe que implementa uma interface de forma direta ou indireta.

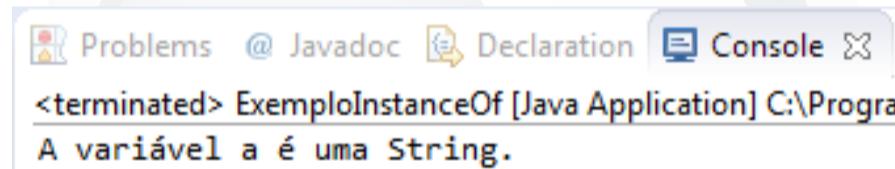
Adicionalmente, recomendamos testar se um objeto nulo refere-se à instância de uma classe. O resultado, neste caso, será sempre falso.

Veja um exemplo:



```
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 <terminated> ExemploInstanceOf [Java Application] C:\Progra
A variável a é uma String.
```



Teste seus conhecimentos
Herança

10

1. 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 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.

Interfaces

11

- ✓ O conceito de interface;
- ✓ Variáveis de referência;
- ✓ Variáveis inicializadas;
- ✓ Métodos estáticos;
- ✓ Métodos default.

11.1. 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 ou default e constantes estáticas (tipos **final static**). Por meio das interfaces, você pode especificar as funcionalidades a serem definidas pelas classes, 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 servir 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. Isso significa que não possuem raízes ascendentes finais.

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:

The screenshot shows two Java code files in a code editor:

- Eletrodomestico.java:**

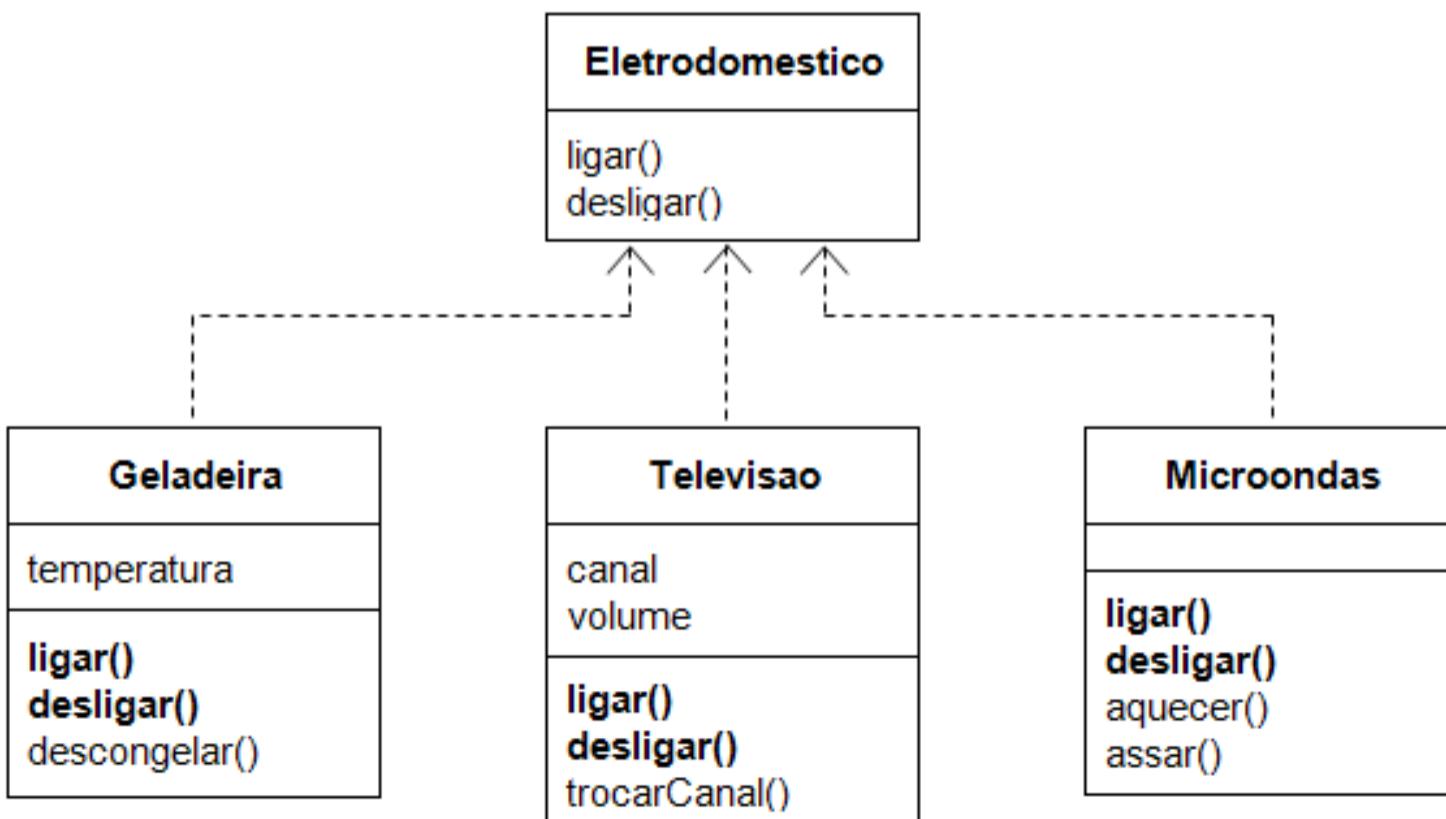
```
1 | public interface Eletrodomestico {  
2 |     void ligar();  
3 |     void desligar();  
4 | }  
5 |  
6 |
```
- Geladeira.java:**

```
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 |
```

11.2. 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.

11.3. Variáveis inicializadas

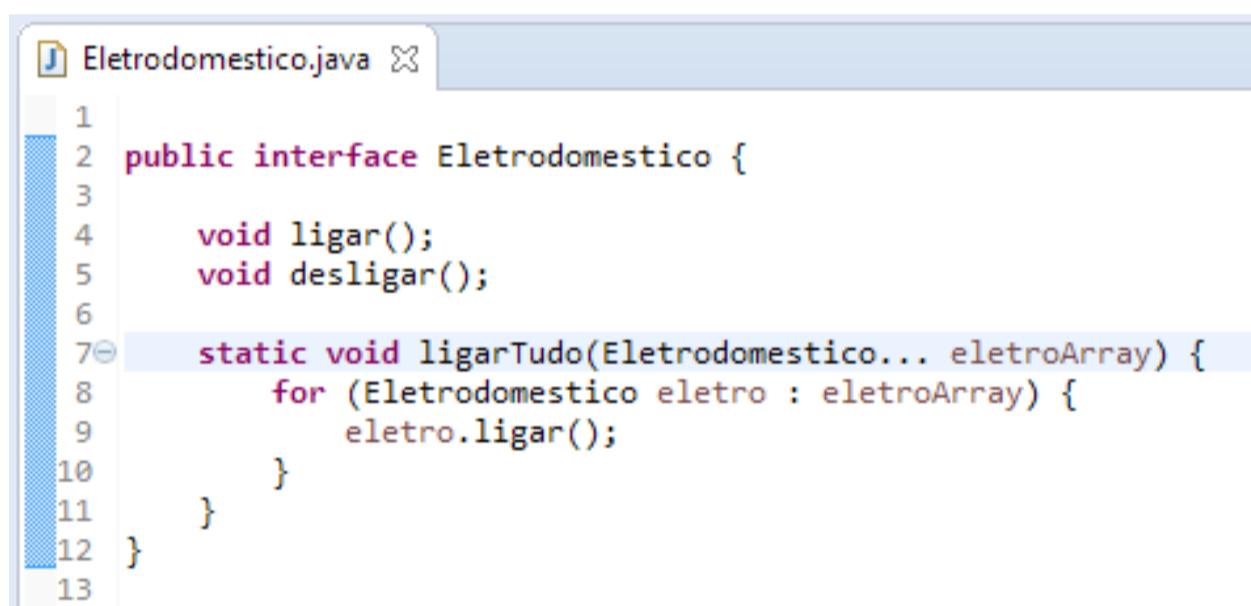
Uma interface pode conter variáveis inicializadas, as quais funcionam como constantes, porém, ela não contém variáveis de instância. Em razão disso, as variáveis que funcionam como constantes são compartilhadas pelas classes que implementam esta interface.

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.

11.4. Métodos estáticos

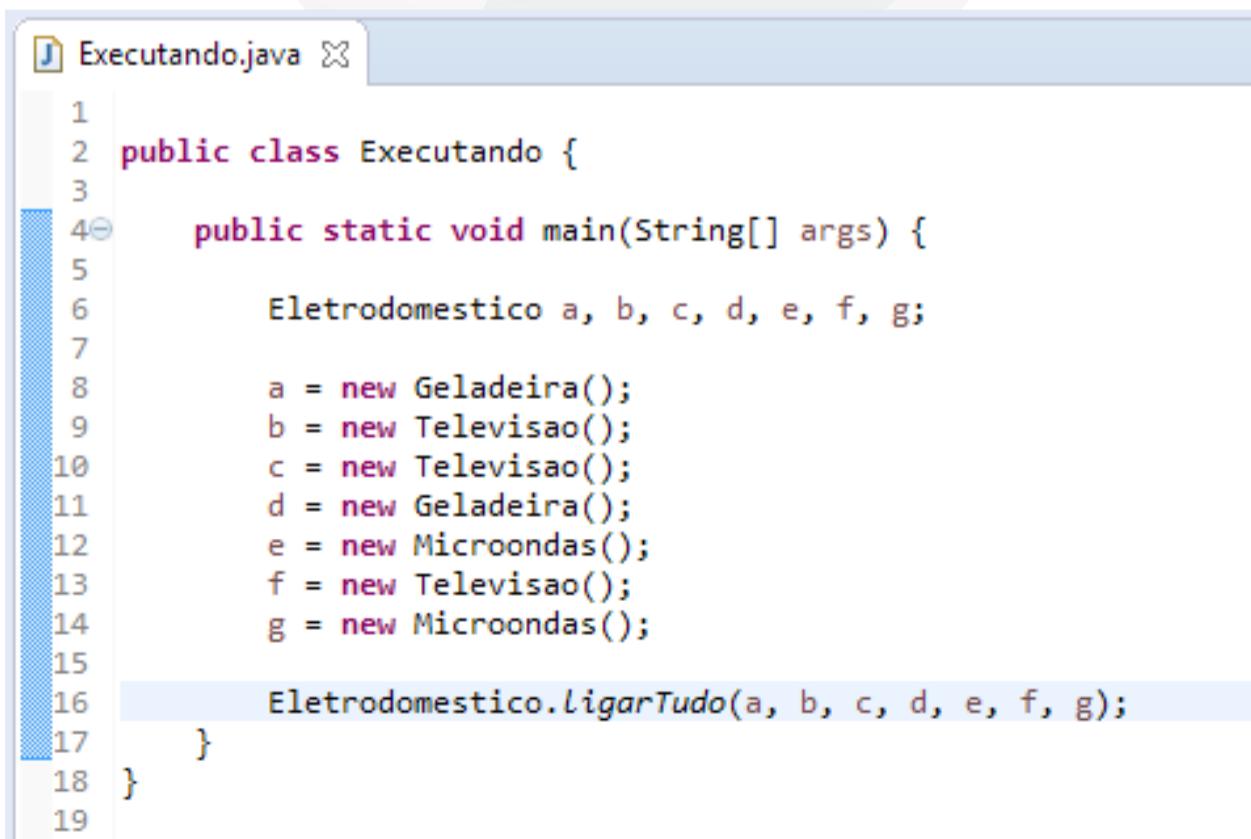
Umas das inovações da versão 8 do Java é a capacidade de implementar métodos estáticos dentro de uma interface. O objetivo desse novo recurso é permitir que sejam criados, na própria interface, métodos de tratamento para entidades que implementam a própria interface.

Observe:



```
1
2  public interface Eletrodomestico {
3
4      void ligar();
5      void desligar();
6
7      static void ligarTudo(Eletrodomestico... electroArray) {
8          for (Eletrodomestico eletr : electroArray) {
9              eletr.ligar();
10         }
11     }
12 }
13
```

Por serem estáticos, estes métodos são chamados a partir do nome da interface, sem a necessidade de nenhuma variável de referência:



```
1
2  public class Executando {
3
4      public static void main(String[] args) {
5
6          Eletrodomestico a, b, c, d, e, f, g;
7
8          a = new Geladeira();
9          b = new Televisao();
10         c = new Televisao();
11         d = new Geladeira();
12         e = new Microondas();
13         f = new Televisao();
14         g = new Microondas();
15
16         Eletrodomestico.LigarTudo(a, b, c, d, e, f, g);
17     }
18 }
19
```

11.5. Métodos default

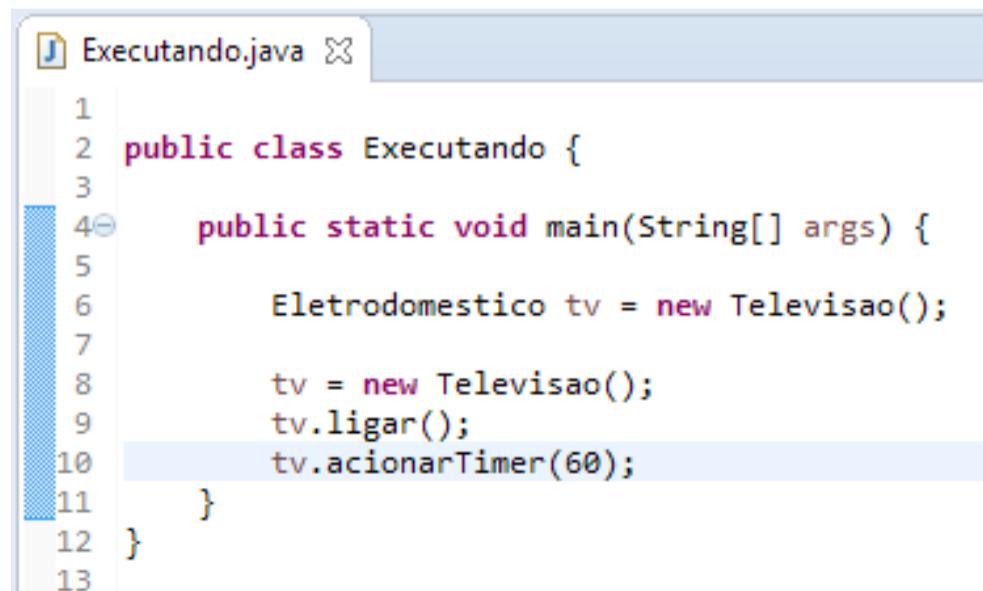
Outra inovação interessante no Java 8 é o método **default** (padrão). Trata-se de um método comum, criado dentro da interface, que permite criar um comportamento padrão ou implementação padrão para todas as classes que implementam aquela interface.

Observe:



```
1  public interface Eletrodomestico {
2
3      void ligar();
4
5      void desligar();
6
7
8      default void acionarTimer(int minutos) {
9
10         /* Aguarda a quantidade de minutos informada. */
11         try { Thread.sleep(minutos * 60000); } catch (Exception e) {}
12
13         /* Executa o método desligar conforme implementado na classe. */
14         desligar();
15     }
16 }
17
```

Este método pode ser normalmente utilizado por todos os objetos daquele tipo, sem que a classe implementadora possua tal método:



```
1  public class Executando {
2
3
4      public static void main(String[] args) {
5
6          Eletrodomestico tv = new Televisao();
7
8          tv = new Televisao();
9          tv.ligar();
10         tv.acionarTimer(60);
11     }
12 }
13
```



**Teste seus conhecimentos
Interfaces**

11

1. Qual das alternativas a seguir não está correta?

- 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. Isso significa que possuem raízes ascendentes finais.
- e) Nenhuma das alternativas anteriores está correta.

2. Para que serve uma variável de referência?

- a) Para referenciar uma instância de qualquer classe do programa.
- b) Para armazenar uma instância de qualquer classe que implemente a interface.
- c) Para referenciar uma classe que não tenha interface.
- d) Para armazenar um objeto.
- e) Nenhuma das alternativas anteriores está correta.

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.

