



THIAGO FARIA E NORMANDES JR

# JAVA

## E ORIENTAÇÃO A OBJETOS

*Presente de Natal  
para nossos queridos  
alunos e seguidores.*

*Dezembro/2014*





# Java e Orientação a Objetos

## por Thiago Faria e Normandes Junior

**1ª Edição, 24/12/2014**

© 2014 AlgaWorks Softwares, Treinamentos e Serviços Ltda. Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da AlgaWorks, exceto para resumos breves em revisões e análises.

AlgaWorks Softwares, Treinamentos e Serviços Ltda  
[www.algaworks.com](http://www.algaworks.com)  
[contato@algaworks.com](mailto:contato@algaworks.com)

# CURSOS ONLINE

## COM VÍDEO AULAS E SUPORTE

---



Java e Orientação a Objetos



Desenvolvimento Web  
com JSF 2



Persistência de Dados com  
JPA 2 e Hibernate



Sistemas Comerciais Java EE  
com CDI, JPA e PrimeFaces

[www.algaworks.com](http://www.algaworks.com)

Cursos presenciais in-company? Entre em contato.

# Sobre os autores



**Thiago Faria de Andrade**

[@ThiagoFAndrade](#)

Fundador e instrutor da AlgaWorks. Graduado em Sistemas de Informação e certificado como programador Java pela Sun. Iniciou seu interesse por programação em 1995, quando desenvolveu um software para entretenimento e se tornou um dos mais populares no Brasil e outros países de língua portuguesa. Já foi sócio e trabalhou em outras empresas de software como programador, gerente e diretor de tecnologia, mas nunca deixou de programar.



**Normandes José Moreira Junior**

[@Normandesjr](#)

Sócio e instrutor da AlgaWorks, formado em Engenharia Elétrica pela Universidade Federal de Uberlândia e detentor das certificações LPIC-1, SCJP e SCWCD. Palestrante internacional reconhecido por contribuir e liderar projetos open source.



# Sumário

1	Como aprender Java? . . . . .	13
1.1	Faça cursos, leia muito e pratique . . . . .	13
1.2	Acompanhe blogs e acesse fóruns de discussão . . . . .	13
1.3	Não menospreze os exemplos . . . . .	14
1.4	Não tenha vergonha em perguntar . . . . .	14
1.5	Desconecte-se dos aplicativos de mensagens instantâneas . . . . .	14
1.6	Concentre-se . . . . .	14
1.7	Não se assuste com a sopa de letrinhas . . . . .	15
2	A história do Java . . . . .	16
3	As plataformas Java . . . . .	18
3.1	A plataforma Java SE . . . . .	18
3.2	A plataforma Java EE . . . . .	19
3.3	A plataforma Java ME . . . . .	20
3.4	Java Card, JavaFX e Java TV . . . . .	20
3.5	Como Java evolui? . . . . .	20
4	Máquina virtual Java . . . . .	22
4.1	O que é JVM? . . . . .	22
4.2	A JVM faz o Java ficar lento? . . . . .	25
5	Baixando, instalando e configurando . . . . .	26
5.1	Preciso do JDK ou JRE? . . . . .	26
5.2	Baixando o JDK da Oracle . . . . .	26
5.3	Instalando o JDK no Windows . . . . .	28
5.4	Instalando o JDK no Linux . . . . .	31
6	Fundamentos da linguagem . . . . .	33
6.1	Vamos instalar a IDE agora? . . . . .	33
6.2	Codificando o programa "oi mundo" . . . . .	33

6.3	Compilando e executando	34
6.4	Entendendo o que foi codificado	36
6.5	Erros comuns dos marinheiros de primeira viagem	37
6.6	Comentários	39
6.7	Sequências de escape	39
6.8	Palavras reservadas	41
6.9	Convenções de código	41
6.10	Trabalhando com variáveis	42
6.11	Nomeando variáveis	44
6.12	Operadores aritméticos	45
6.13	Tipos primitivos	46
6.14	Outros operadores de atribuição	49
6.15	Conversão de tipos primitivos	50
6.16	Promoção aritmética	54
6.17	Trabalhando com strings	56
6.18	Recebendo entrada de dados	57
6.19	Operadores de comparação e igualdade	60
6.20	Estruturas de controle if, else if e else	61
6.21	Programar ifs sem abrir e fechar blocos é legal?	63
6.22	Escopo de variáveis	65
6.23	Operadores lógicos	66
6.24	Estrutura de controle switch	69
6.25	Operador ternário	72
6.26	Operadores de incremento e decremento	73
6.27	Estrutura de controle while	74
6.28	Estrutura de controle do/while	75
6.29	Estrutura de controle for	76
6.30	Cláusulas break e continue	77



7	Use Eclipse e seja feliz! . . . . .	79
7.1	Introdução ao Eclipse IDE . . . . .	79
7.2	Baixando e instalando o Eclipse . . . . .	80
7.3	Iniciando o Eclipse . . . . .	81
7.4	Ambientes de trabalho . . . . .	81
7.5	Criando o primeiro projeto . . . . .	82
7.6	Compilando e executando o programa . . . . .	84
7.7	Outras teclas de atalho . . . . .	84
7.8	Depurando códigos . . . . .	86
8	Introdução a orientação a objetos . . . . .	89
8.1	O que é POO? . . . . .	89
8.2	Classes e objetos . . . . .	90
8.3	Criando uma classe com atributos . . . . .	94
8.4	Instanciando objetos . . . . .	95
8.5	Acessando atributos de objetos . . . . .	98
8.6	Composição de objetos . . . . .	100
8.7	Valores padrão . . . . .	104
8.8	Variáveis referenciam objetos . . . . .	106
8.9	Criando e chamando métodos . . . . .	109
8.10	Nomeando métodos . . . . .	112
8.11	Métodos com retorno . . . . .	113
8.12	Passando argumentos para métodos . . . . .	116
8.13	Argumentos por valor ou por referência . . . . .	117
8.14	Métodos que alteram variáveis de instância . . . . .	120
8.15	O objeto this . . . . .	121
8.16	Sobrecarga de métodos . . . . .	122
8.17	Crie bons métodos . . . . .	124
8.18	Coletor de lixo . . . . .	124

9	Wrappers e boxing	126
9.1	O que são classes wrapper?	126
9.2	Usando classes wrappers	127
9.3	Métodos de conversão	128
9.4	Autoboxing do Java 5	129
10	Trabalhando com arrays	130
10.1	Criando e acessando arrays	130
10.2	Iterando em arrays	132
10.3	Expandindo arrays	133
10.4	Arrays de objetos	134
10.5	Arrays multidimensionais	135
10.6	Lendo os parâmetros da linha de comando	136
10.7	Métodos com argumentos variáveis	137
11	Diagrama de classes, construtores e encapsulamento	139
11.1	Introdução ao diagrama de classes	139
11.2	Construtores	140
11.3	Modificadores de acesso public e private	143
11.4	Encapsulamento	147
11.5	JavaBeans	149
12	Pacotes, outros modificadores e enumerações	151
12.1	Organizando os projetos em pacotes	151
12.2	Modificador de acesso default	156
12.3	Membros de classe	158
12.4	Modificador final e constantes	161
12.5	Enumerações	163
13	Orientação a objetos avançada	166
13.1	Herança e sobrescrita	166
13.2	Modificador de acesso protected	174

13.3	Polimorfismo	176
13.4	Casting de objetos e instanceof	180
13.5	Classes e métodos abstratos	182
13.6	Interfaces	187
13.7	Tratando e lançando exceções	195
13.8	Lançamento de exceções	199
14	Mais sobre a API Java	203
14.1	Classe java.lang.Math	203
14.2	Static import	207
14.3	Classes String, StringBuffer e StringBuilder	208
14.4	Trabalhando com datas	215
14.5	Trabalhando com números	219
15	Collections Framework	225
15.1	Introdução a coleções	225
15.2	Listas - Interface List	226
15.3	Conjuntos - Interface Set	237
15.4	O equals() e o hashCode()	239
15.5	Mapas - Interface Map	244
15.6	Conjunto ordenado - Interface SortedSet	246
15.7	Mapa ordenado - Interface SortedMap	248
15.8	Algoritmos - Classe Collections	250
16	Entrada e saída	254
16.1	O que é I/O Streams?	254
16.2	Gravando arquivos	255
16.3	Lendo arquivos	257
16.4	A classe Scanner	258
17	Arquivos JAR e documentação com javadoc	261
17.1	O que são os arquivos JAR?	261

17.2	Gerando arquivos JAR executáveis	261
17.3	Gerando arquivos JAR como bibliotecas	263
17.4	Importando arquivos JAR no projeto	264
17.5	Gerando documentação com javadoc	266

# Como aprender Java?

## 1.1. Faça cursos, leia muito e pratique

Você está no caminho certo para aprender Java, e quem sabe se tornar um especialista nesta tecnologia. Fazer cursos, ler bons livros e apostilas são excelentes fontes para aprender qualquer tecnologia, mas só isso provavelmente não será suficiente.

Aprender qualquer nova tecnologia exige dedicação, concentração e muita prática. Por isso, ao estudar cada assunto deste livro, reproduza os exemplos e faça diversos exercícios. Mesmo que você ache o exemplo simples e entenda perfeitamente sem praticar, não deixe de escrever o código você mesmo e testar, pois a repetição ajudará bastante a fixar a sintaxe da linguagem e os conceitos realmente importantes.

## 1.2. Acompanhe blogs e acesse fóruns de discussão

Existem vários blogs que falam de desenvolvimento Java. Um exemplo é o [blog da AlgaWorks](#). Acompanhe os que você mais gosta e confia!

Sempre que tiver alguma dúvida, pesquise no Google. Provavelmente, você não é o único com as mesmas dificuldades ou problemas. Se não encontrar uma resposta, acesse fóruns de discussão ou sites de perguntas e respostas, como o [GUJ](#) e [Stack Overflow](#).

Explique bem a sua dúvida, inclua trechos de código que você fez e não funcionou. Nunca pergunte algo sem mostrar o que você já tentou fazer, e nunca, jamais peça uma solução pronta para seu problema. Muitas pessoas estão dispostas a ajudar você, mas a maioria não vai querer trabalhar de graça para você. :)

### 1.3. Não menospreze os exemplos

Se você já tiver algum conhecimento em Java ou outras linguagens que possuam a sintaxe parecida, talvez você ache os primeiros exemplos desse livro simples e queira deixar de fazê-los. A dica é: não menospreze os exemplos! Por mais simples que possam parecer para você, é importante que você os faça para descobrir as diferenças, encontrar problemas (e resolvê-los), exercitar sua criatividade, compilar e executar os códigos.

### 1.4. Não tenha vergonha em perguntar

Se você estiver fazendo um curso, tire suas dúvidas quando elas surgirem, não espere outra oportunidade para fazer isso! É comum alguns alunos ficarem com vergonha de fazer alguma pergunta em sala de aula, ou no caso de curso online, no fórum de discussão.

Faça um esforço para superar esse "medo" e pergunte sempre que tiver alguma dúvida que não conseguiu responder com a ajuda do Google. Ninguém nasce sabendo, e as perguntas dos alunos enriquecem muito mais as aulas.

### 1.5. Desconecte-se dos aplicativos de mensagens instantâneas

Evite perder o foco durante a leitura deste livro ou durante as aulas, se estiver fazendo um curso. O uso de programas de mensagens instantâneas e redes sociais, como Skype, Google Talk e Facebook, são os grandes vilões para você perder todo o investimento de tempo e dinheiro que está fazendo. Se possível, permaneça desconectado de qualquer aplicativo que possa tirar a sua atenção.

### 1.6. Concentre-se

Se estiver fazendo um curso, concentre-se no que o instrutor está explicando. Um instrutor sempre explica os assuntos baseados em suas experiências profissionais. Se você desviar a atenção, pode deixar de aprender coisas muito valiosas para o seu futuro.

Pode parecer estranho, mas muitos alunos desviam a atenção estudando. Enquanto o instrutor explica sobre algum assunto, alguns alunos empolgados podem querer fazer outros exercícios mais avançados. Apesar de ser um bom sinal (pois indica que o aluno aprendeu com facilidade), recomendamos que tome muito cuidado com essa atitude, pois como mencionado anteriormente, você pode estar perdendo informação importante para sua formação.

## 1.7. Não se assuste com a sopa de letrinhas

É muito comum, principalmente em Java, as tecnologias serem identificadas por siglas de poucas letras, e talvez essa seja uma grande barreira que as pessoas que querem aprender Java têm que enfrentar.

Quem quer começar a aprender Java é sobrecarregado com siglas como: JDK, JVM, Java EE, Java SE, JEE, JSE, J2SE, J2EE, J2SDK, JSP, JSF, EL, JSTL, JPA, EJB, JDBC, CDI, JTA e outras milhares de letrinhas. Mantenha a calma e não entre em pânico! Concentre-se em aprender cada coisa de uma vez. Você não é obrigado a conhecer todas as siglas para aprender Java, embora no futuro, se você continuar estudando sempre, vai pelo menos saber para que serve cada uma delas.

## Capítulo 2

# A história do Java

Em 1991, a Sun Microsystems financiou uma pesquisa corporativa interna com o codinome Green, acreditando que a próxima área importante seria os dispositivos eletrônicos inteligentes destinados ao consumidor final. O projeto resultou no desenvolvimento de uma linguagem baseada em C e C++ que seu criador, James Gosling, chamou de Oak (carvalho), em homenagem a uma árvore que dava para a janela do seu escritório na Sun.

Descobriu-se mais tarde que já havia uma linguagem de computador chamada Oak, e quando uma equipe da Sun visitou uma cafeteria local, o nome Java (cidade de origem de um tipo de café importado) foi sugerido e pegou.

O projeto Green demonstrou a linguagem através de um controle interativo voltado para TV a cabo. Era muita inovação para a época, e o mercado para dispositivos eletrônicos inteligentes destinados ao consumidor final não estava se desenvolvendo tão rapidamente como a Sun tinha previsto. Pior ainda, um contrato importante pelo qual a Sun competia fora concedido a outra empresa. Então, o projeto estava em risco de cancelamento.

Por pura sorte, a World Wide Web explodiu em popularidade em 1993 e as pessoas da Sun viram o imediato potencial de utilizar Java para criar páginas da Web com o chamado conteúdo dinâmico (applets). Isso deu nova vida ao projeto.

Em maio de 1995, a Sun anunciou Java formalmente em uma conferência importante. Normalmente, um evento como esse não teria gerado muita atenção. Entretanto, Java





gerou interesse imediato na comunidade comercial por causa do fenomenal interesse pela World Wide Web.

No ano de 2009, durante uma forte crise financeira, a Oracle comprou a Sun, com promessas de continuar inovando e investindo no Java e demais produtos da Sun.

Atualmente, Java está presente em mais de 4,5 bilhões de dispositivos. A plataforma é utilizada para criar páginas da web com conteúdo interativo e dinâmico, para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores da World Wide Web, fornecer aplicativos para dispositivos destinados ao consumidor final e para muitas outras finalidades.

## Capítulo 3

# As plataformas Java

O universo Java é composto por uma gama de plataformas, baseadas na ideia de que um software deveria ser capaz de rodar em diferentes máquinas, sistemas e dispositivos. Por diferentes dispositivos entendemos: computadores, servidores, notebooks, handhelds, PDAs (Palm), celulares, cartões inteligentes (smart cards), TVs, geladeiras e tudo mais o que for possível.

As principais plataformas do mundo Java são: Java SE, Java EE, Java ME, Java Card, JavaFX e Java TV. Vamos conhecer um pouco de cada uma nos próximos tópicos deste capítulo.

### 3.1. A plataforma Java SE

A Java SE (Java Platform, Standard Edition) é a versão básica, destinada ao desenvolvimento da maior parte das aplicações desktop que rodam nas estações de trabalho.

Se você está perdido e não sabe o que baixar no site da Oracle para começar a aprender Java, esta é a plataforma que você precisa.

A Java SE é uma rica plataforma que oferece um completo ambiente para o desenvolvimento de aplicações para clientes e servidores. Ela é também a base de outras plataformas.

A Oracle distribui a Java SE na forma de um SDK (Software Development Kit). O pacote do SDK, também conhecido como JDK (Java Development Kit), vem com ferramentas para compilação, execução, debugging, geração de documentação (javadoc), empacotador de componentes, bibliotecas comuns e etc.

Neste curso nós usaremos esta plataforma, pois é a base para qualquer desenvolvedor Java.

## 3.2. A plataforma Java EE

A Java EE (Java Platform, Enterprise Edition) é uma plataforma padrão para desenvolver aplicações Java de grande porte e/ou para a internet, que inclui bibliotecas e funcionalidades para implementar software Java distribuído, baseado em componentes modulares que executam em servidores de aplicações e que suportam escalabilidade, segurança, integridade e outros requisitos de aplicações corporativas ou de grande porte.

A plataforma Java EE possui uma série de especificações (tecnologias) com objetivos distintos, por isso é considerada uma plataforma guarda-chuva. Entre as especificações da Java EE, as mais conhecidas são:

- Servlets: são componentes Java executados no servidor para gerar conteúdo dinâmico para a web, como HTML e XML.
- JSP (JavaServer Pages): uma especialização de Servlets que permite que aplicações web desenvolvidas em Java sejam mais fáceis de manter. É similar às tecnologias como ASP e PHP, porém mais robusta por ter todas as facilidades da plataforma Java.
- JSF (JavaServer Faces): é um framework web baseado em Java que tem como objetivo simplificar o desenvolvimento de interfaces (telas) de sistemas para a web, através de um modelo de componentes reutilizáveis. A proposta é que os sistemas sejam desenvolvidos com a mesma facilidade e produtividade que se desenvolve sistemas desktop (até mesmo com ferramentas que suportam clicar-e-arrastar componentes).
- JPA (Java Persistence API): é uma API padrão do Java para persistência de dados, que usa um conceito de mapeamento objeto-relacional. Essa tecnologia traz alta produtividade para o desenvolvimento de sistemas que necessitam de integração com banco de dados. Só para citar, essa API possibilita que você desenvolva aplicações usando banco de dados sem precisar escrever uma linha sequer de SQL.
- EJB (Enterprise Java Beans): são componentes que executam em servidores de aplicação e possuem como principais objetivos, fornecer facilidade e produtividade no desenvolvimento de componentes distribuídos, transacionados, seguros e portáteis.

### 3.3. A plataforma Java ME

A Java ME (Java Platform, Micro Edition) é uma plataforma que possibilita a criação de sistemas embarcados em dispositivos compactos, como celulares, PDAs, controles remotos, etc.

A plataforma inclui interfaces visuais flexíveis e suporta o desenvolvimento de aplicações seguras e portáteis.

### 3.4. Java Card, JavaFX e Java TV

A tecnologia Java Card permite que aplicações Java sejam executadas com segurança em cartões inteligentes ou outros dispositivos similares. É muito usada em cartões SIM (chips de telefones celulares) e cartões de bancos. Em cartões SIM, por exemplo, a tecnologia possibilita o armazenamento do identificador do cliente na operadora de telefonia, preferências, agenda de telefones, etc.

A plataforma JavaFX é usada para desenvolver aplicações ricas (no sentido de bonitas e interativas) para a internet, que podem ser executadas em uma ampla variedade de dispositivos conectados, como em ambientes desktop, internet (browser), telefones celulares e TVs.

A Java TV fornece APIs para desenvolvimento de aplicações que são executadas em set-up boxes (receptores de TV digital), players de Blu-ray e outros dispositivos de mídia digital.

### 3.5. Como Java evolui?

A plataforma Java e todas as tecnologias relacionadas evoluem através da força da comunidade mundial.

Para que isso funcione, existe uma entidade chamada JCP (Java Community Process), que organiza e formaliza o processo de definição de uma nova tecnologia ou da atualização de uma tecnologia existente.

Para cada nova ideia de tecnologia ou de atualização, é criada uma JSR (Java Specification Request), que nada mais é que um documento que descreve as propostas de como a tecnologia deve funcionar.

Para iniciar uma JSR, além da ideia inicial do que deve ser feito e qual o problema que se pretende resolver, são nomeados alguns líderes e vários especialistas no assunto, podendo ser pessoas físicas ou empresas. Esses colaboradores conduzem os trabalhos através de reuniões e processos de revisão e votação, até que a versão final seja alcançada.

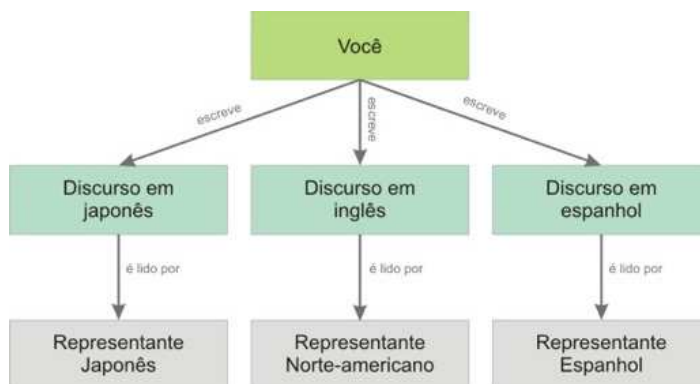
Ao término de uma JSR, tem-se uma documentação completa sobre o funcionamento da tecnologia, uma implementação de referência na forma de código-fonte e um TCK (Technology Compatibility Kit), que é um conjunto de testes que possibilita a validação de um produto desenvolvido, para saber se está em conformidade com a especificação (JSR).

Baseadas nas JSRs (especificações), empresas ou comunidades de todo o mundo podem construir produtos (implementações) e oferecer como alternativas aos já existentes, se diferenciando pela qualidade, suporte, documentação e preço.

# Máquina virtual Java

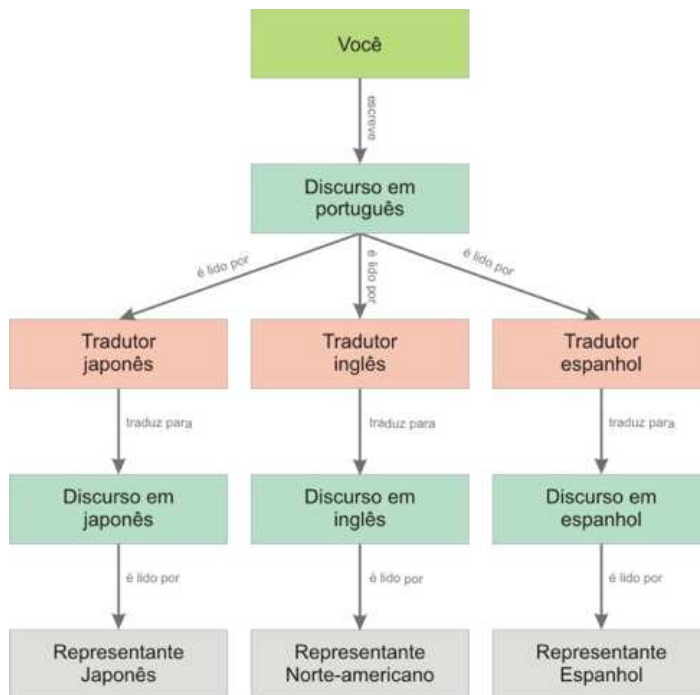
## 4.1. O que é JVM?

Imagine que você queira escrever um discurso para uma reunião que você vai participar com representantes do Japão, Estados Unidos e Espanha. Esses representantes não sabem falar português, por isso, você terá que aprender a língua deles para se comunicar. É um trabalho possível, porém trabalhoso.



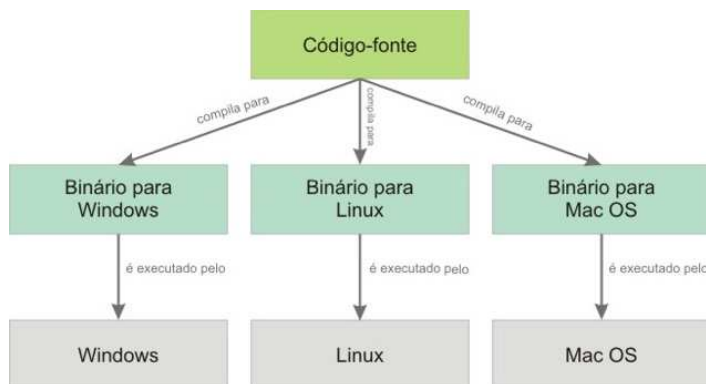
O que você poderia fazer para otimizar seu trabalho? O que acha de contratar tradutores que podem falar o seu idioma, mais japonês, inglês ou espanhol?

Se você fizer isso, estará economizando muito do seu tempo, pois precisará focar apenas no seu texto em português, sem se preocupar com traduções e detalhes de cada língua. É como se fosse uma terceirização do trabalho de tradução para idiomas específicos!



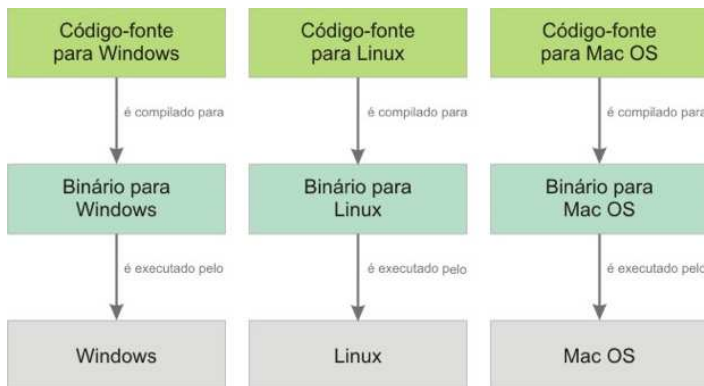
O funcionamento de software funciona quase assim.

Quando você desenvolve e compila um projeto usando a linguagem C, por exemplo, temos o seguinte cenário para um mesmo sistema que deve funcionar em vários sistemas operacionais:



Nessas linguagens de programação, o código-fonte é compilado em código de máquina, que é específico para um SO (sistema operacional) e arquitetura de processador. Isso quer dizer que o binário gerado conhece todos os detalhes do SO para fazer o software funcionar.

Se quisermos que o software seja executado em múltiplos sistemas operacionais ou com tipos de processadores diferentes, precisamos compilar novamente nosso código-fonte. Parece simples, mas não é só isso! Muitas vezes os programadores usam recursos específicos do SO dentro do código-fonte, como bibliotecas nativas de interfaces gráficas ou de gerenciamento de memória, por exemplo. Se este for o caso, uma simples compilação não resolve o problema, precisando ter múltiplos projetos com códigos-fonte para cada SO.

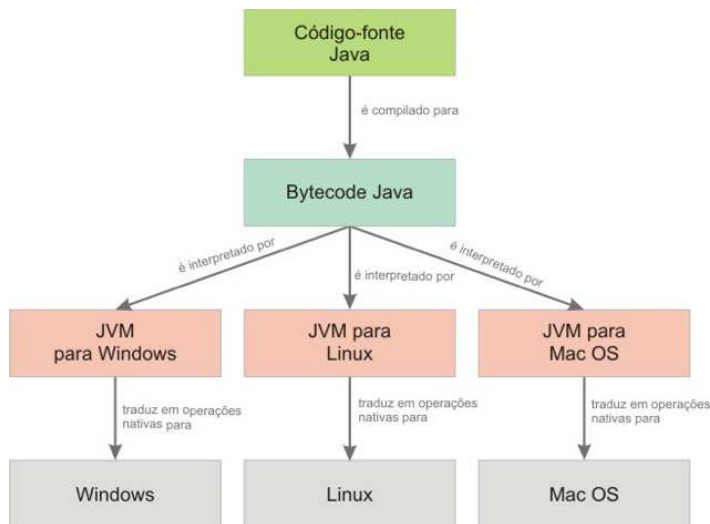


Com Java, você não precisa se preocupar com nada disso! Graças a JVM (Java Virtual Machine ou Máquina Virtual Java), você pode codificar uma única vez e rodar em diferentes sistemas operacionais.

Quando compilamos um código-fonte Java, um arquivo chamado de *bytecode* é gerado. Ele é um arquivo intermediário que não é legível aos programadores (na verdade alguns nerds conseguem ler) e também não é executável por si só (o SO não sabe do que se trata esse arquivo).

Apenas a JVM consegue entender o *bytecode* e traduzir as instruções contidas dentro dele em instruções nativas do sistema operacional.





Dessa forma, um único arquivo compilado (*bytecode*) pode ser executado em qualquer sistema operacional, desde que você possua a JVM compatível para ele. Por exemplo, se você quiser executar um sistema desenvolvido no Windows, basta ter uma JVM para Windows instalada em seu computador, mas se tiver um cliente que use Linux, simplesmente instale uma JVM para Linux. Você não precisa fazer mais nada além disso.

Na verdade, a JVM é muito mais que um simples tradutor. Como o próprio nome diz, é uma máquina virtual, ou seja, uma imitação de um computador que consegue gerenciar a pilha de execução, uso de memória, processamento, segurança, etc.

## 4.2. A JVM faz o Java ficar lento?

Ao contrário do que se parece, a JVM não faz as aplicações desenvolvidas em Java ficarem lentas, mas turbinam elas para que fiquem, em alguns casos, até mais rápidas que aplicações nativas desenvolvidas em C.

Isso é possível graças ao *Hotspot*, uma tecnologia de otimização dinâmica que trabalha para aumentar a performance da JVM em tempo de execução.

Quando codificamos um programa na linguagem C, por exemplo, as otimizações são realizadas em tempo de compilação, enquanto em Java isso é feito *Just in Time (JIT)*, ou seja, no exato momento que é necessário. Isso é vantajoso, pois a JVM consegue fazer estatísticas de execução do código e otimizar a execução de pontos isolados enquanto a aplicação roda.

# Baixando, instalando e configurando

## 5.1. Preciso do JDK ou JRE?

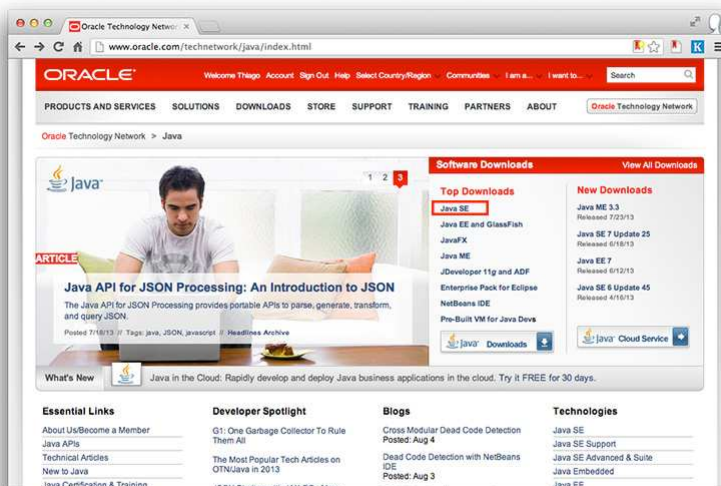
Para desenvolver em Java, precisamos baixar e instalar o JDK (Java Development Kit) da plataforma Java SE. Um kit de desenvolvimento Java possui aplicativos para compilar e debugar seus códigos-fonte, além de diversas outras ferramentas úteis a desenvolvedores de sistemas. O JDK também possui uma JRE (Java Runtime Environment).

JRE é o ambiente de execução da plataforma Java. Usuários de sistemas desenvolvidos em Java precisam instalar apenas o JRE, pois ele possui uma JVM e bibliotecas básicas do Java. Por exemplo, para você usar o software de declaração de imposto de renda desenvolvido pela Receita Federal do Brasil, ou para digitar sua senha de acesso em sites de alguns bancos, você precisará do Java instalado. Isso quer dizer que você precisa, pelo menos, do JRE, pois o software já foi desenvolvido e ele só precisa ser executado no seu computador.

## 5.2. Baixando o JDK da Oracle

Graças ao modelo que o Java é desenvolvido, através de especificações, o próprio kit de desenvolvimento e a JVM podem ser fornecidos por diferentes empresas.

Nós usaremos o JDK da Oracle, que pode ser baixado em <http://java.oracle.com>. Ao acessar este endereço, clique no link "Java SE", na seção "Top Downloads".


















A Oracle fará sugestões para que você possa baixar também outras plataformas ou a IDE de desenvolvimento NetBeans. Neste momento, você não precisa de nada disso, por isso, clique no botão para obter apenas o JDK.



Java Platform (JDK) 7u25

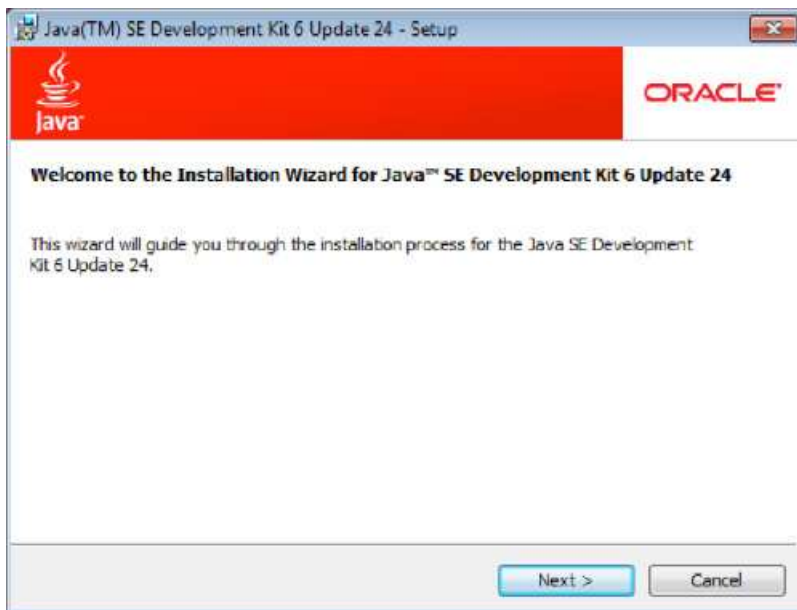
Agora você deve selecionar o seu sistema operacional e arquitetura do processador. Se estiver usando um computador com o sistema operacional rodando em 64-bit, você pode selecionar alguma opção que termine com "x64", mas a opção padrão (32-bit) também deve funcionar. Não esqueça de concordar com a licença da Oracle para fazer o download.

<input checked="" type="radio"/> Accept License Agreement <input type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux x86	80.38 MB	 <a href="#">jdk-7u25-linux-i586.rpm</a>
Linux x86	93.12 MB	 <a href="#">jdk-7u25-linux-i586.tar.gz</a>
Linux x64	81.46 MB	 <a href="#">jdk-7u25-linux-x64.rpm</a>
Linux x64	91.85 MB	 <a href="#">jdk-7u25-linux-x64.tar.gz</a>
Mac OS X x64	144.43 MB	 <a href="#">jdk-7u25-macosx-x64.dmg</a>
Solaris x86 (SVR4 package)	136.02 MB	 <a href="#">jdk-7u25-solaris-i586.tar.Z</a>
Solaris x86	92.22 MB	 <a href="#">jdk-7u25-solaris-i586.tar.gz</a>
Solaris x64 (SVR4 package)	22.77 MB	 <a href="#">jdk-7u25-solaris-x64.tar.Z</a>
Solaris x64	15.09 MB	 <a href="#">jdk-7u25-solaris-x64.tar.gz</a>
Solaris SPARC (SVR4 package)	136.16 MB	 <a href="#">jdk-7u25-solaris-sparc.tar.Z</a>
Solaris SPARC	95.5 MB	 <a href="#">jdk-7u25-solaris-sparc.tar.gz</a>
Solaris SPARC 64-bit (SVR4 package)	23.05 MB	 <a href="#">jdk-7u25-solaris-sparcv9.tar.Z</a>
Solaris SPARC 64-bit	17.67 MB	 <a href="#">jdk-7u25-solaris-sparcv9.tar.gz</a>
Windows x86	89.09 MB	 <a href="#">jdk-7u25-windows-i586.exe</a>
Windows x64	90.66 MB	 <a href="#">jdk-7u25-windows-x64.exe</a>

## 5.3. Instalando o JDK no Windows

Para instalar o JDK no Windows, você já deve ter baixado o arquivo no site da Oracle, conforme explicado na seção anterior. Não vamos cobrir aqui a instalação de kits de desenvolvimento de outros fornecedores.

Encontre o arquivo de instalação do JDK e dê um duplo clique nele. O nome desse arquivo poderia ser, por exemplo, *jdk-7uXX-windows-x64.exe*, onde 7 é o número da versão do Java e XX o número do update (atualização). Ao abrir a primeira tela, clique em **Next**.

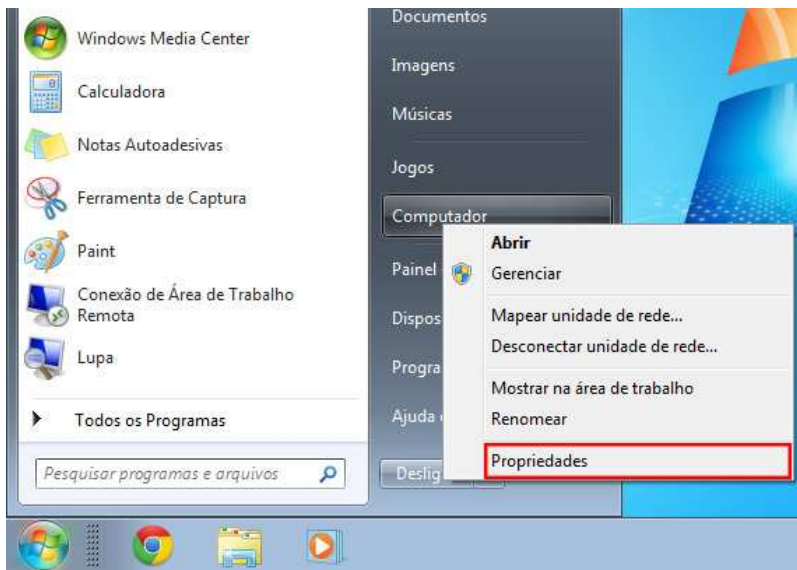


A próxima tela permite selecionar outros recursos que podem ser instalados junto com as ferramentas de desenvolvimento. Também será solicitado o caminho onde o JDK deve ser instalado. Anote o caminho (precisaremos mais adiante), clique em **Next** e aguarde o processo de instalação.

A ferramenta de instalação perguntará onde você quer instalar o JRE. Clique em **Next** e aguarde o término da instalação, depois, clique em **Finish**.

Agora nós precisamos configurar algumas variáveis de ambiente para que as ferramentas de desenvolvimento funcionem adequadamente.

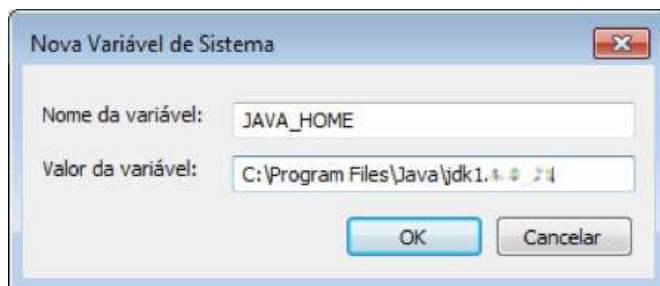
No Windows 7, localize a opção **Computador**, no menu **Iniciar**, e clique com o botão direito sobre ela. Clique sobre a opção **Propriedades**.



Clique no menu **Configurações avançadas do sistema**, no menu lateral direito, depois, clique no botão **Variáveis de ambiente**.

Vamos criar uma nova variável de ambiente do Windows. Para isso, clique no botão **Novo** na seção **Variáveis do sistema**.

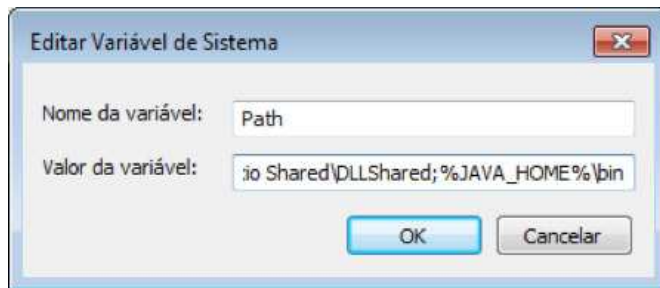
No campo **Nome da variável**, digite `JAVA_HOME`, e no campo **Valor da variável**, digite o caminho onde você instalou o JDK, por exemplo, `C:\Program Files\Java\jdk1.7.0_25`. Clique no botão **OK** para finalizar a inclusão dessa variável.



Repita esse processo de criação de variável, porém agora com o nome da variável igual a `CLASSPATH` e valor igual a `.` (ponto).

Por último, encontre uma variável já existente chamada `PATH` e clique no botão **Editar**.

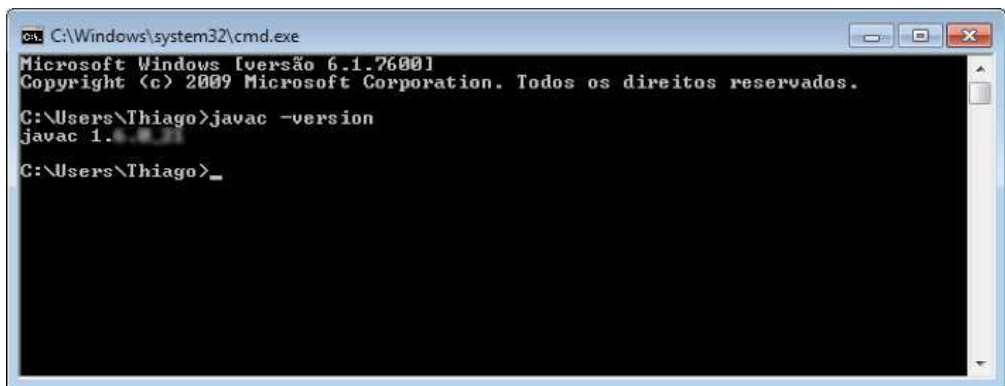
Cuidado para não apagar o conteúdo dessa variável, pois ela é usada pelo sistema operacional. Inclua `;%JAVA_HOME%\bin` ao final dela. Preste atenção, pois você não pode deixar de incluir o ponto e vírgula.



Se você não esqueceu nada e não errou nenhum passo, o ambiente básico com o JDK deve estar funcionando. Para conferir, abra o prompt de comando (`cmd.exe`) e digite:

```
javac -version
```

Você deve conseguir visualizar algo parecido com a tela abaixo.



Se receber uma mensagem dizendo que o comando não é reconhecido, refaça todos os passos, pois provavelmente você errou em alguma coisa.

## 5.4. Instalando o JDK no Linux

O processo de instalação do JDK no Linux depende da distribuição que você usa. No Ubuntu ou outras distribuições similares, a instalação pode ser feita pelo gerenciador de pacotes. Neste caso, você não precisa fazer download do arquivo de instalação do JDK, pois o comando que vamos executar já faz isso através de repositórios que estão na internet.

É muito fácil instalar o JDK da Oracle no Ubuntu. Acesse o terminal e digite:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer
```

Quando a instalação for concluída, verifique se está funcionando. Digite `javac -version` no terminal. Você deve conseguir visualizar algo parecido com a tela abaixo.

```
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) Server VM (build 23.0-b21, mixed mode)
```

Se a versão do Java não for a que você acabou de instalar, tente executar o comando abaixo, para definir a versão que você quer usar.

```
sudo update-java-alternatives -s java-7-oracle
```

Se precisar instalar o JDK em outra distribuição do Linux, talvez o gerenciador de pacotes usado pelo sistema operacional possa fazer isso para você. Caso você queira instalar manualmente (em qualquer distribuição, inclusive no Ubuntu), você pode baixar o arquivo de instalação diretamente do site da Oracle e executá-lo. Neste caso, você precisará configurar as variáveis de ambiente, como `JAVA_HOME`, `CLASSPATH` e `PATH`.



# Fundamentos da linguagem

## 6.1. Vamos instalar a IDE agora?

Quem está começando a aprender uma nova linguagem, é normal que já queira instalar uma ferramenta para ajudar na edição do código-fonte, mas nós não vamos fazer isso agora.

É necessário que você aprenda a sintaxe da linguagem Java antes de contar com a ajuda de uma IDE. Nós usaremos um editor de texto sem muitos recursos, pois assim você cometerá erros e terá a chance de corrigi-los, o que no processo de aprendizado é fundamental.

Você poderá usar seu editor de textos preferido. Caso queira um pouco mais de recursos (como abrir vários arquivos em abas e visualizar o código-fonte com destaques em cores), recomendamos o Sublime Text (<http://www.sublimetext.com>).

Fique tranquilo, pois quando precisarmos de mais produtividade, usaremos uma IDE para continuar os estudos.

## 6.2. Codificando o programa "oi mundo"

Quando você aprende qualquer linguagem de programação, normalmente o primeiro programa que desenvolve é o mais simples possível. Na maioria das vezes ele é chamado de "Olá mundo" ou "Hello World". Para fazer um pouco diferente, chamamos nosso primeiro programa de "Oi Mundo".

O primeiro programa é importante para aprendermos a estrutura básica que usaremos nos próximos exemplos, o processo de compilação e execução e também para resolver erros comuns de principiantes.

Veja abaixo o código-fonte de um programa simples que imprime na saída padrão (console) a mensagem "Oi mundo".

```
class OiMundo {  
  
    public static void main(String[] args) {  
        System.out.println("Oi mundo");  
    }  
  
}
```

Para começar a programar em Java, crie uma pasta para você colocar todos os exemplos e exercícios do curso. Inicie o seu editor de texto favorito e digite o código do exemplo acima.

Por enquanto, não se preocupe com o significado do que escrevemos, pois você aprenderá cada coisa na hora certa. Você deve digitar o código exatamente como foi apresentado no exemplo, inclusive letras maiúsculas e minúsculas, pois a linguagem Java é *case-sensitive*.

Quando terminar de digitar tudo, salve o arquivo com o nome `OiMundo.java`. Preste atenção novamente às letras maiúsculas e minúsculas do nome do arquivo e também na extensão, que deve ser `.java`.

## 6.3. Compilando e executando

Agora vamos compilar nosso primeiro programa. O processo de compilação é o que transforma o código-fonte em *bytecode* (aquele que só a JVM consegue interpretar).

Entre no prompt de comando do Windows (ou terminal no Linux e Mac), acesse a pasta onde você salvou o arquivo `OiMundo.java` e digite:

```
javac OiMundo.java
```

O `javac` é o programa do JDK responsável por compilar um arquivo com código-fonte Java. Se funcionar, o programa ficará silencioso (não aparecerá nenhuma mensagem de sucesso). Se der alguma coisa de errado, você ficará sabendo, pois podem surgir

várias mensagens estranhas no terminal (quando você aprender melhor, não será mais tão estranha assim).

Para confirmar se o arquivo foi compilado, você pode listar todos os arquivos da pasta usando o comando `dir` se estiver usando Windows, ou `ls` se Linux ou Mac. Se um novo arquivo chamado `OiMundo.class` aparecer, é porque você teve sucesso ao compilar seu primeiro programa.

A screenshot of a terminal window titled "estudo — bash — 80x12". The window shows the following commands and output:

```
thiagofa:estudo thiago$ javac OiMundo.java
thiagofa:estudo thiago$ ls
OiMundo.class  OiMundo.java
thiagofa:estudo thiago$
```

O arquivo com extensão `.class` é o *bytecode* gerado (executável). Se você for curioso(a), poderá tentar abri-lo usando um editor de texto. Como disse, só a JVM consegue interpretá-lo (além de, claro, alguns nerds de plantão).

Agora que você já tem o programa compilado, para executá-lo, digite o comando:

```
java OiMundo
```

Preste atenção novamente para as letras maiúsculas e minúsculas.

A terminal window titled 'estudo — bash — 80x12'. The prompt is 'thiagofa:estudo thiago\$'. The user has entered 'java OiMundo' and the output is 'Oi mundo'. The prompt is now 'thiagofa:estudo thiago\$' with a cursor.

```
thiagofa:estudo thiago$ java OiMundo
Oi mundo
thiagofa:estudo thiago$
```

Se funcionar, você deve ver a mensagem "Oi mundo" no seu terminal, como na imagem acima.

Apenas para ter certeza que você entendeu, o que acabamos de executar no último passo foi o arquivo `OiMundo.class`, mas veja que não podemos colocar a extensão dele quando vamos executá-lo. Você poderia ter apagado ou movido o arquivo `OiMundo.java` para outro lugar, e mesmo assim, a execução teria sucesso, pois nesse momento apenas o *bytecode* é lido e executado.

## 6.4. Entendendo o que foi codificado

Vamos estudar um pouco sobre o que codificamos no primeiro programa. Não entraremos em detalhe em tudo para não confundir, mas fique tranquilo que você aprenderá tudo ainda neste livro.

A primeira linha do arquivo declarou um nome de classe. Como ainda não estudamos o que é uma classe (veremos isso em Orientação a Objetos), chamaremos de "programa", portanto, a primeira linha declarou o programa com o nome "OiMundo".

A abertura e fechamento das chaves indicam um bloco de código. Tudo que estiver lá dentro pertence ao programa `OiMundo`.

```
class OiMundo {  
}
```

Dentro do bloco de código do programa, criamos um método chamado `main`. Como ainda não estudamos o que é um método em orientação a objetos, podemos chamá-lo de procedimento ou função (como preferir).

O método `main` é necessário para que nosso programa seja executado quando digitamos o comando `java OiMundo`. Este método é o ponto de entrada do programa, por isso, ele será executado automaticamente quando solicitarmos a execução do programa.

O nome do método não pode ser alterado, pois apesar de não pertencer à sintaxe da linguagem, é um padrão do Java que significa um ponto inicial de um programa desenvolvido.

O bloco de código delimitado pelas chaves deve possuir uma ou mais linhas com a programação do que o sistema deve fazer.

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

Em nosso exemplo, nosso programa apenas imprime "Oi mundo" na tela. Para fazer isso, usamos o método `System.out.println`.

Todo texto (string) em Java é delimitado por aspas duplas, e toda instrução (comando) deve terminar com um ponto e vírgula. Note também que o texto "Oi mundo" está entre parênteses, que indica o início e término de um parâmetro do método.

```
System.out.println("Oi mundo");
```

O nome do arquivo precisou ser exatamente `OiMundo.java`. Em Java, o nome do programa (classe) deve coincidir com o nome do arquivo, portanto, se seu programa chamasse `QueroAprenderJava`, o seu arquivo com o código-fonte deveria ter o nome `QueroAprenderJava.java`.

## 6.5. Erros comuns dos marinheiros de primeira viagem

Marinheiros de primeira viagem costumam cometer erros comuns, pois a linguagem Java é um pouco burocrática. Vejamos alguns erros que você pode cometer e como corrigi-los:

1. Ao compilar, aparece o erro:

OiMundo.java:3: cannot find symbol symbol : class string

Você digitou string com a letra "s" em minúsculo. O correto é:

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

2. Ao compilar, aparece o erro:

package system does not exist

Você digitou system com a letra "s" em minúsculo. O correto é:

```
System.out.println("Oi mundo");
```

3. Ao compilar, aparece o erro:

';' expected

Você esqueceu de finalizar a instrução com um ponto e vírgula. Veja:

```
System.out.println("Oi mundo");
```

4. Ao compilar, aparece o erro (ou algo parecido):

class x is public, should be declared in a file named x.java

Você mudou o nome do programa, mas não mudou o nome do arquivo adequadamente, ou esqueceu de colocar as iniciais do nome do programa em letras maiúsculas:

```
class OiMundo
```

5. Ao executar, aparece o erro (ou algo parecido):

Exception in thread "main" java.lang.NoClassDefFoundError: x (wrong name: X)

Você digitou o nome do programa sem levar em consideração as letras maiúsculas e minúsculas. Digite o comando java OiMundo com as iniciais do nome do programa usando letras maiúsculas.

6. Ao compilar, aparece o erro (ou algo parecido):

Exception ... "main" java.lang.NoClassDefFoundError: OiMundo

A variável de ambiente CLASSPATH de seu computador não está configurada ou está incorreta. Verifique se executou todos os passos de instalação e configuração corretamente.

## 6.6. Comentários

Comentários são textos que podem ser incluídos no código-fonte, normalmente para descrever como determinado programa ou bloco de código funciona. Os comentários são ignorados pelo compilador, por isso, eles não modificam o comportamento do programa.

Em Java, você pode comentar blocos de códigos inteiros ou apenas uma linha. Para comentar uma única linha, faça como no exemplo a seguir:

```
class OiMundo {  
  
    public static void main(String[] args) {  
        // imprime uma mensagem na saída padrão  
        System.out.println("Oi mundo");  
    }  
}
```

Para comentar um bloco de código, use `/* */` para abrir e fechar. Veja um exemplo:

```
class OiMundo {  
  
    public static void main(String[] args) {  
        /*  
        Esta linha será ignorada pelo compilador.  
        System.out.println("Esta instrução será ignorada também");  
        E esta linha também. */  
        System.out.println("Oi mundo");  
    }  
}
```

## 6.7. Sequências de escape

Sequências de escape são combinações de caracteres iniciadas por `\` (contra barra) e usadas para representar caracteres de controle, aspas, quebras de linha, etc.

Para ficar melhor entendido, vamos fazer um teste. O exemplo abaixo tenta imprimir a mensagem *Oi "Maria"* (com o nome "Maria" entre aspas duplas).

```
class ExemploEscape {

    public static void main(String[] args) {
        System.out.println("Oi "Maria");
    }

}
```

Ao tentarmos compilar, temos um belo erro (não tão bonito assim):

```
ExemploEscape.java:4: error: ')' expected
    System.out.println("Oi "Maria");
                        ^
ExemploEscape.java:4: error: not a statement
    System.out.println("Oi "Maria");
                        ^
ExemploEscape.java:4: error: ';' expected
    System.out.println("Oi "Maria");
                        ^
3 errors
```

O problema aqui é que o compilador Java não conseguiu entender o que significa a palavra "Maria". Se você prestar atenção, notará que a segunda aspa fechou a primeira, e a quarta fechou a terceira logo após o nome "Maria". Você percebeu que não existem aspas envolvendo o nome "Maria"?

E se quisermos dizer ao compilador que "Maria" é um texto, mas que as aspas que envolvem o nome também são textos? Eis que usamos uma sequência de escape.

```
System.out.println("Oi \"Maria\")");
```

A sequência \" diz ao compilador que a aspa deve ser considerada como um texto, e não como um delimitador de String. Agora nosso programa deve compilar e rodar normalmente.

Ao executar nosso programa, as aspas que envolvem o nome aparecem na saída padrão:

```
Oi "Maria"
```

Existem várias outras sequências de escape, sendo que as mais conhecidas são:

- \n para nova linha
- \\ para uma barra invertida
- \" para aspas duplas



## 6.8. Palavras reservadas

As palavras-chave são palavras especiais, reservadas em Java, que têm significados para o compilador, que as usa para determinar o que seu código-fonte está tentando fazer. Você não poderá usar as palavras-chave como identificadores (nomes) de classes, métodos ou variáveis. As palavras-chave reservadas estão listadas a seguir:

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

Você aprenderá a maioria das palavras-chave da listagem acima no momento que for adequado, por isso, não se preocupe em memorizá-las.

Apesar de goto e const serem palavras-chave reservadas, elas não possuem significado para o compilador. Na verdade, se você tentar usá-las, receberá um erro ao compilar seu código-fonte.

## 6.9. Convenções de código

Ao desenvolver em Java, encorajamos você a usar as convenções de codificação da linguagem Java, fornecido oficialmente pela Oracle. Esse padrão é usado internamente pela Oracle no desenvolvimento do próprio Java e por praticamente todas as empresas do mundo que desenvolve algum software sério em Java.

O documento completo, que aborda apenas forma de escrita, nomes de arquivos, classes, variáveis, indentação, comentários, métodos e organização do código-fonte, pode ser lido em:

<http://www.oracle.com/technetwork/java/index-135089.html>

As vantagens em utilizar as convenções é que, além de você programar usando o mesmo padrão em todo o mundo, também facilita a manutenção no futuro por outros desenvolvedores e até mesmo por você, pois aumenta a legibilidade do código-fonte.

Sempre que acharmos necessário, citaremos algumas convenções de código para você já ir se acostumando.

Para começar, é muito comum, praticamente uma regra, que os nomes de seus programas (classes) usem uma prática chamada *CamelCase*. Isso significa que as palavras em uma frase que nomeia seu programa devem ser iniciadas com letras maiúsculas e unidas sem espaços. Por exemplo, se você tiver um programa responsável por gerar notas fiscais, um nome válido seria `GeradorNotaFiscal`, mas não seria correto, de acordo com esta prática, usar os nomes `Geradornotafiscal`, `geradornotafiscal` ou `Gerador_Nota_Fiscal`. Veja:

```
class GeradorNotaFiscal {  
}
```

## 6.10. Trabalhando com variáveis

Em linguagens de programação, variáveis são nomes simbólicos dados a informações alocadas na memória do computador. As variáveis são definidas, atribuídas, acessadas e calculadas através do código-fonte do programa. Durante a execução de um programa, os conteúdos das variáveis podem mudar através de algum processamento.

Em Java, as variáveis devem ser declaradas com um tipo fixo para poderem ser usadas. Isso quer dizer que, uma variável do tipo inteiro, por exemplo, não poderá ser alterada para um tipo real (decimal).

Para começar, vamos declarar uma variável chamada `quantidade`, do tipo `int`. O tipo `int` é capaz de armazenar apenas valores inteiros negativos ou positivos. Veja:

```
int quantidade; // declarando variável inteira
```

A variável acima foi apenas declarada, isso quer dizer que não atribuímos nenhum valor para ela. Para fazermos isso, usamos o operador `=` (igual) seguido por um número inteiro.

```
quantidade = 10; // atribuindo o valor 10
```

Agora a variável `quantidade` possui o valor `10`. Se precisarmos alterar o valor da variável `quantidade`, podemos atribuir um novo valor a ela. Vamos dizer que a variável deve possuir o valor `15`.

```
quantidade = 15; // atribuindo o valor 15
```

É muito comum precisarmos mostrar o valor de uma variável na tela do usuário. Para fazermos isso de uma forma bem simples, podemos usar `System.out.println`, passando como parâmetro o nome da variável.

```
// imprimindo o valor da variável
System.out.println(quantidade);
```

Veja como ficou este exemplo completo. Incluímos mais uma instrução de impressão da variável `quantidade` entre a atribuição do valor `10` e do valor `15` para podermos ver o valor da variável antes de depois de ser modificada.

```
class Variaveis1 {

    public static void main(String[] args) {
        int quantidade;
        quantidade = 10;
        System.out.println(quantidade);
        quantidade = 15;
        System.out.println(quantidade);
    }

}
```

Se você quiser economizar uma linha de código, pode declarar e atribuir a variável na mesma linha, como no exemplo abaixo:

```
int quantidade = 10; // declarando e atribuindo
```

Veja como ficaria no exemplo completo.

```
class Variaveis1 {

    public static void main(String[] args) {
        int quantidade = 10;
        System.out.println(quantidade);
        quantidade = 15;
        System.out.println(quantidade);
    }

}
```

## 6.11. Nomeando variáveis

As variáveis em Java podem conter letras, dígitos, `_` (*underscore*) e `$` (dólar), porém elas não podem ser iniciadas por um dígito e não podem ser palavras reservadas.

Veja alguns nomes de variáveis válidos:

```
int quantidade; // pode ser toda em letras minúsculas
int quantidade_alunos; // pode ter underscore
int QUANTIDADE; // pode ser toda em letras maiúsculas
int QuantidadeAlunos; // pode ter letras maiúsculas e minúsculas
int $quantidade; // pode iniciar com dólar
int _quantidade; // pode iniciar com underscore
int quantidade_alunos_nota_10; // pode ter dígitos
```

Agora alguns nomes de variáveis inválidos (que nem compila):

```
int 2alunos; // não pode iniciar com dígitos
int quantidade alunos; // não pode ter espaços
int new; // new é uma palavra reservada do Java
```

Apesar da linguagem suportar letras maiúsculas e minúsculas, *underscore*, dólar e dígitos nos nomes das variáveis, a convenção de código Java diz que elas devem ser nomeadas com a inicial em letra minúscula e as demais iniciais das outras palavras em letras maiúsculas. Veja alguns exemplos:

```
int quantidadeAlunos;
int quantidadeAlunosNota10;
int numeroDeAlunosAprovados;
int totalAlunosReprovados;
```

As declarações de variáveis abaixo estão corretas para o compilador, mas não estão de acordo com o padrão de código usado mundialmente, por isso, evite-as a todo custo.

```
int quantidade_alunos;
int QuantidadeAlunosNota10;
int NUMERODEALUNOSAPROVADOS;
int Total_Alunos_Reprovados;
```

É uma boa prática escrever as palavras completas quando vamos declarar variáveis em Java. Abreviações devem ser usadas somente se forem muito conhecidas no domínio do negócio. Por exemplo, você deve **evitar**:

```
int qtAlu;
int quantAlunNt10;
int nAlunosAprov;
int totAlunosRep;
```

Essas regras não servem apenas para a linguagem Java, mas existe uma cultura muito forte entre os bons programadores Java que prezam pela clareza do código, e um nome de variável mal definido pode atrapalhar muito a legibilidade do código.

## 6.12. Operadores aritméticos

Existem 5 operadores aritméticos em Java que podemos usar para efetuar cálculos matemáticos. Uma operação pode ser de adição (+), subtração (-), multiplicação (\*), divisão (/) ou módulo (%). Só para lembrar quem faltou nas aulas de matemática no colégio, módulo é o resto da divisão entre dois números. Outras operações, como exponenciação (potência), raiz quadrada e outras são fornecidas de maneiras diferentes (que estudaremos mais adiante).

Vejamos um exemplo simples usando as 5 operações aritméticas:

```
int soma = 2 + 10;
int subtracao = 6 - 10;
int multiplicacao = 8 * 3;
int divisao = 8 / 2;
int resto = 7 % 2;

System.out.println(soma);
System.out.println(subtracao);
System.out.println(multiplicacao);
System.out.println(divisao);
System.out.println(resto);
```

Os resultados das operações acima são: 12, -4, 24, 4 e 1.

No último exemplo, calculamos valores literais (digitados "na mão"). Podemos ainda calcular valores de variáveis, tornando o programa muito mais dinâmico.

```
int notaAluno1 = 99;
int notaAluno2 = 80;
int notaAluno3 = 53;

int totalGeral = notaAluno1 + notaAluno2 + notaAluno3;
System.out.println(totalGeral);
```

Pense rápido! Qual será o resultado da operação acima? Se você disse 232, parabéns!

Para praticar um pouco mais, queremos agora descobrir qual é a média de notas dos 3 alunos que temos. O que você acha que acontece se dividirmos por 3?

```
int totalGeral = notaAluno1 + notaAluno2 + notaAluno3 / 3;
```

A ideia é muito boa. Para descobrirmos a média de notas de 3 alunos, basta somarmos todas as notas e dividir por 3, mas como fizemos no exemplo acima, estamos dividindo a última nota por 3, e não o resultado da somatória de todas as notas. Por isso, o resultado dessa operação é 196.

Para ficar correto, temos que agrupar a somatória usando parênteses, assim, dizemos ao compilador que queremos realizar a operação de soma antes da divisão.

```
int totalGeral = (notaAluno1 + notaAluno2 + notaAluno3) / 3;
```

Agora sim, o resultado da operação ficará correto, resultando na média de 77 pontos por aluno. Nada mal.

## 6.13. Tipos primitivos

Nos exemplos anteriores, vimos como usar variáveis para armazenar apenas valores inteiros. Agora vamos estudar como criar variáveis para armazenar valores do tipo ponto-flutuante (com casas decimais). Por exemplo, vamos declarar uma variável e atribuir um valor com o preço de um produto:

```
double precoProduto = 9.43;
```

O tipo `double` é capaz de armazenar valores reais, que possuem casas decimais, mas também pode armazenar números inteiros.

Outro tipo primitivo bastante usado é o `boolean`. O tipo booleano pode armazenar os valores verdadeiro ou falso (`true` ou `false`). Por exemplo:

```
boolean alunoMatriculado = true; // recebe valor "verdadeiro"
boolean clienteBloqueado = false; // recebe valor "falso"
```

Os valores literais `true` e `false` são palavras reservadas do Java. Não é válido atribuir números a uma variável booleana. Não existe nenhuma referência entre um valor booleano e os números 0 e 1 ou qualquer outro número, como em outras linguagens que talvez você já conheça, por isso, o código abaixo é inválido e não compila:

```
boolean alunoMatriculado = 1; // não compila
boolean clienteBloqueado = 0; // não compila
```

Uma variável booleana pode receber como valor uma expressão com operadores de comparação ou igualdade, mas estudaremos isso em breve.

Depois de aprender como usar tipos inteiros, reais e booleanos, vamos conhecer mais sobre outros tipos primitivos do Java.

Os tipos de dados básicos (tipos primitivos) da linguagem Java são: boolean, char, byte, short, int, long, float e double.

A tabela abaixo mostra os tipos primitivos e a capacidade de armazenamento de cada um.

Tipo	Tamanho (bits)	Menor valor	Maior valor
boolean	1	false	true
char	16	0	$2^{16} - 1$
byte	8	$-2^7$	$2^7 - 1$
short	16	$-2^{15}$	$2^{15} - 1$
int	32	$-2^{31}$	$2^{31} - 1$
long	64	$-2^{63}$	$2^{63} - 1$
float	32	-	-
double	64	-	-

Como pode ver na tabela dos tipos primitivos do Java, o tipo boolean ocupa apenas 1 bit para armazenar um valor booleano, ou seja, este tipo precisa apenas de uma da menor unidade de armazenamento na computação.

Uma variável do tipo char ocupa 16 bits, ou seja, 2 bytes (cada byte tem 8 bits) para armazenar um valor que representa um caractere. Veja abaixo alguns exemplos de declaração e atribuição de variáveis do tipo char:

```
char turmaAluno1 = 'A';
char tipoCliente = '2';
char simbolo = '@';
System.out.println(turmaAluno1);
System.out.println(tipoCliente);
System.out.println(simbolo);
```

O tipo char não pode ser usado para armazenar um texto. Na verdade, este tipo é capaz de armazenar apenas um caractere. Nós aprenderemos o tipo String ainda neste livro para armazenar textos (com mais de 1 caractere, naturalmente).

Os tipos `byte` e `short` são tipos numéricos inteiros com uma capacidade de armazenamento menor que o tipo `int`. Para saber exatamente o menor e maior número que cada tipo suporta, basta fazer o cálculo da potência, conforme apresentado na tabela. Por exemplo, o menor valor do tipo `byte` é  $-2^7$ , ou seja,  $-128$ , e o maior é  $2^7 - 1$ , ou seja,  $127$ .

O tipo `long` é um tipo numérico inteiro (longo) com capacidade de armazenamento superior ao tipo `int`. Para se ter uma ideia, o tipo `long` é capaz de armazenar o valor máximo igual a `9223372036854775807`, enquanto o `int` suporta até o número `2147483647`.

No exemplo abaixo, declaramos e atribuímos uma variável para armazenar o número total de habitantes na cidade de Uberlândia/MG. Esta variável poderia ser do tipo `int`, mas já que estamos falando de `long`, exageramos um pouco no exemplo e declaramos como um inteiro longo.

```
long populacaoUberlandia = 650000;  
System.out.println(populacaoUberlandia);
```

O código acima funciona, não temos nenhum problema com ele. Agora veja o código abaixo:

```
long populacaoMundial = 7000000000; // não compila  
System.out.println(populacaoMundial);
```

O último exemplo não compila! Pode parecer estranho, mas isso acontece porque o número `7000000000` não é compatível com o tipo `int`. Espere... mas não estávamos usando um tipo `int`, e sim um `long`. Certo? Sim e não, certo e errado! Quando atribuímos um valor literal (digitado manualmente no código-fonte) a uma variável do tipo `long`, o valor literal é por natureza do tipo `int`, ou seja, os literais numéricos inteiros são do tipo `int` por padrão, a não ser se usarmos um pequeno truque. Veja como é fácil:

```
long populacaoMundial = 7000000000L; // compila!  
System.out.println(populacaoMundial);
```

A única diferença do último exemplo para o que não compila é a letra `L` após o número `7000000000`. Ao incluir a letra `L` ao final de um número literal, indicamos ao compilador que queremos que o número seja interpretado como um tipo `long`, e não um `int`. Capcioso, não?



Os tipos `float` e `double` são os únicos tipos ponto-flutuante que são capazes de armazenar números reais (com casas decimais). Enquanto o tipo `float` possui 32 bits de precisão, o `double` possui 64 bits.

Já vimos como declarar variáveis do tipo `double`, agora tentaremos fazer o mesmo com o tipo `float`.

```
float saldoConta = 1030.43; // não compila
System.out.println(saldoConta);
```

O código acima não compila porque todo literal decimal em Java é por padrão do tipo `double` (independente do valor). Por isso, precisamos mais uma vez tirar uma "carta na manga"! Para fazer o código compilar e rodar como desejamos, precisamos apenas incluir a letra `F` após o número `1030.43`.

```
float saldoConta = 1030.43F; // compila!
System.out.println(saldoConta);
```

A letra `F` diz ao compilador que queremos que o número seja entendido como um valor do tipo `float`, e não `double`.

Os tipos `float` e `double` não devem ser usados para armazenar valores que devem ter muita precisão, como valores monetários. Para isso, nós vamos aprender outro tipo (que não é primitivo) mais a frente, chamado `BigDecimal`. Até lá, você pode usar `float` e `double` para o que precisar.

## 6.14. Outros operadores de atribuição

Você já sabe para que serve o operador de atribuição `=` (igual), pois já usamos em vários exemplos anteriores. Agora vamos estudar outros operadores de atribuição que combinam com operadores aritméticos. Para começar, vejamos um exemplo do operador `+=`, que atribui um valor à variável somando o valor da própria variável com o número (ou variável) à direita do operador.

```
int total = 10;
total += 3;
System.out.println(total);
```

O código acima imprime na tela o número 13. A linha `total += 3` é o mesmo que `total = total + 3`. Então porque usar essa forma abreviada? A resposta é óbvia... simplesmente para abreviar.

Outros operadores de atribuição básicos da linguagem Java são -=, \*=, /= e %=. Veja um exemplo usando todos eles.

```
int total = 10;
total += 3; // soma total com 3
System.out.println(total);
total -= 1; // subtrai total com 1
System.out.println(total);
total *= 2; // multiplica total por 2
System.out.println(total);
total /= 4; // divide total por 4
System.out.println(total);
total %= 5; // resto de total dividido por 5
System.out.println(total);
```

Existem outros operadores de atribuição em Java para manipulação de bits, mas não é nosso foco neste livro.

## 6.15. Conversão de tipos primitivos

Durante a programação de um sistema, pode surgir a necessidade de atribuir a uma variável, o valor de outra, porém de tipos diferentes. Isso é chamado de *casting* (conversão), pois antes da atribuição, um processo de conversão de um tipo para o outro deve ser realizado pela JVM. Vejamos o seguinte exemplo:

```
// declaramos a variável x do tipo long
long x = 10;

// agora tentamos atribuir x a y, do tipo int
int y = x; // não compila
```

O código acima não compila, pois não é possível atribuir x à y. Pode parecer estranho, pois x possui o valor 10, que é perfeitamente compatível com o tipo int (lembre-se que int possui um limite máximo até o número 2147483647). Porque então não compila?

Não tem como o compilador saber que a variável x possui o valor 10, pois como o próprio nome diz, é uma variável! Quem garante que x não possa ter um valor absurdamente longo, que não cabe em um int? Por isso, para garantir que nada vai dar de errado em tempo de execução de seu programa, o compilador Java nos ajuda, evitando que nosso programa seja compilado.

Para clarear um pouco mais, imagine o seguinte: um tipo long possui capacidade de 64 bits. É como se existissem 64 pequenos espaços na memória do computador para

armazenar um valor. Mesmo que o número seja bastante pequeno, todos os espaços sempre serão ocupados.

Tamanho do tipo long



Já um tipo `int` possui a capacidade de 32 bits. Usando a mesma analogia, são como 32 pequenos espaços na memória para armazenar um valor.

Tamanho do tipo int



Agora imagine que você não saiba qual valor possui dentro das variáveis (o compilador não sabe) e responda: você consegue colocar com segurança o valor que tem na variável do tipo `long` dentro da variável do tipo `int`?

Se você entendeu bem e analisou os fatos friamente, com certeza respondeu que "não". É impossível saber com certeza se o valor do tipo `long` caberá na variável do tipo `int`.

Mas Java é uma linguagem muito poderosa, e por isso não nos deixaria na mão. Se você tiver certeza ou precisar muito fazer essa conversão milagrosa, poderá forçar um *casting*, ou seja, você pode dizer ao compilador Java: "tudo bem, eu sou responsável e assumo todos os riscos que isso pode trazer". Veja como é fácil:

```
long x = 10;

// agora compila, mas os riscos são todos seus
int y = (int) x;
System.out.println(y);
```

Graças à instrução de *casting* que incluímos, o conteúdo da variável `x` será convertido para `int` e atribuído a `y`. Neste caso, não existirá nenhum efeito colateral, pois o código é bastante simples e estamos vendo que o valor de `x` é realmente 10, porém no mundo real, muitas vezes essa não é a situação, portanto, muito cuidado nessa hora.

E se ignorarmos as forças maiores e solicitarmos um *casting* de uma variável do tipo `long`, que tem um valor tão longo, que não cabe em um `int`? Um exemplo diz mais que mil palavras. Vejamos:

```
long x = 9300000035L;

// o valor mudará absurdamente (cuidado)
int y = (int) x;
System.out.println(y);
```

A execução do código acima imprimirá na tela o número 710065443. O que tem haver esse número com 9300000035? Nada... é verdade, mas assumimos todos os riscos, forçamos o *casting* da variável `x` e aconteceu o pior, o valor da variável `x` não coube em um tipo `int`, e por isso perdemos o valor original e ganhamos um outro que não queríamos.

Neste caso, o correto a fazer, seria não fazer! Isso mesmo, o correto seria não fazer o *casting*, pois como bons programadores, nós deveríamos saber que a variável `x` poderia, em algum momento, ter um valor tão longo ao ponto de não ser possível fazer a conversão com segurança.

Mas porque o valor mudou para outro totalmente diferente? Porque estávamos usando muitos dos 64 "espaços" (bits) do tipo `long`, e a conversão eliminou vários destes "espaços" para fazer caber dentro do tipo `int`. Quando fazemos isso, estamos perdendo bits que representam o número original, e por isso é natural que o resultado seja outro bem diferente.

Outra situação bastante comum que necessitamos fazer *casting* é de variáveis do tipo `int` para `long`. Veja um exemplo:

```
int y = 102344;

long x = y; // casting feito automaticamente
System.out.println(x);
```

O código acima compila e executa sem problemas. Quando tivermos um tipo `int` e desejarmos atribuir a outra variável do tipo `long`, podemos fazer isso sem correr riscos, por isso, não é necessário instruir ao compilador que você quer que seja feita uma conversão (ele fará automaticamente).

As conversões de tipos `float` para `double` e vice-versa acontecem da mesma forma que para os tipos `int` e `long`. Veja um exemplo:

```
double a = 20.5;

// você assume os riscos desta conversão
float b = (float) a;
System.out.println(b);

float c = 934.5f;

// conversão automática
double d = c;
System.out.println(d);
```

Como o tipo `double` possui 64 bits, é necessário deixar explícito a conversão para `float`, pois existem riscos de perder informação, já o contrário (de `float` para `double`) você não precisa dizer que existirá uma conversão, pois o Java fará isso de forma implícita para você.

Outra situação comum acontece quando precisamos atribuir valores de tipos ponto-flutuante para variáveis inteiras. O exemplo abaixo é uma tentativa de fazer isso:

```
double largura = 100;
int tamanho = largura; // não compila
```

O último exemplo não compila, pois o compilador não assume o risco de converter o valor de uma variável do tipo `double` (ou `float`) para uma variável do tipo `int`, `long` ou qualquer outra inteira.

Mesmo sabendo que o conteúdo da variável `largura` seria perfeitamente representada dentro da variável `tamanho`, o coitado do compilador não consegue identificar isso. Se quisermos realmente assumir todos os riscos, precisamos deixar explícita a conversão:

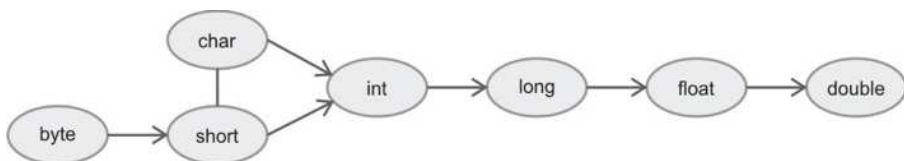
```
double largura = 100;
int tamanho = (int) largura; // agora sim, compila!
System.out.println(tamanho);
```

No caso do exemplo acima, está visível que não haverá nenhum problema. Mas e o código abaixo? O que acontecerá?

```
double largura = 100.37;
int tamanho = (int) largura; // compila, mas perdemos precisão
System.out.println(tamanho);
```

A execução do exemplo acima imprimirá na tela o número `100`. Isso acontece porque fizemos a conversão de `double` para `int`, mas dessa vez perdemos precisão. O número `100.37` não pôde ser representado como um inteiro, e por isso ele foi "truncado", ou seja, o valor decimal foi eliminado.

Se você precisar fazer *casting* de tipos diferentes ao que estudamos, veja a imagem abaixo e siga algumas dicas.



- No sentido das flechas, a conversão é implícita, ou seja, aquela que você não precisa escrever nada (e mesmo assim acontecerá automaticamente).
- No sentido contrário ao das flechas, a conversão deve ser explícita, ou seja, você deve escrever no código-fonte que deseja assumir os riscos.

## 6.16. Promoção aritmética

Quando calculamos duas variáveis ou valores do tipo `int`, por exemplo, o resultado é do tipo `int`.

```
int x = 10;
int y = 5;
int z = x + y;
System.out.println(z);
```

Se realizarmos um cálculo com duas variáveis ou valores do tipo `long`, o resultado é do tipo `long`.

```
long x = 10;
long y = 5;
long z = x * y;
System.out.println(z);
```

Agora, se realizarmos uma operação entre uma variável do tipo `int` com uma variável do tipo `long`, podemos atribuir o resultado em um tipo `int`?

```
int x = 10;
long y = 5;
int z = x * y; // não compila!
System.out.println(z);
```

Não, não... não podemos! Quando realizamos operações aritméticas entre variáveis de tipos diferentes, o resultado será igual ao maior tipo, e isso se chama promoção aritmética. Como `long` tem maior capacidade de armazenamento que `int`, o resultado é do tipo `long`, por isso, seria correto atribuímos em uma variável desse tipo.

```
int x = 10;
long y = 5;
long z = x * y; // agora compila, pois o resultado é long
System.out.println(z);
```

Para sabermos exatamente qual o tipo resultante em uma operação aritmética, basta olharmos a tabela de tipos primitivos estudada anteriormente. O tipo com a maior capacidade em bits será o tipo do resultado do cálculo. A exceção a essa regra é

quando realizamos uma operação entre números de tipos inteiros (int, long, etc) e ponto-flutuante (float ou double).

```
long x = 10;
float y = 9.34;
long z = x * y; // não compila
System.out.println(z);
```

O exemplo acima não compila, pois o resultado de uma operação entre float e long é float. Apesar de long ter 64 bits e float ter 32 bits, float leva a vantagem por ser ponto-flutuante, portanto, o correto seria atribuir o resultado dessa operação em um tipo float.

```
long x = 10;
float y = 9.34;
float z = x * y; // compila
System.out.println(z);
```

Por último, apenas para você refletir um pouco, tente descobrir o que será impresso na tela com o código abaixo:

```
int x = 3;
int y = 2;
float z = x / y;
System.out.println(z);
```

O exemplo é bem simples, estamos simplesmente tentando dividir x por y, ou seja, 3 por 2. A grande questão é que estamos dividindo dois números inteiros, portanto, o resultado também será do tipo inteiro.

Se você acha que aparecerá na tela o valor 1.5 (3 dividido por 2 é 1.5), se enganou! Nós veremos 1.0 ao executar esse código. Apesar de atribuirmos o resultado em um tipo float (que suporta casas decimais), o cálculo é feito usando duas variáveis do tipo int, portanto, o resultado também é do tipo int.

Se não quisermos perder os valores decimais do resultado da operação, podemos ter pelo menos uma das variáveis que participam do cálculo do tipo float.

```
int x = 3;
float y = 2;
float z = x / y;
System.out.println(z);
```

Agora sim, o resultado que veremos na tela será 1.5, pois x é promovido a float no momento da operação.

Se não existir essa possibilidade (de mudar o tipo da variável para float), podemos fazer um *casting* de x ou y para float antes de realizar o cálculo.

```
int x = 3;
int y = 2;
float z = x / (float) y; // o casting acontece antes do cálculo
System.out.println(z);
```

O resultado continua sendo 1.5, pois convertermos o valor de y para float, obrigando x a ser promovido também para float durante a operação aritmética.

## 6.17. Trabalhando com strings

Para apresentarmos um texto na tela usando Java, tudo que precisamos fazer é colocá-lo entre aspas duplas dentro de um `System.out.println`.

```
System.out.println("Oi galera do bem");
```

Se precisarmos juntar um texto com o valor de uma variável, podemos concatená-los. Na linguagem Java, usamos o símbolo + (mais) para fazer isso.

```
int x = 10;
int y = 5;
int z = x + y; // adição
System.out.println("Resultado: " + z); // concatenação
```

Quando usamos o + em textos, ele é entendido pelo compilador como concatenação, e não adição.

No exemplo acima, poderíamos eliminar a variável z e efetuar o cálculo diretamente no `println`. Veja abaixo uma tentativa de fazer isso:

```
int x = 10;
int y = 5;
System.out.println("Resultado: " + x + y);
```

A execução do último exemplo exibirá na tela "Resultado: 105", pois o valor da variável x é concatenado com o valor da variável y. Isso acontece porque o compilador verifica que existe um texto sendo concatenado com a variável x, e faz o mesmo para a variável y.

Agora veja outro exemplo:



```
int x = 10;
int y = 5;
System.out.println(x + y + " foi o resultado");
```

Você apostaria que o código acima imprime na tela "15 foi o resultado"?

Apesar de parecer estranho, é isso que acontece. Neste caso, as variáveis `x` e `y` são somadas, e não concatenadas. Isso acontece porque o compilador só começa a concatenar a partir do momento que encontra um texto.

E se quisermos efetuar o cálculo de `x` com `y` depois que um texto foi digitado, como na primeira situação? Basta incluirmos a operação entre parênteses para dizer ao compilador que a operação tem precedência.

```
int x = 10;
int y = 5;
System.out.println("Resultado: " + (x + y));
```

Agora o resultado será "Resultado: 15".

Para declarar variáveis que contenham textos, usamos o tipo `String`.

```
String nome = "Maria";
```

Apesar do tipo `String` ser nativo da linguagem Java, ele não é um tipo primitivo. Por enquanto, você não precisa se preocupar com isso. Iremos aprofundar no tipo `String` mais adiante.

No exemplo abaixo, declaramos a variável `nome` do tipo `String` e `idade` do tipo `int`, e depois imprimimos na tela uma mensagem, concatenando o nome e a idade mais outros textos para formar uma frase completa.

```
String nome = "Maria";
int idade = 30;
System.out.println(nome + " tem " + idade + " anos");
```

O resultado na tela é "Maria tem 30 anos".

## 6.18. Recebendo entrada de dados

Até agora, criamos exemplos usando variáveis com valores definidos diretamente no código-fonte (literals), sem entrada de dados pelo usuário. Está na hora de evoluir nossos códigos e aprender como solicitar informações ao usuário durante a execução do programa.

A forma mais simples de fazer isso em Java é usando uma classe chamada `Scanner`. Por enquanto, não se preocupe com o que é uma classe e nem o que é essa tal de `Scanner`. Concentre-se apenas em como deve ser feito para obter o resultado esperado e tenha paciência, pois você aprenderá tudo sobre classes nos capítulos sobre orientação a abjetos.

```
import java.util.Scanner;

public class EntradaDeDados {

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

        System.out.println("Digite seu nome");
        String nome = entrada.nextLine();

        System.out.println("Oi " + nome);
    }
}
```

O exemplo acima solicita a entrada de dados e aguarda uma resposta do usuário.

Ao digitar alguma informação (no caso, o seu nome), o valor digitado é atribuído à variável `nome` e depois impresso uma mensagem com o nome concatenado.

A screenshot of a terminal window titled "estudo — bash — 80x12". The prompt is "thiagofa:estudo thiago\$". The user enters "java EntradaDeDados". The program outputs "Digite seu nome", then "Thiago" (the user's input), and finally "Oi Thiago". The prompt returns to "thiagofa:estudo thiago\$".

```
thiagofa:estudo thiago$ java EntradaDeDados
Digite seu nome
Thiago
Oi Thiago
thiagofa:estudo thiago$
```

Para usar a classe `Scanner`, temos que importá-la em nosso código-fonte. Para fazer isso, basta incluir a linha abaixo no topo do arquivo do código-fonte.

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

Novamente, não há motivos para se preocupar, pois você aprenderá sobre importações mais adiante. Apenas faça isso!

Declaramos uma variável nomeada como entrada do tipo Scanner. Esse tipo não é um tipo primitivo, mas não precisa nem dizer que você aprenderá sobre isso mais pra frente, certo?

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

Finalmente, para solicitar a entrada de um texto, invocamos o método `nextLine()` através da variável `entrada`.

```
String nome = entrada.nextLine();
```

Neste momento, o programa ficará paralisado aguardando o usuário digitar alguma coisa. Quando a tecla *Enter* for pressionada, o conteúdo informado será atribuído à variável `nome`.

A classe Scanner possui vários métodos para obter dados já convertidos em tipos específicos. No segundo exemplo, usamos dois novos métodos para obter entrada de dados dos tipos `int` e `double`. O programa solicitará, além do nome do usuário, o peso e altura e fará o cálculo do Índice de Massa Corporal (IMC).

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

System.out.print("Nome: ");

// obtém entrada do tipo String
String nome = entrada.nextLine();

System.out.print("Peso: ");

// obtém entrada do tipo int
int peso = entrada.nextInt();

System.out.print("Altura: ");

// obtém entrada do tipo double
double altura = entrada.nextDouble();

double imc = peso / (altura * altura);

System.out.println("IMC de " + nome + ": " + imc);
```

Usamos `System.out.print` (e não `println`) para exibir as mensagens que solicitam dados ao usuário. A diferença de `print` com `println` é que o primeiro não exibe quebra de linha. Dessa forma, o usuário pode digitar os dados na frente da mensagem.

Não tente chamar o método `nextLine()` depois de ter chamado `nextInt()` ou `nextDouble()` através da mesma variável do `Scanner`, pois isso não funciona bem. Se precisar, crie duas variáveis do tipo `Scanner` para solucionar esse problema.

## 6.19. Operadores de comparação e igualdade

Operadores de comparação e de igualdade são usados para realizar comparações de conteúdos de variáveis ou valores literais. Os resultados das comparações sempre resultam em valores booleanos.

Os operadores de comparação disponíveis são `>` (maior), `>=` (maior ou igual), `<` (menor) e `<=` (menor ou igual), e os operadores de igualdade são `==` (igual) e `!=` (diferente). O exemplo abaixo usa todos estes operadores.

```
boolean maior = b > a; // 'b' é maior que 'a'?
boolean maiorOuIgual = b >= a; // 'b' é maior ou igual a 'a'?
boolean menor = a < b; // 'a' é menor que 'b'?
boolean menorOuIgual = a <= 10; // 'a' é menor ou igual a '10'?
boolean igual = a == b - c; // 'a' é igual a 'b' menos 'c'?
boolean diferente = a != c; // 'a' é diferente de 'c'?

System.out.println(maior);
System.out.println(maiorOuIgual);
System.out.println(menor);
System.out.println(menorOuIgual);
System.out.println(igual);
System.out.println(diferente);
```

Coincidentemente, todas as comparações acima resultam no valor booleano `true` (verdadeiro). Perceba que na quarta linha incluímos um valor literal para fazer a comparação, e na quinta linha fizemos uma operação aritmética subtraindo duas variáveis antes de realizar a comparação. Fizemos isso só para exercitarmos um pouco mais a linguagem Java.

Já que falamos de operadores de igualdade, vale a pena falarmos do operador unário `!` (ponto de exclamação). Este operador serve para negar um valor booleano ou uma expressão booleana.

Se tivermos uma variável booleana com `true` atribuído a ela, podemos negar a própria variável para o valor passar a valer `false`.

```
boolean bloqueado = true;
bloqueado = !bloqueado; // passa a valer false
```

Podemos também negar uma expressão que possui um operador de comparação. Por exemplo:

```
boolean resultado = !(b > a); // ruim, mas válido
```

O código acima compara se *b* é maior que *a* e inverte o resultado. É o mesmo que fazer *b <= a*, porém com uma legibilidade um pouco pior (pois exige mais raciocínio para entender o código).

## 6.20. Estruturas de controle if, else if e else

Qualquer bom programa deve ser capaz de tomar decisão durante sua execução, seguindo caminhos diferentes de acordo a lógica do sistema.

Como em muitas linguagens de programação, em Java, usamos a instrução *if* (se) para controlar um fluxo básico.

No exemplo abaixo, melhoramos um pouco o código-fonte que calcula o IMC de uma pessoa para exibir uma mensagem caso seu peso esteja abaixo do ideal. Para saber se o peso está abaixo do ideal, precisamos verificar se o resultado do cálculo do IMC é menor que 18.5.

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

System.out.print("Nome: ");
String nome = entrada.nextLine();

System.out.print("Peso: ");
int peso = entrada.nextInt();

System.out.print("Altura: ");
double altura = entrada.nextDouble();

double imc = peso / (altura * altura);

if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
}
```

Veja que usamos uma instrução *if* para comparar se a variável *imc* é menor que 18.5, através da expressão *imc < 18.5*.

```
if (imc < 18.5) {
    System.out.println("Abaixo do peso ideal.");
}
```

A instrução `if` deve receber uma expressão booleana agrupada por parênteses.

No exemplo acima, a chamada de `System.out.println` só será executada caso o valor da variável `imc` seja menor que 18.5.

Se o cálculo do IMC não for menor que 18.5, queremos que uma nova condição seja verificada para identificar se o peso do usuário é o ideal. Para isso, usamos a instrução `else if` (senão se), que recebe uma expressão booleana e é analisada apenas se a primeira instrução `if` for falsa.

```
if (imc < 18.5) {  
    System.out.println("Abaixo do peso ideal.");  
} else if (imc < 25) {  
    System.out.println("Peso ideal.");  
}
```

Agora, caso o IMC do usuário não seja menor que 18.5, verificamos se é menor que 25 (ou seja, maior que 18.5 e menor que 25) e imprimimos outra mensagem dizendo que o peso é ideal.

Finalizamos nosso programa incluindo outras condições que indicam o grau de obesidade de uma pessoa. Veja no último caso que usamos a instrução `else` (caso contrário ou senão). Caso nenhuma das condições expressadas no `if` e nos `else ifs` forem verdadeiras, o bloco de código de `else` será executado.

```
if (imc < 18.5) {  
    System.out.println("Abaixo do peso ideal.");  
} else if (imc < 25) {  
    System.out.println("Peso ideal.");  
} else if (imc < 30) {  
    System.out.println("Acima do peso.");  
} else if (imc < 35) {  
    System.out.println("Obesidade grau I.");  
} else if (imc < 40) {  
    System.out.println("Obesidade grau II.");  
} else {  
    System.out.println("Obesidade grau III.");  
}
```

A instrução `else` não recebe nenhuma expressão booleana, pois ela será executada apenas se todas as instruções anteriores forem falsas.

## 6.21. Programar ifs sem abrir e fechar blocos é legal?

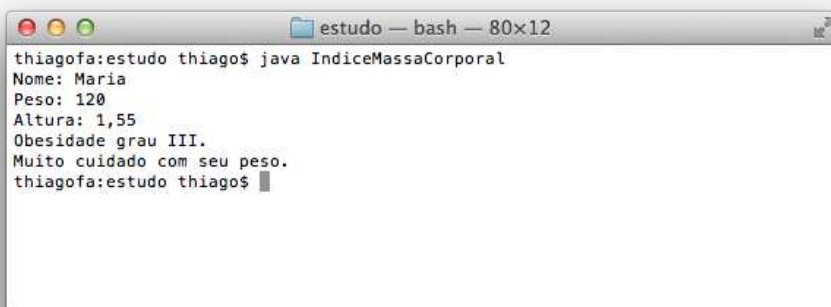
Existe a possibilidade modificar o código-fonte do último exemplo programado sem mudar o comportamento do programa, eliminando as aberturas e fechamentos de blocos (chaves).

```
if (imc < 18.5)
    System.out.println("Abaixo do peso ideal.");
else if (imc < 25)
    System.out.println("Peso ideal.");
else if (imc < 30)
    System.out.println("Acima do peso.");
else if (imc < 35)
    System.out.println("Obesidade grau I.");
else if (imc < 40)
    System.out.println("Obesidade grau II.");
else
    System.out.println("Obesidade grau III.");
```

O resultado é o mesmo, e, apesar de parecer que o código está mais limpo e bonito, existe um risco muito grande quando programamos assim. Imagine se precisássemos adicionar uma nova mensagem a ser exibida para o usuário, quando o grau de obesidade for III. Poderíamos tentar incluir uma linha que deve ser executada na instrução else.

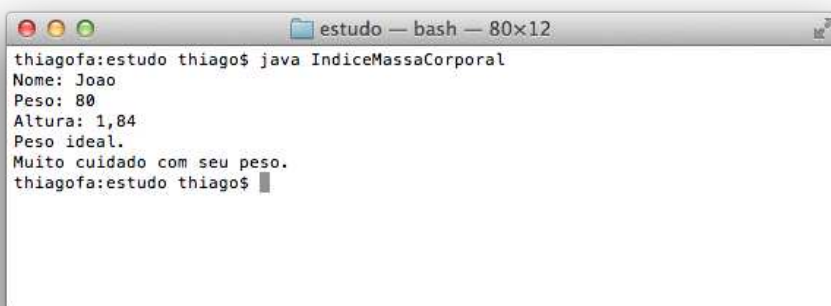
```
if (imc < 18.5)
    System.out.println("Abaixo do peso ideal.");
else if (imc < 25)
    System.out.println("Peso ideal.");
else if (imc < 30)
    System.out.println("Acima do peso.");
else if (imc < 35)
    System.out.println("Obesidade grau I.");
else if (imc < 40)
    System.out.println("Obesidade grau II.");
else
    System.out.println("Obesidade grau III.");
    System.out.println("Muito cuidado com seu peso.");
```

Ao executar esse programa, se o IMC for acima de 40 (nível máximo de obesidade), as mensagens aparecerão como esperado, como se o nosso sistema estivesse funcionando corretamente.

A terminal window titled 'estudo — bash — 80x12' showing the execution of a Java program. The program outputs the name 'Maria', weight '120', height '1,55', and BMI 'Obesidade grau III.', followed by the message 'Muito cuidado com seu peso.'

```
thiagofa:estudo thiago$ java IndiceMassaCorporal
Nome: Maria
Peso: 120
Altura: 1,55
Obesidade grau III.
Muito cuidado com seu peso.
thiagofa:estudo thiago$
```

Agora, se executamos o programa novamente (sem fazer qualquer alteração) e informarmos um peso ideal para uma determinada altura, veja que algo de errado acontece.

A terminal window titled 'estudo — bash — 80x12' showing the execution of the same Java program. The program outputs the name 'Joao', weight '80', height '1,84', and BMI 'Peso ideal.', followed by the message 'Muito cuidado com seu peso.'

```
thiagofa:estudo thiago$ java IndiceMassaCorporal
Nome: Joao
Peso: 80
Altura: 1,84
Peso ideal.
Muito cuidado com seu peso.
thiagofa:estudo thiago$
```

A mensagem "Muito cuidado com seu peso" apareceu para alguém que tem peso ideal, e essa não era nossa intenção.

O problema é que, quando não usamos as chaves para abrir e fechar blocos, nos enganamos facilmente. Sem as chaves, cada instrução `if`, `else if` ou `else` faz referência apenas para a primeira instrução logo em seguida.

O último exemplo é como se tivéssemos feito:

```
...
else
```



```
System.out.println("Obesidade grau III.");  
  
System.out.println("Muito cuidado com seu peso.");
```

Ou ainda:

```
...  
else {  
    System.out.println("Obesidade grau III.");  
}  
  
System.out.println("Muito cuidado com seu peso.");
```

Por existir essa "facilidade" de errar, é recomendado que você evite programar if/else/else if sem usar blocos.

## 6.22. Escopo de variáveis

O escopo de variáveis define em qual parte do programa a variável pode ser referenciada através de seu nome.

Para usarmos uma variável, ela deve ser declarada antes, mas só isso não é suficiente. Deve-se observar se a variável não foi declarada em um bloco mais interno que o código que está tentando usá-la. O código-fonte abaixo é um exemplo que nem mesmo compila:

```
Scanner entrada = new Scanner(System.in);  
  
System.out.print("Idade: ");  
int idade = entrada.nextInt();  
boolean podeDirigir = idade >= 18;  
  
if (!podeDirigir) {  
    System.out.print("Nome do pai: ");  
  
    // variável nomePai está sendo declarada apenas para o  
    // o bloco deste if  
    String nomePai = entrada.next();  
}  
  
System.out.println("Voce pode dirigir? ");  
  
if (podeDirigir) {  
    System.out.println("Sim, claro.");  
} else {  
    // não compila! variável nomePai não está acessível  
    System.out.println("Nao, se fizer isso, seu pai "
```

```
        + nomePai + " vai preso.");
    }
```

O código acima não compila porque a variável `nomePai` ficaria visível apenas para o bloco do `if (!podeDirigir) { ... }`.

```
$ java ExemploEscopoVariaveis
ExemploEscopoVariaveis.java:26: error: cannot find symbol
    System.out.println("Nao, se fizer isso, seu pai " + nomePai + ...
                                                ^
    symbol:   variable nomePai
    location: class ExemploEscopoVariaveis
1 error
```

Quando criamos variáveis em Java, elas ficam acessíveis apenas dentro do bloco o qual ela foi declarada e seus sub-blocos.

Para o exemplo anterior compilar, precisaríamos declarar a variável `nomePai` fora do bloco do `if`, deixando-a visível para todo o método `main` do programa.

```
String nomePai = "";

if (!podeDirigir) {
    System.out.print("Nome do pai: ");

    nomePai = entrada.next();
}
```

Agora o programa pode ser compilado e executado.

```
$ java ExemploEscopoVariaveis
Idade: 14
Nome do pai: Manoel
Voce pode dirigir?
Nao, se fizer isso, seu pai Manoel vai preso.
```

## 6.23. Operadores lógicos

Nos últimos exemplos, usamos as regras da Organização Mundial de Saúde para calcular o IMC. Existem outras regras mais detalhadas, como as da NHANES II survey (USA 1976-1980), que indicam os seguintes critérios para adultos:

Condição	IMC em mulheres	IMC em homens
Abaixo do peso	Menor que 19.1	Menor que 20.7

No peso ideal	Entre 19.1 e 25.8	Entre 20.8 e 26.4
Um pouco acima do peso	Entre 25.9 e 27.3	Entre 26.5 e 27.8
Acima do peso ideal	Entre 27.4 e 32.3	Entre 27.9 e 31.1
Obeso	Maior que 32.3	Maior que 31.1

Como você pode ver, essas regras são um pouco mais complicadas, pois usam critérios diferentes entre homens e mulheres. Apesar disso, um programador experiente não deve sentir dificuldade em programá-las.

Existem diversas formas de codificar um programa com esses critérios. Uma forma seria aninhar (agrupar) ifs, colocando um bloco dentro de outro.

```
System.out.print("Nome: ");
String nome = entrada.nextLine();

System.out.print("Peso: ");
int peso = entrada.nextInt();

System.out.print("Altura: ");
double altura = entrada.nextDouble();

System.out.print("Sexo (1 para 'M' ou outro numero para 'F'): ");
char sexo = entrada.nextShort() == 1 ? 'M' : 'F';

double imc = peso / (altura * altura);

if (sexo == 'F') {
    if (imc < 19.1) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 25.8) {
        System.out.println("Peso ideal.");
    }

    // e continua...
} else {
    if (imc < 20.7) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 26.4) {
        System.out.println("Peso ideal.");
    }

    // e continua...
}
```

No exemplo acima, pedimos que o usuário informe seu sexo, sendo que deve ser digitado o número 1 para que o sistema entenda que o sexo é masculino ou qualquer outro código se for feminino.

```
System.out.print("Sexo (1 para 'M' ou outro numero para 'F'): ");
char sexo = entrada.nextShort() == 1 ? 'M' : 'F';
```

Quando colocamos ifs dentro de ifs, estamos dizendo que o if mais interno só será avaliado caso o primeiro seja verdadeiro.

```
if (sexo == 'F') {

    // só será avaliado se sexo for Masculino
    if (imc < 19.1) {
        System.out.println("Abaixo do peso.");
    } else if (imc <= 25.8) {
        System.out.println("Peso ideal.");
    }

    // e continua...
}
```

Antes que nos esqueçamos, estamos em um tópico sobre operadores lógicos! O último exemplo nada tem haver com isso, mas foi apenas uma motivação para falarmos agora sobre esses operadores.

Ao invés de incluímos ifs dentro de ifs, podemos usar o operador lógico "E", que é representado por &&.

```
if (sexo == 'F' && imc < 19.1) {
    System.out.println("Abaixo do peso.");
} else if (sexo == 'F' && imc <= 25.8) {
    System.out.println("Peso ideal.");
} else if (sexo == 'F' && imc <= 27.3) {
    System.out.println("Um pouco acima do peso.");
} else if (sexo == 'F' && imc <= 32.3) {
    System.out.println("Acima do peso ideal.");
} else if (sexo == 'F') {
    System.out.println("Obeso.");
} else if (sexo == 'M' && imc < 20.7) {
    System.out.println("Abaixo do peso.");
} else if (sexo == 'M' && imc <= 26.4) {
    System.out.println("Peso ideal.");
} else if (sexo == 'M' && imc <= 27.8) {
    System.out.println("Um pouco acima do peso.");
} else if (sexo == 'M' && imc <= 31.1) {
    System.out.println("Acima do peso ideal.");
} else if (sexo == 'M') {
    System.out.println("Obeso.");
}
```

O operador lógico && avalia as expressões do lado esquerdo e direito e retorna apenas um resultado booleano. Para a expressão completa ser verdadeira, tanto o lado direito

como o lado esquerdo devem ser verdadeiras, mas para que a expressão completa seja falsa, pelo menos um lado deve ser falso.

Ainda existe o operador lógico "OU", representado por `||`. Podemos ainda mudar o código-fonte de nosso exemplo para usá-lo.

```
if ((sexo == 'F' && imc < 19.1) || (sexo == 'M' && imc < 20.7)) {  
    System.out.println("Abaixo do peso.");  
} else if ((sexo == 'F' && imc <= 25.8)  
    || (sexo == 'M' && imc <= 26.4)) {  
    System.out.println("Peso ideal.");  
} else if ((sexo == 'F' && imc <= 27.3)  
    || (sexo == 'M' && imc <= 27.8)) {  
    System.out.println("Um pouco acima do peso.");  
} else if ((sexo == 'F' && imc <= 32.3)  
    || (sexo == 'M' && imc <= 31.1)) {  
    System.out.println("Acima do peso ideal.");  
} else {  
    System.out.println("Obeso.");  
}
```

Veja que, usando o operador `||`, conseguimos diminuir bastante a quantidade de linhas de programação, porém deixamos as expressões booleanas dos `ifs` mais complexas, pois agrupamos duas expressões que usam o operador `&&` dentro de outra que usa o operador `||`.

O operador lógico `||` avalia as expressões dos dois lados e retorna um único resultado booleano. Para que a expressão completa seja verdadeira, pelo menos um lado deve ser verdadeiro.

Não existe uma regra geral que define o que é melhor, usar `ifs` dentro de `ifs` ou aumentar a complexidade das expressões booleanas. Normalmente, o que for mais simples de ser lido é o melhor. Com orientação a objetos (que iremos estudar adiante), seria possível deixar esse código muito mais elegante e legível.

## 6.24. Estrutura de controle switch

A estrutura de controle `switch` pode receber um valor do tipo primitivo `byte`, `short`, `char` e `int`, um tipo de enumeração (que você aprenderá em outro capítulo), `String` e outros tipos *wrappers* (que também abordaremos em outro capítulo) que envolvem tipos primitivos, como o `Character`, `Byte`, `Short` e `Integer`.

O comando switch pode ter vários possíveis caminhos de decisão (casos). O primeiro caso que estiver de acordo com o valor passado para o switch inicia a execução das instruções do caso.

Vamos construir um programa que solicita o número final da placa de um veículo ao usuário e informa a data do vencimento do IPVA. Por enquanto, para simplificar, vamos programar apenas as opções de placas que terminam com os números "1" e "2".

```
public class PagamentoIpva {  
  
    public static void main(String[] args) {  
        Scanner entrada = new Scanner(System.in);  
  
        System.out.print("Digite o final da placa: ");  
        int finalPlaca = entrada.nextInt();  
  
        switch (finalPlaca) {  
            case 1:  
                System.out.println("Vencimento dia 11 de Janeiro");  
            case 2:  
                System.out.println("Vencimento dia 12 de Janeiro");  
        }  
    }  
}
```

Se executarmos o exemplo acima e digitarmos o número "2", visualizamos na tela a data de vencimento, como esperávamos.

```
$ java PagamentoIpva  
Digite o final da placa: 2  
Vencimento dia 12 de Janeiro
```

Agora vamos executar o programa novamente e informar o número "1". Veja o resultado:

```
$ java PagamentoIpva  
Digite o final da placa: 1  
Vencimento dia 11 de Janeiro  
Vencimento dia 12 de Janeiro
```

O programa executou as instruções do primeiro e segundo caso!

Isso não é um erro, mas um detalhe sobre o funcionamento do switch. Quando um caso encontra o valor igual ao passado para o switch, todas as instruções a partir desse caso são executadas, independente se os valores dos casos não têm nada haver com o valor do switch.

Se quisermos evitar isso, precisamos incluir a instrução `break` no fim de cada caso. Assim, a execução do `switch` é interrompida e as demais linhas de outros casos não são executadas.

```
switch (finalPlaca) {  
    case 1:  
        System.out.println("Vencimento dia 11 de Janeiro");  
        break;  
    case 2:  
        System.out.println("Vencimento dia 12 de Janeiro");  
        break;  
}
```

Agora podemos compilar e executar o programa novamente.

```
$ java PagamentoIpva  
Digite o final da placa: 1  
Vencimento dia 11 de Janeiro
```

Nada acontece se informarmos um número que não existe um caso para ele (todos os casos são ignorados). Se precisarmos executar alguma coisa quando nenhum caso for encontrado, podemos usar o `default`.

O `default` é um bloco padrão, que é executado quando nenhum caso é satisfeito. Geralmente o colocamos no final do `switch`, mas pode ser incluído em qualquer ordem.

```
switch (finalPlaca) {  
    case 1:  
        System.out.println("Vencimento dia 11 de Janeiro");  
        break;  
    case 2:  
        System.out.println("Vencimento dia 12 de Janeiro");  
        break;  
    default:  
        System.out.println("Vencimento sem data, desculpe.");  
}
```

Ao compilarmos e executarmos o programa e digitar um número que não satisfaz nenhum caso, por exemplo, "9", veja que agora o caso `default` é processado.

```
$ java PagamentoIpva  
Digite o final da placa: 9  
Vencimento sem data, desculpe.
```

## 6.25. Operador ternário

O algoritmo abaixo solicita ao usuário a idade e exibe uma mensagem dizendo se a idade é classificada como adulto ou criança/adolescente. Para fazer isso, usamos apenas o que aprendemos até agora.

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

System.out.print("Digite sua idade: ");
int idade = entrada.nextInt();

String indicacao = "";

if (idade >= 18) {
    indicacao = "adulto";
} else {
    indicacao = "criança/adolescente";
}

System.out.println("Resultado: " + indicacao);
```

O operador ternário é uma forma simples de reduzir linhas de código com instruções if. O operador permite que um valor seja atribuído a uma variável baseado em uma expressão booleana.

O exemplo abaixo faz o mesmo que o último código-fonte, porém usando o operador ternário:

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

System.out.print("Digite sua idade: ");
int idade = entrada.nextInt();

String indicacao = (idade >= 18) ? "adulto" : "criança/adolescente";

System.out.println("Resultado: " + indicacao);
```

O operador ternário possui a seguinte sintaxe:

expressão booleana ? valor caso verdadeiro : valor caso falso;

Também é possível usar operadores ternários aninhados. O exemplo abaixo atribui à variável `indicacao` o valor "adulto", "criança" ou "adolescente", dependendo do valor da variável `idade`.

```
String indicacao = (idade >= 18) ? "adulto"
    : (idade <= 12 ? "criança" : "adolescente");
```



O uso de operadores ternários aninhados pode deixar o código ilegível, por isso, para manter sua boa reputação, não é recomendado que se faça isso.

## 6.26. Operadores de incremento e decremento

Os operadores de incremento e decremento também são conhecidos como acréscimo e decréscimo. Esses operadores aumentam ou diminuem o valor de uma variável em exatamente uma unidade. Eles são compostos por dois sinais ++ (adição) ou -- (subtração).

O exemplo abaixo incrementa a variável `idade` em uma unidade, mudando o valor de 10 para 11.

```
int idade = 10;
idade++;
```

A instrução `idade++` é uma versão mais simples dos exemplos abaixo:

```
// é o mesmo que
idade = idade + 1;
```

```
// e também
idade += 1;
```

É importante entender que a forma pós-fixada (com ++ depois do nome da variável) incrementa o valor após usar a variável.

No exemplo abaixo, a variável `novaIdade` será atribuída com o valor 10 e logo em seguida o valor da variável `idade` será incrementado em uma unidade, mudando seu valor para 11.

```
int idade = 10;

// novaIdade recebe 10 e idade é incrementada para 11
int novaIdade = idade++;
```

Existe também a forma pré-fixada (com ++ antes do nome da variável), que incrementa o valor antes de usar a variável.

No próximo exemplo, a variável `idade` será incrementada e terá o valor igual a 11 antes de atribuir um valor para `novaIdade`, que também ficará com valor 11.

```
int idade = 10;
```

```
// idade é incrementada para 11 e novaIdade recebe 11
int novaIdade = ++idade;
```

Assim como existe o operador de incremento, você pode usar o operador de decremento para diminuir o valor da variável em uma unidade, podendo ser da forma pós-fixa ou pré-fixa.

```
int idade = 10;

// novaIdade recebe 10 e idade é decrementada para 9
int novaIdade = idade--;

// idade é decrementada para 8 e outra recebe 8
int outraIdade = --idade;
```

Os operadores de incremento e decremento são usados principalmente para controlar loops (laços), onde é necessário um contador para saber quantas vezes o bloco do laço foi executado.

## 6.27. Estrutura de controle while

O while é uma das formas de fazer loops (laços) em Java. Loops são usados para executar um bloco de código diversas vezes, dependendo de uma condição ser verdadeira.

O exemplo abaixo imprime todos os números em um intervalo informado pelo usuário.

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

System.out.print("Digite o numero inicial: ");
int numeroInicial = entrada.nextInt();

System.out.print("Digite o numero final: ");
int numeroFinal = entrada.nextInt();

int numeroAtual = numeroInicial;

while (numeroAtual <= numeroFinal) {
    System.out.println(numeroAtual);
    numeroAtual++;
}
```

O bloco de código dentro do while será executado enquanto o valor da variável numeroAtual for menor ou igual ao valor da variável numeroFinal. Veja que usamos

o operador de incremento para adicionar à variável `numeroAtual` uma unidade a cada execução do bloco de código do loop.

## 6.28. Estrutura de controle `do/while`

O loop do tipo `do/while` é parecido com o `while`, com a diferença que a condição do laço é testada somente após a execução do bloco de código. Por isso, o bloco de código do loop é executado pelo menos uma vez.

O exemplo abaixo solicita ao usuário a entrada de um número, sendo que se for digitado `0`, o laço é encerrado (de acordo com a condição), mas se outro número for digitado, uma somatória é feita e o programa imprime o valor acumulado.

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

int valor = 0;
int soma = 0;

do {
    System.out.print("Digite 0 para sair ou qualquer numero para "
        + "somar: ");
    valor = entrada.nextInt();

    soma += valor;
    System.out.println("Soma: " + soma);
} while (valor != 0);
```

Ao compilar e imprimir o programa, note que o bloco de código do `do/while` é executado pelo menos uma vez. Na execução abaixo, informamos o número `0` e saímos do laço imediatamente.

```
$ java ExemploDoWhile
Digite 0 para sair ou qualquer numero para somar: 0
Soma: 0
```

Na execução a seguir, informamos vários números antes de digitar o valor `0`, por isso o laço foi executado diversas vezes.

```
$ java ExemploDoWhile
Digite 0 para sair ou qualquer numero para somar: 10
Soma: 10
Digite 0 para sair ou qualquer numero para somar: 3
Soma: 13
Digite 0 para sair ou qualquer numero para somar: 14
Soma: 27
Digite 0 para sair ou qualquer numero para somar: 1
```

Soma: 28  
Digite 0 para sair ou qualquer numero para somar: 0  
Soma: 28

## 6.29. Estrutura de controle for

Uma das estruturas de controle mais usadas para fazer laços é o `for`, pois ela fornece uma forma fácil de iterar (percorrer) em um intervalo de valores. A forma básica da instrução `for` é a seguinte:

```
for (inicição; condição; incremento) {  
    // bloco de código do loop  
}
```

A expressão de iniciação é executada uma única vez assim que o loop é iniciado. Normalmente é declarada uma variável de controle com um valor inicial.

A expressão de condição é semelhante às outras estruturas de controle. Você deve incluir uma expressão booleana que definirá a permanência ou término do laço. Essa expressão é analisada a cada iteração do loop.

A última expressão, de incremento, é executada após cada iteração do laço, e normalmente é usada para modificar o valor da variável de controle, seja incrementando ou decrementando.

Vamos a um exemplo:

```
Scanner entrada = new Scanner(System.in);  
  
System.out.print("Ultimo numero: ");  
int numeroFinal = entrada.nextInt();  
  
for (int i = 1; i <= numeroFinal; i++)  
    System.out.println(i);  
}
```

O código acima executa o laço `for` e imprime números de 1 até um valor informado pelo usuário. Note que uma variável de controle `i` foi declarada na seção de iniciação, avaliada na condição e incrementada na seção de incremento.

## 6.30. Cláusulas break e continue

Quando trabalhamos com loops, pode surgir a necessidade de abandonar a execução do laço antes mesmo que a condição booleana seja falsa. Neste caso, devemos usar a cláusula `break`.

O exemplo abaixo solicita ao usuário um número (divisor) e imprime na tela de 100 até um número anterior ao primeiro múltiplo do valor informado (desde que seja até o número 200).

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

System.out.print("Digite um numero: ");
int divisor = entrada.nextInt();

for (int i = 100; i <= 200; i++) {
    if (i % divisor == 0) {
        break;
    }

    System.out.println(i);
}

System.out.println("Fim do programa.");
```

Na execução abaixo, informamos o número 8 e o programa exibiu os valores de 100 a 103, pois o número 104 é o primeiro múltiplo de 8 dentro do intervalo de 100 a 200. A cláusula `break` interrompeu o loop, mas não o programa.

```
$ java ExemploBreak
Digite um numero: 8
100
101
102
103
Fim do programa.
```

Outra necessidade que pode surgir durante o uso de laços é o abandono de apenas a iteração atual, mas não do loop. A cláusula responsável por fazer isso é a `continue`.

O exemplo abaixo é parecido com o anterior, porém ele imprime na tela todos os números entre 100 e 120, com exceção dos múltiplos do número informado.

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

System.out.print("Digite um numero: ");
int divisor = entrada.nextInt();
```

```

for (int i = 100; i <= 120; i++) {
    if (i % divisor == 0) {
        continue;
    }

    System.out.println(i);
}

System.out.println("Fim do programa.");

```

Executamos o programa e informamos o número 2. Veja que foram exibidos apenas os números ímpares (não múltiplos de 2).

```

$ java ExemploBreak
Digite um numero: 2
101
103
105
107
109
111
113
115
117
119
Fim do programa.

```

Quando você está trabalhando com um laço dentro de outro, a chamada das cláusulas `break` e `continue` abandonam o laço ou a iteração do laço mais interno, a não ser que um rótulo seja especificado.

Se tiver curiosidade, pesquise na internet sobre cláusulas `break` e `continue` rotuladas. O melhor seria você não usar essas cláusulas rotuladas e pensar em alguma lógica mais simples, pois elas podem deixar seu código-fonte difícil de ser compreendido.

# Use Eclipse e seja feliz!

## 7.1. Introdução ao Eclipse IDE

Uma IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) é um programa que possui funcionalidades para programadores construir software. Normalmente, as IDEs possuem um editor de texto com facilidades para determinada linguagem, compiladores, editores gráficos que auxiliam na criação e edição de arquivos e interfaces gráficas e um debugador para ajudar no rastreamento de erros de lógica de programação.

O Eclipse é a IDE de desenvolvimento Java mais popular no mercado. O projeto da ferramenta open source é liderado pela IBM.

O Eclipse fornece um ambiente completo para programação de aplicações em Java e C++, além de existirem vários plug-ins (componentes que adicionam novas funcionalidades) que incluem a possibilidade de trabalhar também com Python, PHP, Cobol, etc. Além de programação, o Eclipse pode dar suporte a ferramentas de controle de versão (como CVS, SVN, Git, etc), servidores de aplicação, ferramentas de testes automatizados, ferramentas CASE (para modelar, por exemplo, diagramas UML ou diagramas de entidades e relacionamentos), etc. O Eclipse funciona em diferentes sistemas operacionais, como por exemplo, Windows, Linux, Mac OS, etc.

Aprenderemos a instalar e usar o Eclipse, e, a partir de agora, todos os nossos exemplos e exercícios serão feitos usando esta IDE.

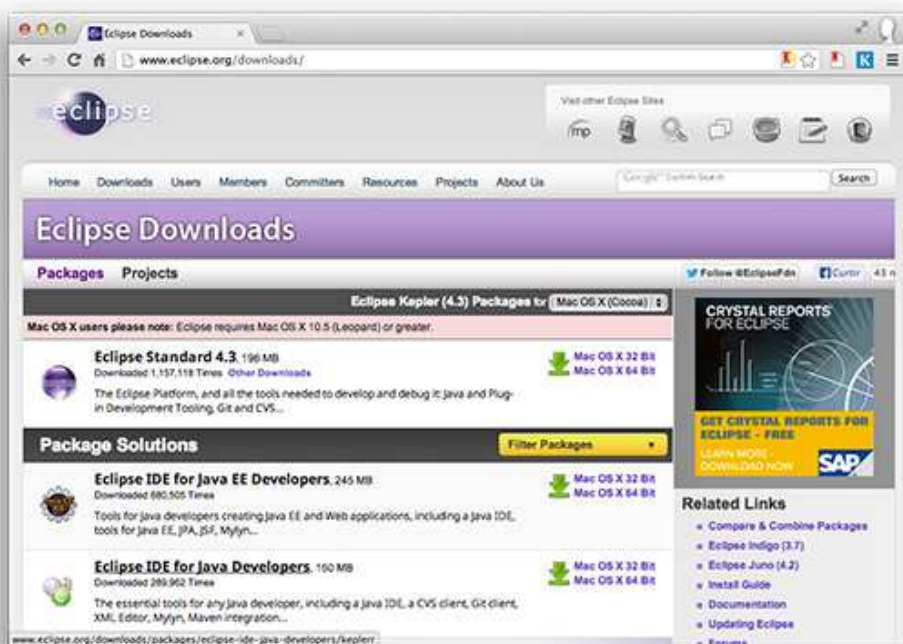
Além do Eclipse, existem várias outras IDEs para desenvolvimento em Java, como o NetBeans ([www.netbeans.org](http://www.netbeans.org)), MyEclipse ([www.myeclipseide.com](http://www.myeclipseide.com)), IntelliJ IDEA

([www.jetbrains.com/idea](http://www.jetbrains.com/idea)), JDeveloper ([www.oracle.com/technetwork/developer-tools/jdev](http://www.oracle.com/technetwork/developer-tools/jdev)), etc.

## 7.2. Baixando e instalando o Eclipse

O Eclipse é distribuído através de vários pacotes. Cada pacote dá suporte a um tipo de trabalho, como por exemplo, para programação em Java, Java EE, PHP, C++, etc. A diferença entre eles é basicamente os plugins pré-configurados. Nada impede que você baixe uma versão mais simples e instale os plugins que dão suporte a outras funcionalidades.

Antes de instalar o Eclipse, precisamos baixá-lo para nosso computador. Acesse [www.eclipse.org](http://www.eclipse.org) e clique no link "Downloads". Se tiver interesse em desenvolvimento Java EE (para criar páginas na web dinâmicas, por exemplo), baixe o *Eclipse IDE for Java EE Developers*, caso contrário, baixe o *Eclipse IDE for Java Developers*.



A instalação do Eclipse é muito simples. Basta descompactar o arquivo baixado em uma pasta de seu computador.



## 7.3. Iniciando o Eclipse

Para iniciar o Eclipse, localize e execute o arquivo *eclipse.exe* (se for Windows), *eclipse* (se for Linux) ou *Eclipse.app* (se estiver usando Mac OS) da pasta *eclipse*. Você já deve ter a JDK instalada.

Ao iniciar o Eclipse, você será perguntado sobre qual workspace deseja usar. Workspace é um conceito do Eclipse que permite você armazenar seus projetos e preferências do ambiente de desenvolvimento em um diretório específico. Você pode ter vários workspaces em seu computador para, por exemplo, separar seus projetos pessoais dos projetos profissionais ou para separar projetos de diferentes clientes.


Quando você abrir o Eclipse pela primeira vez, a tela de boas vindas será exibida. Feche essa tela clicando no "X" da aba "Welcome", e o ambiente de trabalho será exibido.

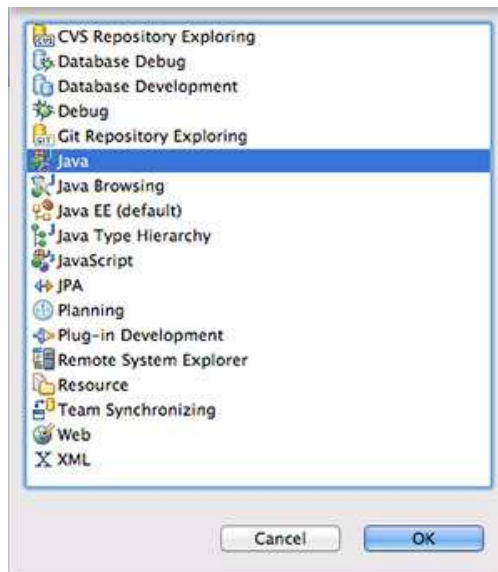


## 7.4. Ambientes de trabalho

O Eclipse possui o conceitos de *perspectives* (perspectivas) e *views* (visões) para organizar o ambiente de trabalho. Uma perspective é um conjunto de *views* e editores que formam o ambiente de trabalho dentro do Eclipse, enquanto uma *view* é um

componente visual (uma sub-janela) agregada ao ambiente do Eclipse que possui a capacidade de editar a estrutura ou realizar consultas em um projeto.

Se você tiver instalado o Eclipse para desenvolvimento Java EE, a perspectiva padrão após a instalação será a "Java EE". Para alterar a perspectiva para "Java", clique no ícone  na parte superior direita do Eclipse e selecione "Java".



O Eclipse mudará o ambiente de trabalho para incluir *views* que fazem mais sentido para o desenvolvimento Java. As janelas *Package Explorer*, *Problems*, *Javadoc* e etc são exemplos de *views*.

Se você fechar alguma *view* por engano ou desejar exibir outras *views* que não estão aparecendo no ambiente, basta acessar o menu *Window*, *Show View* e clicar na *view* desejada, ou *Other...* para conhecer outras opções.

## 7.5. Criando o primeiro projeto

Praticamente, todas as IDEs possuem o conceito de projetos. Um projeto é um agrupamento de diversas classes e arquivos que funcionam juntos para compor um software ou módulo de software completo. Até agora nós desenvolvemos programas criando apenas uma classe, mas quando iniciarmos os estudos sobre orientação a objetos, nós criaremos diversas classes que se comunicarão entre si, tudo dentro do mesmo projeto.

Agora vamos criar um projeto Java bem simples no Eclipse. Acesse o menu *File, New* e clique em *Java Project*.

A janela *New Java Project* será exibida. Digite o nome do projeto no campo *Project name*, deixe as outras opções como estão e clique no botão *Finish*.

Você verá o nome do projeto em Package Explorer, o que indica que o projeto foi criado. Clique na seta ao lado do nome do projeto para exibir o conteúdo do projeto.

Veja que dentro do projeto existe um tipo de pasta chamada "src". Essa pasta é conhecida dentro do Eclipse como uma pasta de código-fonte (*Source Folder*), pois dentro dela ficarão todos os arquivos .java (os arquivos com os códigos-fonte). Os arquivos .class (compilados) ficam em uma pasta que o Eclipse não exibe, porém ela existe também dentro do mesmo projeto, e normalmente possui o nome "bin". O item *JRE System Library* indica a JRE que estamos usando para esse projeto (cada projeto pode optar por usar uma versão de JRE diferente).

Vamos criar agora uma nova classe chamada *OláEclipse*. Clique com o botão direito do mouse sobre o projeto, acesse o menu *New* e depois *Class*.

A tela *New Java Class* será exibida. Digite o nome da classe e clique em *Finish*. Opcionalmente, você poderá selecionar a opção para criar o método *main* automaticamente.

O Eclipse criará o arquivo *OláEclipse.java* e abrirá um editor para este arquivo, incluindo uma declaração da classe *OláEclipse*. Não se preocupe com a nova palavra-chave *public* que o Eclipse incluiu na declaração, pois você aprenderá sobre ela mais adiante.

Agora podemos criar o método *main*. Uma forma bem simples de fazer isso no Eclipse é digitar a palavra *main* (ou parte dela) e pressionar *Ctrl+Espaço* e depois *Enter*.

O Eclipse irá gerar o código completo do método *main* para você.


O que acabamos de usar foi um template (modelo) de código pré-configurado no Eclipse. Você mesmo pode criar seus próprios templates, se desejar. O Eclipse é cheio de templates e teclas de atalho, que ajuda muito em nossa produtividade.

Vamos fazer uma chamada de *System.out.println* para imprimir algo na saída do programa. Digite *sysout* e pressione *Ctrl+Espaço* para usar um template de código.

É hora de salvar o arquivo `OlaEclipse.java`. Pressione *Ctrl+S* ou acesse o menu *File* e depois *Save*. É sempre bom salvar seus arquivos frequentemente, por isso, tente criar o hábito de usar o atalho.

## 7.6. Compilando e executando o programa

Por padrão, os arquivos de classes desenvolvidos no Eclipse são compilados automaticamente ao serem salvos (se não houver erros). Se você não deseja que o Eclipse faça isso, desmarque a opção *Build Automatically* no menu *Project*.

Se a opção for desmarcada, você precisará compilar manualmente através do menu *Project* e *Build All* ou *Build Project*. Para executar o programa, clique no ícone  da barra de ferramentas do Eclipse. A saída do programa será exibida na *view* chamada *Console*.

Você pode usar a tecla de atalho *Alt+Shift+X* e depois pressionar a tecla *J* para executar o programa Java. Apesar de precisar pressionar diversas teclas e parecer complicado, é bastante útil, pois podemos executar os programas sem precisar usar o mouse.

Depois que você já executou o programa uma vez, poderá usar o atalho *Ctrl+F11* para executá-lo novamente, pois este atalho memoriza a última execução feita pelo Eclipse.

## 7.7. Outras teclas de atalho

Além dos atalhos do Eclipse que você já estudou, podemos citar outros também interessantes. Use-os sem moderação.

### Windows

- *Ctrl+F* - Exibe uma tela de pesquisa/substituição de texto.
- *Ctrl+L* - Posiciona o cursor em um número de linha especificado.
- *Ctrl+Shift+F* - Formata o código-fonte de acordo com as convenções da Oracle.
- *Ctrl+Shift+O* - Organiza as importações de uma classe (muito útil quando estivermos usando diversas classes, em Orientação a Objetos).

- *Ctrl+/* - Inclui ou exclui um comentário de linha.
- *Ctrl+Shift+/* - Inclui um comentário de bloco para o código selecionado.
- *Ctrl+1* - Exibe um painel com sugestões para corrigir determinado código (normalmente usado em cima de um código que possui erro).
- *Ctrl+M* - Alterna o modo de exibição da view atual ou do editor de código-fonte para tela cheia ou normal.
- *Ctrl+PgUp* - Quando você estiver trabalhando com vários arquivos e classes, este atalho navega nas abas em direção à sua esquerda.
- *Ctrl+PgDown* - O mesmo que o atalho anterior, porém a navegação é para a direita.
- *Ctrl+O* - Abre um menu com os membros da classe para navegação rápida. Muito bom quando estivermos trabalhando com Orientação a Objetos e quisermos buscar um método ou atributo específico dentro de uma classe com vários métodos/atributos.
- *Ctrl+Shift+T* - Abre tela de pesquisa de tipos (classes). Quando você estiver desenvolvendo um software real com milhares de classes, você vai querer usar esse atalho o tempo todo.
- *Ctrl+Shift+R* - Abre tela de pesquisa de recursos (qualquer arquivo que faz parte do projeto).
- *Ctrl+3* - Exibe um menu de opções com praticamente todas as funções do Eclipse. Você precisará digitar parte do nome da funcionalidade do Eclipse que deseja ou as iniciais e pressionar *Enter* para chamá-la.

## Mac

- *Cmd+F* - Exibe uma tela de pesquisa/substituição de texto.
- *Cmd+L* - Posiciona o cursor em um número de linha especificado.
- *Cmd+Shift+F* - Formata o código-fonte de acordo com as convenções da Oracle.

- *Cmd+Shift+O* - Organiza as importações de uma classe (muito útil quando estivermos usando diversas classes, em Orientação a Objetos).
- *Cmd+/-* - Inclui ou exclui um comentário de linha.
- *Cmd+Option+/-* - Inclui um comentário de bloco para o código selecionado.
- *Cmd+I* - Exibe um painel com sugestões para corrigir determinado código (normalmente usado em cima de um código que possui erro).
- *Ctrl+M* - Alterna o modo de exibição da view atual ou do editor de código-fonte para tela cheia ou normal.
- *Ctrl+Fn+↑* - Quando você estiver trabalhando com vários arquivos e classes, este atalho navega nas abas em direção à sua esquerda.
- *Ctrl+Fn+↓* - O mesmo que o atalho anterior, porém a navegação é para a direita.
- *Cmd+O* - Abre um menu com os membros da classe para navegação rápida. Muito bom quando estivermos trabalhando com Orientação a Objetos e quisermos buscar um método ou atributo específico dentro de uma classe com vários métodos/atributos.
- *Cmd+Shift+T* - Abre tela de pesquisa de tipos (classes). Quando você estiver desenvolvendo um software real com milhares de classes, você vai querer usar esse atalho o tempo todo.
- *Cmd+Shift+R* - Abre tela de pesquisa de recursos (qualquer arquivo que faz parte do projeto).
- *Cmd+3* - Exibe um menu de opções com praticamente todas as funções do Eclipse. Você precisará digitar parte do nome da funcionalidade do Eclipse que deseja ou as iniciais e pressionar *Enter* para chamá-la.

## 7.8. Depurando códigos

Quando um programa não está funcionando corretamente (em termos de lógica de programação, e não de compilação), você poderá depurá-lo (*debug*), ou seja, acompanhar a execução passo a passo para identificar onde está o erro. O Eclipse fornece um excelente depurador que lhe permite fazer isso.

Para testar, vamos trazer o código-fonte de um exemplo anterior que faz um loop com `for`. O código não possui nenhum erro, mas queremos depurá-lo.

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

System.out.print("Digite um numero: ");
int divisor = entrada.nextInt();

for (int i = 100; i <= 120; i++) {
    if (i % divisor == 0) {
        continue;
    }

    System.out.println(i);
}

System.out.println("Fim do programa.");
```

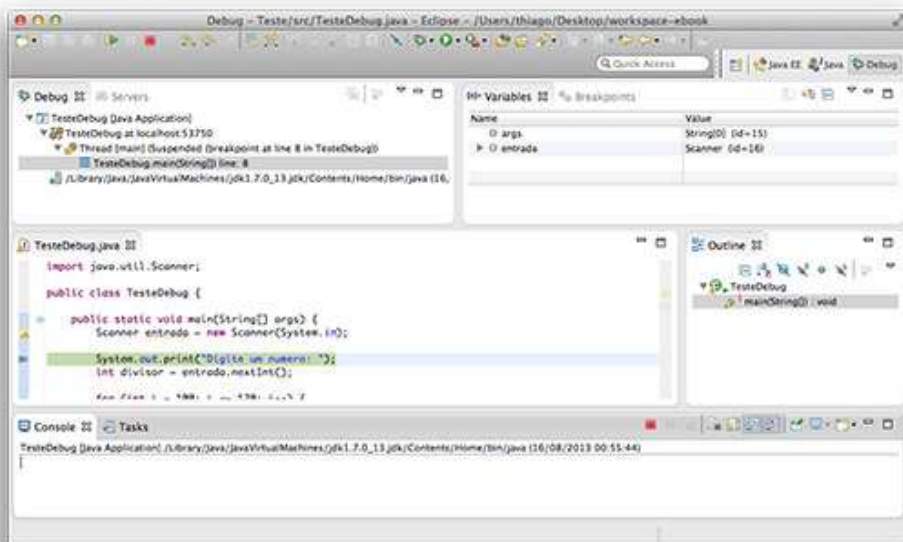
Colocamos o código-fonte acima em uma classe chamada `TesteDebug`, posicionamos o cursor na linha onde queremos que o depurador seja paralisado (para que possamos analisar com calma) e pressionamos `Ctrl+Shift+B` (ou `Cmd+Shift+B` no Mac) para adicionar um *breakpoint* (ponto de parada).

Uma bolinha azul deve aparecer na frente da linha, indicando que existe um *breakpoint*. Além da tecla de atalho, você pode dar um duplo clique na faixa azul em frente a linha que você deseja adicionar ou remover um *breakpoint*.

Agora executamos o programa em modo de *debug* clicando no ícone da barra de ferramentas do Eclipse ou usamos o atalho `Alt+Shift+D` (ou `Ctrl+Option+Cmd+D` no Mac) e depois `J`.

Se estiver usando o Windows Vista ou 7, o sistema operacional pode pedir uma permissão. Autorize para continuar. O Eclipse perguntará se você deseja mudar a perspectiva para outra específica para depuração. Clique em *Yes*.

A execução do programa será iniciada e paralisada exatamente na linha onde incluímos um *breakpoint*. Podemos ver essa linha destacada com um verde claro.



A perspectiva de depuração inclui *views* que nos ajudam acompanhar a execução do programa. A *view* chamada *Debug* exibe uma pilha de execução e os processos e *Variables* exibe todas as variáveis e seus valores no momento do *breakpoint*.

Quando a execução do programa está paralisada, podemos executá-lo passo a passo, observando o que acontece com as variáveis. Existem três funções úteis para executar um código passo a passo. Veja quais são, com seus atalhos:

- *Step Into (F5)* – entra no método da linha atual (ainda não estudamos a fundo os métodos em Java, por isso é melhor não usar essa função agora).
- *Step Over (F6)* – executa a linha atual e passa para a próxima.
- *Step Return (F7)* – executa o restante do método atual e paralisa a execução logo após (também é melhor usar depois, quando estudarmos sobre os métodos).

Ao depurar nosso programa usando o atalho *F6 (Step Over)*, podemos notar os valores das variáveis sendo alterados. O Eclipse destaca a variável com uma linha amarela imediatamente no momento que ela foi alterada, na *view Variables*.

Não se esqueça que, para voltar a programar, você deve selecionar a perspectiva "Java" novamente.



## Capítulo 8

# Introdução a orientação a objetos

### 8.1. O que é POO?

Programação Orientada a Objetos (POO) é um paradigma de programação que ajuda a definir a estrutura de programas de computadores, baseado nos conceitos do mundo real, sejam eles reais ou abstratos. A ideia é simular as coisas que existem e acontecem no mundo real no mundo virtual.

O termo foi usado pela primeira vez na linguagem Smalltalk, criada por Alan Kay, mas algumas ideias já eram usadas na década de 1960 na linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard.

Atualmente, diversas linguagens de programação utilizam este paradigma, como por exemplo: Java, VB.NET, C#, C++, Object Pascal, Ruby, Python, etc.

Dentre as vantagens que a OO proporciona, podemos destacar o aumento de produtividade, reuso de código, redução das linhas de código programadas, separação de responsabilidades, encapsulamento, polimorfismo, componentização, maior flexibilidade do sistema, etc.

A Orientação a Objetos (OO) permite criar programas componentizados, separando as partes do sistema por responsabilidades e fazendo que essas partes se comuniquem entre si, por meio de mensagens. O programador é responsável por definir como os objetos devem se comportar e como eles devem interagir entre si.

## 8.2. Classes e objetos

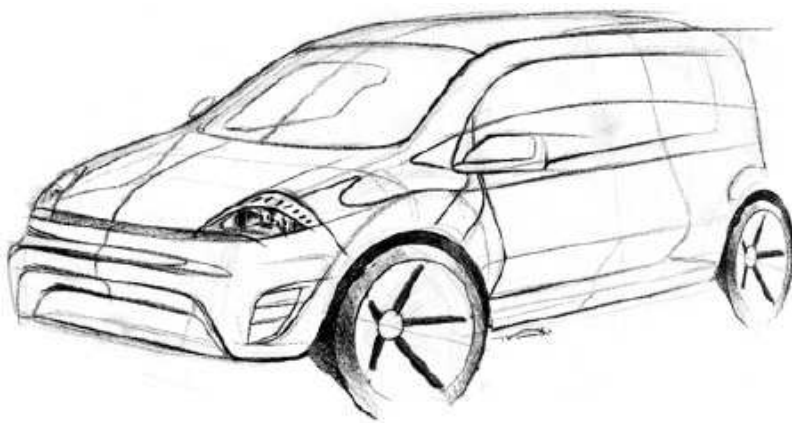
Uma classe, nada mais é do que a descrição de um conjunto de entidades (reais ou abstratas) do mesmo tipo e com as mesmas características e comportamentos. As classes definem a estrutura e o comportamento dos objetos daquele determinado tipo.

Podemos dizer que as classes são, na verdade, modelos de objetos do mesmo tipo. Por falar nisso, é comum usarmos também o termo "tipo" ao falar de "classe", apesar de que "tipo" é mais abrangente.

Para exemplificar, vamos imaginar a classe "Carro" do mundo real e trazer para o mundo virtual. Um carro tem rodas, pneus, lanternas, portas, motor, pára-choque, etc.

Qual é o modelo do carro? Qual é a cor do carro? Qual é a marca dos pneus? As lanternas estão ligadas? O motor do carro está funcionando? Não sabemos! Na verdade, não é possível responder essas perguntas, pois ao pensar na classe "Carro", nós definimos apenas **o que o carro pode ter e o que ele faz**, e não o que realmente ele é.

Para ficar mais claro, pense como se não existissem carros no mundo. Você é o grande inventor dos carros! Como qualquer invenção, você precisará fazer algum desenho, escrever um rascunho, criar algum projeto ou apenas formar uma ideia em sua cabeça de como "a coisa" deve funcionar. Isso são as classes! Ideias, modelos, protótipos, rascunhos... algo que ainda não existe, mas que, a partir da ideia, alguém pode começar a construir o produto final (que existe de verdade).



Ao criar a classe "Carro", precisamos pensar em o que a classe deve ter (características/propriedades) e o que ela poderá fazer (comportamentos/ações). Devemos pensar

apenas no essencial para que o problema que estamos tentando resolver seja solucionado.

Por exemplo, podemos pensar que o carro deve ter:

- Fabricante
- Modelo
- Cor
- Tipo de combustível
- Ano de fabricação
- Valor de mercado

E agora os comportamentos do carro (o que ele pode fazer):

- Ligar
- Desligar
- Mudar marcha
- Acelerar
- Frear

Agora já temos a classe "Carro", mas ainda não podemos ligar, acelerar ou frear um carro, e nem mesmo saber o fabricante, modelo ou a cor dele, pois ainda não existe um carro. Tudo que existe é uma ideia de um carro, uma especificação!

Nós podemos construir um ou vários carros a partir da classe "Carro", e isso se chama **instanciação**, na orientação a objetos.

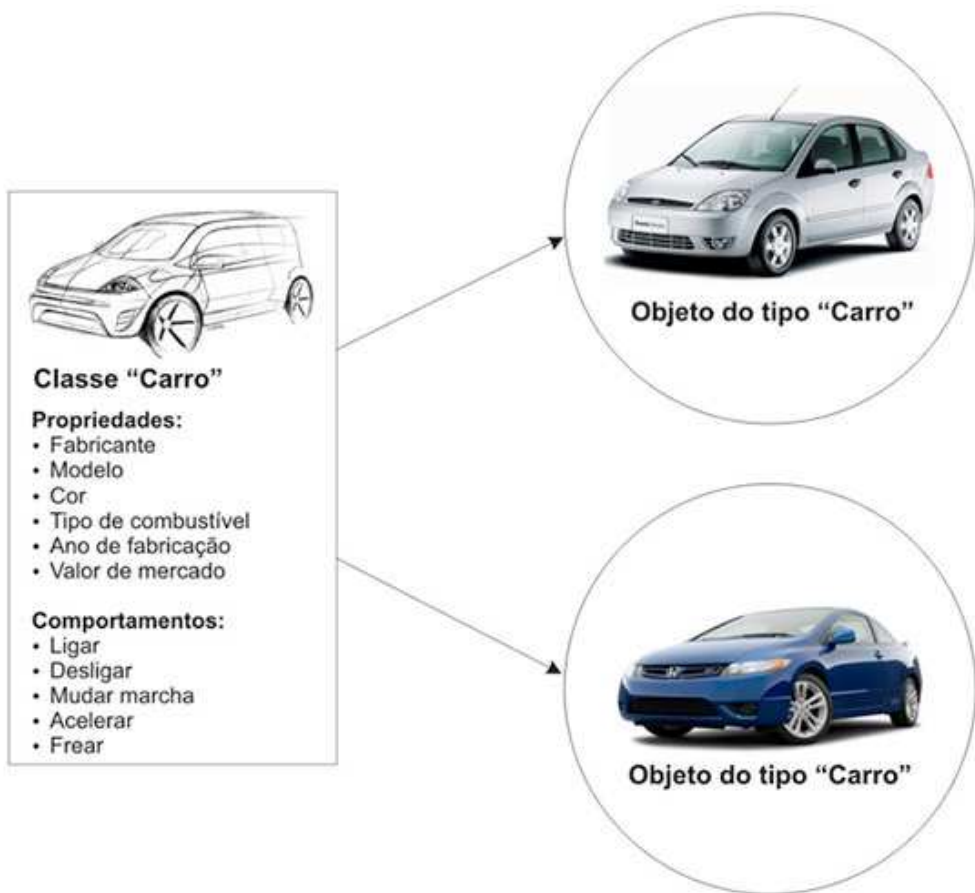


Quando usamos a classe para instanciar algo, o resultado final é um **objeto** do tipo da classe. No caso do tipo "Carro", instanciamos um objeto que, errr... é um carro, mas agora real, que ocupa lugar no espaço, que acelera, freia, tem cor, modelo, fabricante, etc.

Um objeto nada mais é que uma instância particular de um tipo de dado específico (classe), ou seja, em outras palavras, objeto é uma entidade, do mundo computacional, que representa uma entidade do mundo real especificamente.

Os objetos possuem estados (atributos ou propriedades), comportamentos (métodos ou ações) e identidade (cada objeto é único). O objeto que "instanciamos mentalmente" (pois ainda não estamos programando) é um Fiesta Sedan da Ford, tem cor prata, bicombustível e podemos pisar no acelerador e sair passeando por aí.

O Ford Fiesta que instanciamos é único no mundo! Podem existir outros muito parecidos (praticamente iguais), mas o carro da imagem acima tem uma identidade única, um chassi único, um único dono (ou dona), e o estado atual dele é só dele e de mais nenhum outro.



Os objetos se comunicam entre si por meio de mensagens (chamadas aos métodos, que veremos em breve). Por exemplo, o João (um objeto do tipo "Motorista") pode bater o carro dele (um objeto do tipo "Carro") em seu carro (que é outro objeto do tipo "Carro") se você (um objeto do tipo "Motorista") parar bruscamente seu carro na frente. Se você usar bem a imaginação (não é tão difícil assim), perceberá que você comunica com seu carro (acelerando, freando, etc), da mesma forma que o João também comunica com o carro dele, e os dois carros também podem se comunicar (através de uma colisão, por exemplo).

Podemos pensar no mundo real como um conjunto de "objetos" que interagem uns com os outros. Por exemplo, em um ambiente de negócios, o cliente interage com o fornecedor, ambos interagem com produtos, com transportadoras, notas fiscais, que também interagem com órgãos do governo, e outros milhares de objetos. A programação orientada a objetos é uma forma de criar sistemas "mapeando" as características e comportamentos de entidades do mundo real e como elas se

comunicam, porém com um escopo mais limitado (apenas o que faz sentido para o domínio do problema).

## 8.3. Criando uma classe com atributos

Agora que você já sabe o que é uma classe, vamos criar a classe Carro em Java. Crie um novo projeto no Eclipse, acesse o menu "New", clique em "Class" e crie a nova classe com o código abaixo.

```
class Carro {  
  
}
```

O código acima é referente ao tipo "Carro", mas não incluímos os atributos e métodos, por isso o carro que definimos não tem nenhuma característica e nem comportamentos. Vamos agora incluir os atributos de um carro, também conhecidos como propriedades, características ou ainda variáveis de instância. No caso do carro, os atributos que definimos são:

- Fabricante
- Modelo
- Cor
- Tipo de combustível
- Ano de fabricação
- Valor de mercado

O código Java fica da seguinte forma:

```
class Carro {  
  
    String fabricante;  
    String modelo;  
    String cor;  
    String tipoCombustivel;  
    int anoDeFabricacao;  
    double valorDeMercado;  
  
}
```

Os atributos são variáveis com o escopo do objeto. Eles são acessíveis e estão disponíveis enquanto o objeto "possuir vida", e são iniciados durante a criação de

seu objeto. Neste caso, podemos chamá-los de variáveis de instância, pois só existirão valores para as variáveis quando existirem objetos (instâncias).

Os tipos dos atributos podem ser um tipo primitivo (como `int`, `double`, etc) ou um tipo de classe. Por exemplo, `String` é uma classe (não é um tipo primitivo) que representa uma cadeia de caracteres, mas poderíamos definir um atributo do tipo `Motorista` ou `Proprietario` (se essas classes existissem) para outros atributos, se fosse o caso.

Lembre-se que a classe `Carro` que acabamos de criar não representa nenhum carro real (objeto), mas apenas especifica como os carros devem ser quando forem instanciados.

## 8.4. Instanciando objetos

O código-fonte da classe `Carro` compila com sucesso, mas não é possível executá-lo. Se você tentar executar a classe pelo Eclipse, receberá a mensagem de erro:

```
Editor does not contain a main type
```

Você pode ficar tentado a colocar um método `main()` na classe `Carro` para executá-la, mas isso é muito, mas muito errado. A classe `Carro` não pode ter a responsabilidade de iniciar a execução de um programa, pois ela é apenas um dos muitos componentes que podemos ter em um sistema completo.

Quando você estiver programando uma aplicação real, você terá centenas ou milhares de classes que especificam pequenas partes do sistema (componentes). Essas classes se comunicam entre si para fornecer as funcionalidades completas do sistema. Normalmente, você terá apenas uma classe com o método `main()` para iniciar o sistema. Esse método deve ser responsável por instanciar objetos de outras classes existentes e a partir daí esses objetos darão continuidade na execução do sistema.

Para desenvolver o método `main()`, é melhor criarmos uma classe que não tem nada haver com o domínio de nosso problema (e muito menos com orientação a objetos). Vamos dar o nome `Principal` para essa classe.

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

```
}
```

Ainda não colocamos código dentro do método `main()`. Precisamos nos perguntar: o que queremos fazer? Para começar, vamos apenas instanciar um objeto do tipo `Carro` e armazená-lo em uma variável.

```
class Principal {  
  
    public static void main(String[] args) {  
        Carro meuCarro; // declaração de variável  
        meuCarro = new Carro(); // instanciação de um carro  
    }  
  
}
```

O código do exemplo acima apenas declara uma variável chamada `meuCarro` do tipo `Carro` e depois instancia um objeto do tipo `Carro` e atribui à variável `meuCarro`. Usamos uma palavra `new` para indicar que desejamos instanciar um novo objeto do tipo `Carro`.

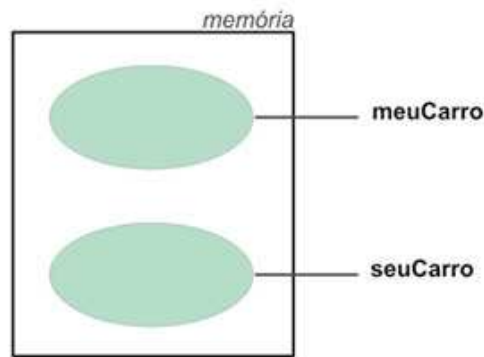
As variáveis não guardam os objetos, mas sim uma referência para a área de memória onde os objetos são alocados. Se criarmos duas instâncias da classe `Carro` e atribuirmos cada instância para variáveis diferentes, `meuCarro` e `seuCarro`, temos o código abaixo. Perceba que agora nós declaramos e instanciamos cada variável em apenas uma linha.

```
class Principal {  
  
    public static void main(String[] args) {  
        Carro meuCarro = new Carro();  
        Carro seuCarro = new Carro();  
    }  
  
}
```

Quando executamos o último código, parece que nada acontece. Na verdade, não conseguimos visualizar nada na tela, pois não incluímos nenhuma instrução para isso, mas dois objetos são instanciados e referenciados pelas variáveis de nomes `meuCarro` e `seuCarro`.

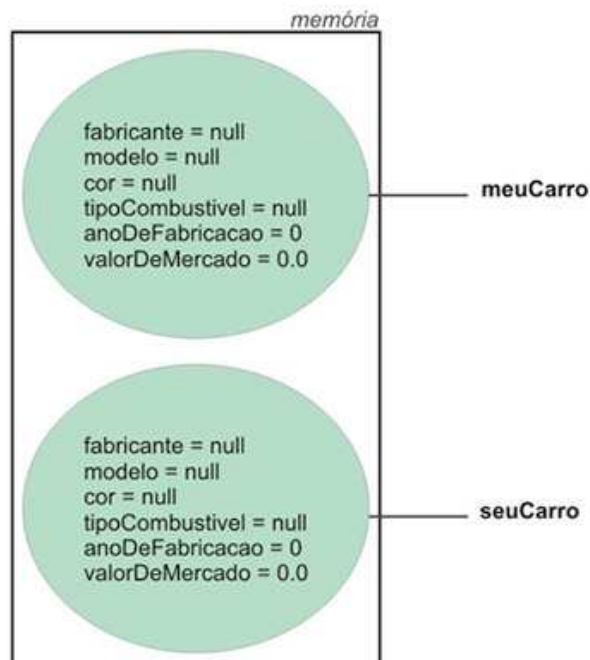
Os objetos instanciados são alocados na memória da máquina virtual Java, e os nomes de variáveis apenas apontam para esses objetos.





Veja que as variáveis referenciam objetos diferentes, apesar de serem do mesmo tipo.

Você pode estar se perguntando agora: Qual é a cor do carro? Qual é o preço de mercado? E o ano de fabricação? Como nós não atribuímos valores para essas variáveis, elas receberam valores padrão. Por exemplo, o ano de fabricação recebeu o valor 0, valor de mercado recebeu o valor 0.0 e todas as variáveis do tipo String receberam o valor null (que indica "nada", ou seja, sem referência).



Veremos no próximo tópico como acessar os atributos de objetos para atribuição e leitura.

## 8.5. Acessando atributos de objetos

Quando instanciamos um objeto, é comum precisarmos acessar os valores de seus atributos para realizar cálculos, tomar decisões ou simplesmente mostrá-los na tela. Para acessá-los, basta digitarmos o nome da variável que contém o objeto seguido por `.` (ponto) e o nome do atributo da instância. Veja um exemplo:

```
class Principal {  
  
    public static void main(String[] args) {  
        Carro meuCarro = new Carro();  
        System.out.println("Modelo: " + meuCarro.modelo);  
        System.out.println("Ano: " + meuCarro.anoDeFabricacao);  
    }  
}
```

A saída da execução do código acima é:

```
Modelo: null  
Ano: 0
```

Nós conseguimos acessar os atributos `modelo` e `anoDeFabricacao` de `meuCarro` e imprimir na tela, porém não existia uma descrição do modelo e nem um ano de fabricação para o carro instanciado, por isso tudo que vimos foram os valores padrão.

Se desejarmos atribuir (modificar) os valores das variáveis da instância `meuCarro`, basta usarmos o operador de atribuição `=` (igual), conforme o exemplo abaixo:

```
Carro meuCarro = new Carro();  
meuCarro.anoDeFabricacao = 2011;  
meuCarro.cor = "Prata";  
meuCarro.fabricante = "Fiat";  
meuCarro.modelo = "Palio";  
meuCarro.tipoCombustivel = "Bicombustivel";  
meuCarro.valorDeMercado = 30000;  
  
System.out.println("Modelo: " + meuCarro.modelo);  
System.out.println("Ano: " + meuCarro.anoDeFabricacao);
```

No exemplo acima, modificamos os valores de todas as variáveis de instância de `meuCarro` e depois acessamos `modelo` e `anoDeFabricacao` para exibir na tela. A execução do código tem a seguinte saída na tela:

```
Modelo: Palio  
Ano: 2011
```

Para que não restem dúvidas sobre o funcionamento das variáveis de instância, vamos fazer mais um exemplo, declarando e instanciando duas variáveis do tipo Carro.

```
Carro meuCarro = new Carro();
meuCarro.anoDeFabricacao = 2011;
meuCarro.cor = "Prata";
meuCarro.fabricante = "Fiat";
meuCarro.modelo = "Palio";
meuCarro.tipoCombustivel = "Bicombustível";
meuCarro.valorDeMercado = 30000;

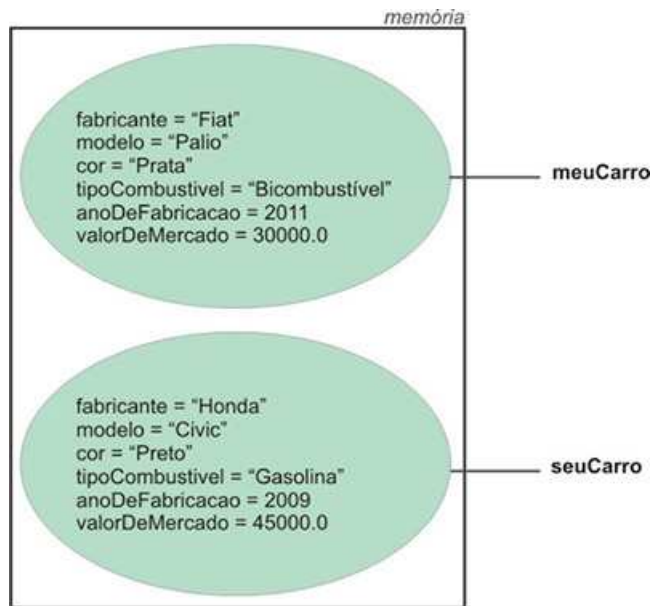
Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
seuCarro.valorDeMercado = 45000;

System.out.println("Meu carro");
System.out.println("-----");
System.out.println("Modelo: " + meuCarro.modelo);
System.out.println("Ano: " + meuCarro.anoDeFabricacao);

System.out.println();

System.out.println("Seu carro");
System.out.println("-----");
System.out.println("Modelo: " + seuCarro.modelo);
System.out.println("Ano: " + seuCarro.anoDeFabricacao);
```

O exemplo acima é bem simples, mas geralmente confunde algumas pessoas que estão iniciando os estudos em orientação a objetos. Nós instanciamos dois carros, meuCarro e seuCarro, e atribuímos valores diferentes às variáveis de instância de cada um. Veja abaixo uma representação das instâncias para entender melhor.



Veja que meuCarro e seuCarro têm os mesmos atributos (pois são do mesmo tipo), porém os valores de cada atributo podem ser diferentes. É como na vida real! Todos os carros possuem esses mesmos atributos, mas os valores desses atributos normalmente são diferentes entre o seu carro, o carro do seu vizinho, o carro do seu pai, etc.

Agora que você entendeu, deve ficar claro para você que a execução do último exemplo exibe na console:

Meu carro

-----

Modelo: Palio

Ano: 2011

Seu carro

-----

Modelo: Civic

Ano: 2009

## 8.6. Composição de objetos

Imagine que desejamos incluir atributos do proprietário do carro na classe Carro. O código ficaria assim:

```
class Carro {  
  
    String fabricante;
```

```

String modelo;
String cor;
String tipoCombustivel;
int anoDeFabricacao;
double valorDeMercado;

// atributos do proprietário
String nomeProprietario;
String cpfProprietario;
int idadeProprietario;
String logradouroProprietario;
String bairroProprietario;
String cidadeProprietario;
}

```

O exemplo acima até compila, mas não é bem visto por programadores de software mais experientes, pois a classe Carro passou a ter baixa coesão. Dizemos que uma classe não está coesa quando ela tem responsabilidades demais, como é o caso do último código. A classe Carro passou a guardar não só informações sobre o carro, mas também sobre o proprietário do carro e seu endereço.

No mundo real, um carro não tem nome, CPF, idade, logradouro, bairro ou cidade, mas sim o proprietário do carro. Por isso, vamos criar uma nova classe chamada Proprietario e incluir esses atributos.

```

class Proprietario {

    String nome;
    String cpf;
    int idade;
    String logradouro;
    String bairro;
    String cidade;

}

```

Agora alteramos a classe Carro para incluir um atributo que faz referência a Proprietario.

```

class Carro {

    String fabricante;
    String modelo;
    String cor;
    String tipoCombustivel;
    int anoDeFabricacao;
    double valorDeMercado;
    Proprietario dono;
}

```

```
}
```

A variável de instância dono recebeu esse nome apenas para não coincidir com o nome da classe, mas não teria nenhum problema se a variável tivesse o nome proprietario. Neste caso, achamos melhor os nomes não coincidirem para você entender que o nome da classe não tem nada haver com o nome do atributo.

O que acabamos de fazer foi uma composição de objetos. Composição é uma forma de combinar objetos simples em objetos mais complexos. Essa técnica traz grandes benefícios para a reutilização de objetos e legibilidade do código, além do código-fonte expressar melhor o que está acontecendo, de acordo com o mundo real.

Normalmente, podemos dizer que objetos compostos fazem parte de um relacionamento do tipo "tem um". Por exemplo, podemos dizer que o carro tem um proprietário.

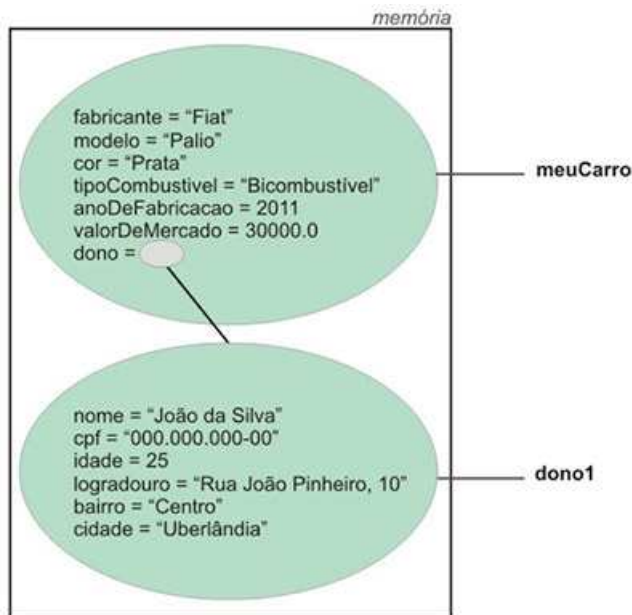
Olhando para a classe Proprietario, poderíamos ainda pensar em criar outra classe chamada Endereco para incluir o logradouro, bairro e cidade e a classe Proprietario incluiria um atributo do tipo Endereco. Não há nada de errado em fazer isso, mas vamos deixar como está, pois no momento não faz muito sentido para nossos exemplos.

Agora vamos instanciar um carro e um proprietário dentro do método main:

```
Proprietario dono1 = new Proprietario();
dono1.nome = "João da Silva";
dono1.cpf = "000.000.000-00";
dono1.idade = 25;
dono1.logradouro = "Rua João Pinheiro, 10";
dono1.bairro = "Centro";
dono1.cidade = "Uberlândia";

Carro meuCarro = new Carro();
meuCarro.anoDeFabricacao = 2011;
meuCarro.cor = "Prata";
meuCarro.fabricante = "Fiat";
meuCarro.modelo = "Palio";
meuCarro.tipoCombustivel = "Bicombustível";
meuCarro.valorDeMercado = 30000;
meuCarro.dono = dono1; // atribuímos o dono do carro
```

Veja no exemplo acima que instanciamos um Proprietario e depois um Carro isoladamente, e apenas na última linha que atribuímos dono1 à variável dono de meuCarro.

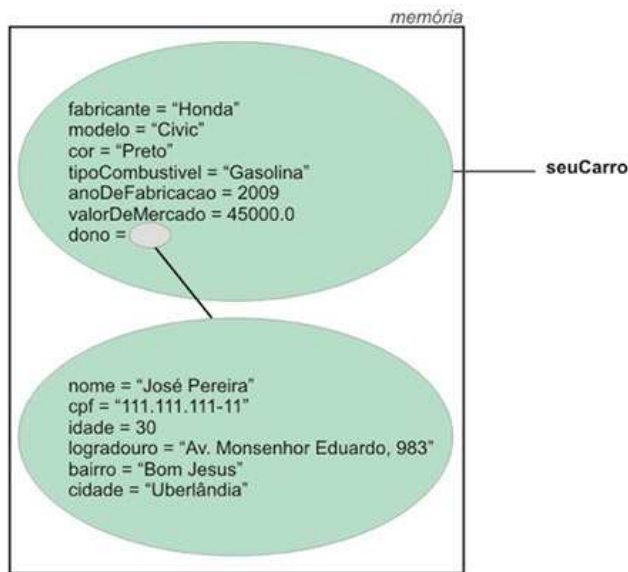


Outra forma comum de atribuir valores às variáveis de dono é como no exemplo abaixo:

```
Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
seuCarro.valorDeMercado = 45000;

// atribuindo valores às variáveis do dono do seuCarro
seuCarro.dono = new Proprietario();
seuCarro.dono.nome = "José Pereira";
seuCarro.dono.cpf = "111.111.111-11";
seuCarro.dono.idade = 30;
seuCarro.dono.logradouro = "Av. Monsenhor Eduardo, 983";
seuCarro.dono.bairro = "Bom Jesus";
seuCarro.dono.cidade = "Uberlândia";
```

Veja que no exemplo acima nós instanciamos Proprietario e atribuímos diretamente à variável dono de seuCarro. O resultado final é o mesmo, apenas não usamos outra variável intermediária para armazenar o proprietario instanciado.



## 8.7. Valores padrão

Você já estudou que, quando instanciamos um objeto usando o operador `new`, as variáveis de instância são inicializadas com valores padrão (*default*). Vamos fazer uma pequena revisão para lembrar os valores que cada tipo de variável recebe:

- Tipos numéricos primitivos recebem `0` ou `0.0`
- Tipo booleano recebe `false`
- Referências a objetos (String, Proprietario, Carro, etc) recebem `null`

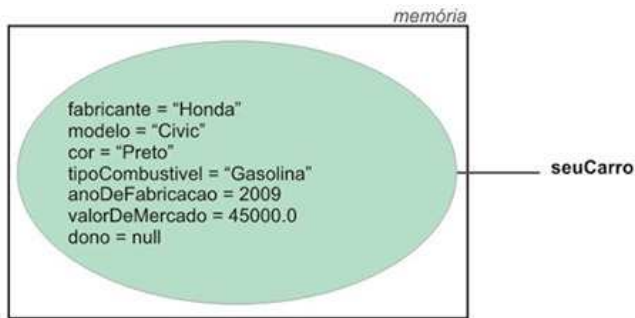
No exemplo abaixo, tentamos atribuir um valor à variável `nome` em `dono` de `seuCarro`.

```
Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
seuCarro.valorDeMercado = 45000;

// lança erro em tempo de execução
seuCarro.dono.nome = "José Pereira";
```

Veja na representação do objeto referenciado em `seuCarro` que não existe um proprietário (`dono`) associado:





A execução do código do exemplo acima gera um erro em tempo de execução chamado `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException  
    at Principal3.main(Principal3.java:11)
```

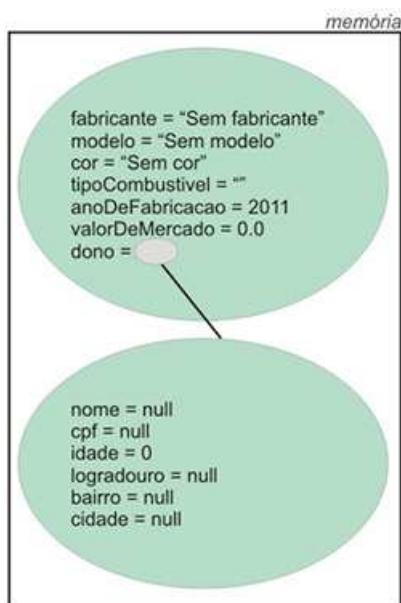
Nós estudaremos sobre os erros (exceções) mais adiante, porém o importante neste momento é você entender que não podemos atribuir um valor em uma referência nula (que não possui objeto associado).

Se quisermos evitar o `NullPointerException`, podemos deixar a variável `dono` com um valor padrão, instanciando um novo `Proprietario`. Vamos aproveitar e inicializar também as outras variáveis da classe `Carro`.

```
class Carro {  
  
    String fabricante = "Sem fabricante";  
    String modelo = "Sem modelo";  
    String cor = "Sem cor";  
    String tipoCombustivel = "";  
    int anoDeFabricacao = 2011;  
    double valorDeMercado = 0;  
    Proprietario dono = new Proprietario();  
  
}
```

Para inicializar as variáveis de instância, basta incluir o símbolo `=` (igual) seguido pelo valor ou pela instanciação do objeto logo após a declaração da variável. No caso da classe `Carro`, não faz muito sentido inicializar as variáveis com esses valores, mas mesmo assim fizemos a título de exemplo.

Agora, quando chamamos `new Carro()`, o novo objeto instanciado tem a seguinte representação (automaticamente):



Agora é seguro executar o código abaixo sem se preocupar com o erro `NullPointerException`.

```
Carro seuCarro = new Carro();
seuCarro.dono.nome = "José Pereira";
```

Apesar de parecer mais fácil sempre instanciar os objetos com valores padrão nas variáveis de instância de uma classe, conceitualmente pode estar incorreto, pois neste caso, seria o mesmo que dizer que todos os carros que acabaram de ser fabricados já possuem um dono automaticamente (o que pode não ser verdade).

## 8.8. Variáveis referenciam objetos

Quando instanciamos um objeto e atribuímos a uma variável, a variável não armazena o objeto, mas **faz referência** ao objeto. Esse conceito é muito importante para a programação orientada a objetos.

Veja os exemplos abaixo e as representações dos objetos na memória da máquina virtual. Não incluímos atribuições para algumas variáveis apenas para deixar o exemplo mais curto e fácil de entender.

Primeiramente, instanciamos um `Proprietario` e um `Carro` e atribuímos às variáveis `dono1` e `meuCarro`, respectivamente.

```
Proprietario dono1 = new Proprietario();
dono1.nome = "João da Silva";
```

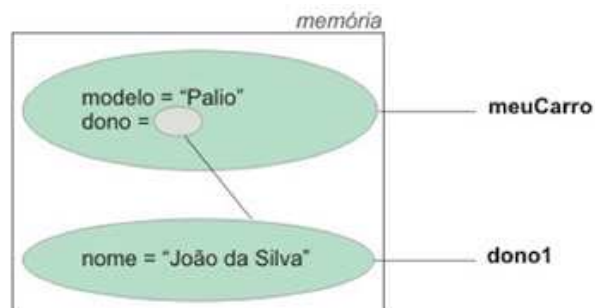
```
Carro meuCarro = new Carro();
meuCarro.modelo = "Palio";
```



Veja que os dois objetos não possuem nenhuma associação. O meuCarro não possui um dono (pois a variável dono está nula).

Quando atribuímos dono1 em meuCarro.dono, o proprietário "João da Silva" passa a ser referenciado também pela variável dono de meuCarro.

```
meuCarro.dono = dono1;
```



É importante entender que o objeto não é copiado (duplicado), mas apenas uma nova referência é feita até ele. Dessa forma, é correto dizer que:

```
// a linha abaixo
meuCarro.dono.nome = "Maria Joaquina";

// tem o mesmo efeito que
dono1.nome = "Maria Joaquina";
```

Quando dizemos que as duas instruções têm o mesmo efeito, quer dizer que o nome "João da Silva" será substituído por "Maria Joaquina" no mesmo objeto, pois só existe um objeto do tipo Proprietario, que é o que representa o "João da Silva".

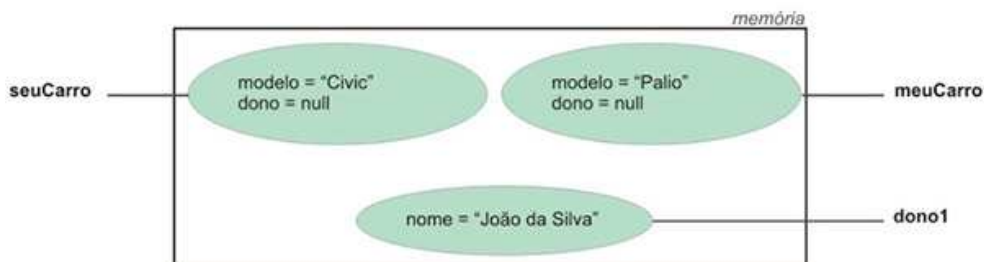
Para ficar ainda mais claro, vamos a mais um exemplo.

```
Proprietario dono1 = new Proprietario();  
dono1.nome = "João da Silva";
```

```
Carro meuCarro = new Carro();  
meuCarro.modelo = "Palio";
```

```
Carro seuCarro = new Carro();  
seuCarro.modelo = "Civic";
```

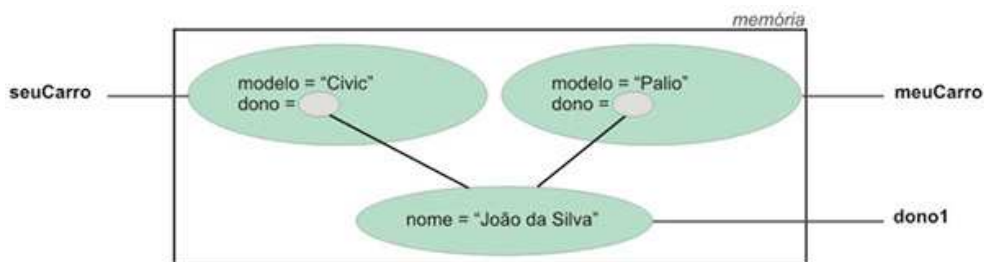
Instanciamos um Proprietario e dois objetos do tipo Carro, porém ainda não associamos nenhum deles.



Agora associamos a variável dono1 à variável dono de meuCarro e seuCarro.

```
meuCarro.dono = dono1;  
seuCarro.dono = dono1;
```

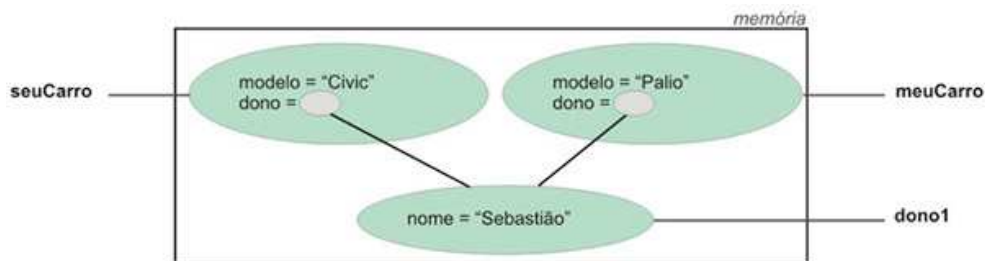
Quando fazemos isso, estamos dizendo que meuCarro e seuCarro possuem o mesmo dono (objeto do tipo Proprietario), que é o "João da Silva".



Chegamos a uma parte muito importante. Vamos mudar o nome do dono do meuCarro para "Sebastião".

```
meuCarro.dono.nome = "Sebastião";
```

Veja agora a representação dos objetos.



Quando mudamos o nome do dono do meuCarro para "Sebastião", na verdade estávamos mudando também o nome do dono do seuCarro, pois é o mesmo objeto, ou seja, é o mesmo dono, exatamente como acontece no mundo real.

Imagine se no mundo real o João tem dois carros, um do modelo Palio e outro do modelo Civic. Se por algum motivo estranho, o João conseguir mudar seu nome para Sebastião, ele ainda é o dono dos dois carros, porém agora ele tem um novo nome. Se o João ficou doente, é o dono do Civic ou o dono do Palio que está doente? Ahãaa... agora você entendeu!

## 8.9. Criando e chamando métodos

Os comportamentos das classes são definidos através de métodos. Métodos são trechos de código que são referenciados por um nome e podem ser chamados por alguma outra parte do software através de seu nome.

Os métodos podem receber argumentos (valores e objetos de entrada) e retornar um valor ou objeto para quem o chamou. A sintaxe dos métodos é semelhante às das funções de um programa procedural, mas em orientação a objetos, não é comum chamá-los de funções.

Nós definimos anteriormente que os comportamentos do carro devem ser:

- Ligar
- Desligar
- Mudar marcha
- Acelerar

- Frear

Cada comportamento dessa lista deve se tornar um método na classe Carro. Primeiramente, vamos implementar o método ligar:

```
void ligar() {  
    System.out.println("Ligando carro " + modelo);  
}
```

No método ligar, usamos a palavra-reservada void para dizer que o método não tem retorno, ou seja, nenhuma informação será retornada para quem chamar (invocar) o método. Estudaremos outros tipos de retorno adiante. Após o nome do método, abrimos e fechamos os parênteses para dizer que esse método não recebe argumentos, ou seja, quem invocar este método não terá opção de enviar informações (valores ou objetos) a ele.

O código da programação do método deve estar dentro de um bloco, delimitado pela abertura e fechamento de chaves, mesmo que seja apenas uma linha, como é o caso do exemplo acima.

Para não complicar neste momento, programamos o método ligar para apenas exibir uma mensagem dizendo que o carro de um determinado modelo está sendo ligado. Preste atenção que modelo é uma variável de instância da classe Carro.

Para chamar o método ligar, precisamos de uma instância de carro. Vamos programar a chamada no método main da classe Principal.

```
public static void main(String[] args) {  
    Carro seuCarro = new Carro();  
    seuCarro.modelo = "Civic";  
    // atribuição de outras variáveis de instância aqui...  
  
    // chamando o método ligar  
    seuCarro.ligar();  
}
```

Veja que para chamar um método, precisamos indicar o nome da variável que aponta para o objeto (neste caso, seuCarro), seguido por ponto e o nome do método. A abertura e fechamento de parênteses indicam que não será passado nenhum argumento ao método.

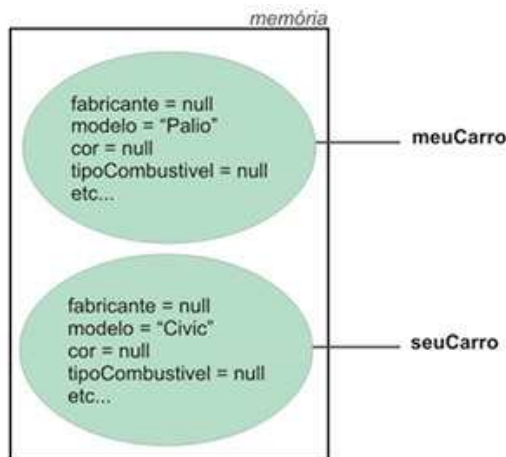
Ao executar a classe Principal, a saída na console é:

Ligando carro Civic

Agora vamos alterar a classe Principal para instanciar dois objetos do tipo Carro e chamar o método ligar em cada um deles.

```
public static void main(String[] args) {  
    Carro seuCarro = new Carro();  
    seuCarro.modelo = "Civic";  
  
    Carro meuCarro = new Carro();  
    meuCarro.modelo = "Palio";  
  
    seuCarro.ligar();  
    meuCarro.ligar();  
}
```

Veja a imagem abaixo que representa os objetos que acabamos de instanciar no código-fonte acima. Perceba que meuCarro é um Palio e seuCarro é um Civic.



Milagrosamente, ao executarmos a classe Principal, temos seguinte saída:

```
Ligando carro Civic  
Ligando carro Palio
```

Muitas pessoas iniciantes em programação orientada a objetos "travam" aqui. É comum elas falarem "Mas como pode? Palio não substituiu Civic quando atribuí à variável?".

Lembre-se que estamos programando orientado a objetos. Pense no mundo real! Se o meu carro é um Palio e o seu é um Civic (ambos são do tipo Carro, não é mesmo?), ao sair de casa para o trabalho com meu carro, qual deles eu vou ligar? Claro que é o Palio, que é meu. O mesmo acontece com você... ao ligar o seu carro, é o Civic que levará você onde quiser.

## 8.10. Nomeando métodos

Os métodos em Java podem conter letras, dígitos, \_ (underscore) e \$ (dólar), porém eles não podem ser iniciados por um dígito e não podem ser palavras reservadas.

Veja alguns nomes de métodos válidos para o compilador Java:

```
void conferirEstoque() // pode ser todo em letras minúsculas
void despacha_produtos() // pode ter underscore
void DESLIGAR() // pode ser todo em letras maiúsculas
void ExibirSaldo() // pode ter letras maiúsculas e minúsculas
void $salvados() // pode iniciar com dólar
void _exibirFoto() // pode iniciar com underscore
void listar_alunos_nota_10() // pode ter dígitos
```

Agora alguns nomes de métodos inválidos (que nem compila):

```
void 2calcular() // não pode iniciar com dígitos
void ver extrato() // não pode ter espaços
void new() // new é uma palavra reservada do Java
```

Apesar da linguagem suportar letras maiúsculas e minúsculas, underscore, dólar e dígitos nos nomes dos métodos, a convenção de código Java diz que eles devem ser nomeados com a inicial em letra minúscula e as demais iniciais das outras palavras em letras maiúsculas. Além disso, os métodos devem usar verbos (se possível no infinitivo), pois assim fica fácil de ler e entender a ação que ele se propõe a fazer. Veja alguns exemplos:

```
void verExtrato()
void listarAlunosNota10()
void despacharPedido()
void exibirFotoEmMiniatura()
```

É uma boa prática escrever as frases e palavras completas quando vamos declarar métodos em Java. Abreviações devem ser usadas somente se forem muito conhecidas no domínio do negócio. Por exemplo, você evitar:

```
void verificar() // verifica o que? Que tal verificarEstoque?
void lstAlu() // listar alunos? Até que seu cérebro interprete isso...
void sCliente() // que isso? O que se pretende fazer com o cliente?
void calcQtdCliInat() // não é natural ler isso
```

Ao programar, lembre-se dessas convenções. Os próximos programadores que forem pegar o seu código para dar manutenção agradecem. :)



## 8.11. Métodos com retorno

A execução de um método retorna para o código que o chamou, quando acontece qualquer uma das seguintes situações:

- O bloco de código do método chega ao fim
- Uma instrução `return` é encontrada
- Uma exceção é lançada (ensinaremos sobre isso mais adiante)

Para um método poder retornar um valor, precisamos informar o tipo de retorno na declaração dele. A classe `Paciente`, definida no exemplo abaixo, possui um método `calcularIndiceDeMassaCorporal` que retorna um valor do tipo `double`, indicando o IMC do objeto (paciente).

```
class Paciente {  
  
    double peso;  
    double altura;  
  
    double calcularIndiceDeMassaCorporal() {  
        double imc = peso / (altura * altura);  
  
        // retornamos o valor calculado do tipo double  
        return imc;  
    }  
}
```

A instrução `return` no exemplo acima é obrigatória. Quando um método define um tipo de retorno, o método **deve** retornar sempre algum valor.

O tipo do valor retornado deve sempre coincidir com o tipo do retorno declarado no método, ou pelo menos ser possível fazer uma conversão implícita. Por exemplo, um método declarado para retornar um tipo `int` não pode devolver um valor do tipo `long`, `double` ou `float`, mas pode retornar um valor do tipo `int`, `short`, `char` ou `byte`.

O método de cálculo do IMC de um paciente pode ser chamado através de outra classe, que pode obter o retorno e usá-lo como desejar. Veja o exemplo abaixo:

```
public static void main(String[] args) {  
    Paciente p = new Paciente();  
    p.peso = 70.5;  
    p.altura = 1.75;  
  
    // invocamos o método e guardamos o retorno em uma variável  
    double imcCalculado = p.calcularIndiceDeMassaCorporal();  
}
```

```

        System.out.println("IMC do paciente: " + imcCalculado);
    }

```

O exemplo do método de cálculo do IMC retorna um tipo primitivo `double`, mas podemos também retornar objetos. Por exemplo, imagine que ao invés do método de cálculo do IMC retornar apenas o índice (um número), queremos retornar várias informações que obtemos durante a análise do IMC. Para isso, criamos uma classe chamada `IMC` com alguns atributos de instância. Veja:

```

class IMC {

    double indice;
    boolean abaixoDoPesoIdeal;
    boolean pesoIdeal;
    boolean obeso;
    String grauObesidade;

}

```

Agora o método de cálculo foi alterado para retornar um objeto do tipo `IMC` contendo dados já analisados, baseado no índice obtido.

```

IMC calcularIndiceDeMassaCorporal() {
    // instanciamos um objeto do tipo IMC
    IMC imc = new IMC();

    // calculamos o índice de massa corporal
    double indice = peso / (altura * altura);

    // analisamos o índice do paciente
    if (indice < 18.5) {
        imc.abaixoDoPesoIdeal = true;
    } else if (indice < 25) {
        imc.pesoIdeal = true;
    } else {
        imc.obeso = true;

        if (indice < 30) {
            imc.grauObesidade = "Acima do peso";
        } else if (indice < 35) {
            imc.grauObesidade = "I";
        } else if (indice < 40) {
            imc.grauObesidade = "II";
        } else {
            imc.grauObesidade = "III";
        }
    }

    // retornamos o objeto do tipo IMC com a análise feita
    return imc;
}

```

Ao chamar o método `calcularIndiceDeMassaCorporal`, recebemos não só o índice, mas também dados analisados indicando se o índice representa obesidade, peso abaixo do ideal ou peso ideal.

Essa é uma boa prática de programação, pois reduz a possibilidade de ter essa mesma lógica para analisar o IMC em diversos pontos do seu código-fonte.

Agora é só chamar o método e verificar os atributos do objeto.

```
public static void main(String[] args) {
    Paciente p = new Paciente();
    p.peso = 100;
    p.altura = 1.65;

    IMC imc = p.calcularIndiceDeMassaCorporal();

    System.out.println("Abaixo do peso ideal: "
        + imc.abaixoDoPesoIdeal);
    System.out.println("Peso ideal: " + imc.pesoIdeal);
    System.out.println("Obeso: " + imc.obeso);
    System.out.println("Grau de obesidade: " + imc.grauObesidade);
}
```

O resultado da execução do código acima é:

```
Abaixo do peso ideal: false
Peso ideal: false
Obeso: true
Grau de obesidade: II
```

Métodos que declaram `void` como seu tipo de retorno não podem retornar nenhum valor. Nesse caso, nós não precisamos incluir uma instrução `return`. Mesmo assim, se em algum momento desejarmos sair da execução do método atual e voltar para quem chamou, podemos simplesmente usar a instrução `return` sem nenhum valor. Veja um exemplo:

```
void ligar() {
    if (modelo == null) {
        return;
    }

    System.out.println("Ligando carro " + modelo);
}
```

O método acima pára de ser executado se a variável de instância `modelo` for nula, pois não faz sentido ligar o carro se ele não tiver um modelo.

Apesar de parecer interessante, na maioria das vezes não é nada elegante usar o `return` para sair no meio de um método. As boas práticas dizem que um método deve ter um único ponto de saída, ou seja, apenas uma instrução `return`.

Podemos fazer um refactoring no exemplo acima para deixar o código mais legível e dentro das boas práticas de programação. Veja abaixo:

```
void ligar() {
    if (modelo != null) {
        System.out.println("Ligando carro " + modelo);
    }
}
```

## 8.12. Passando argumentos para métodos

Ao declarar métodos em Java, podemos definir alguns argumentos que devem ser passados para executá-los. No exemplo abaixo, criamos um método que calcula o valor do salário de um funcionário baseado no número de horas normais e horas extras trabalhadas e nos valores em reais combinados por hora.

```
class FolhaPagamento {

    double calcularSalario(int horasNormais, int horasExtras,
        double valorHoraNormal, double valorHoraExtra) {

        // cálculo do salário
        double valorHorasNormais = horasNormais * valorHoraNormal;
        double valorHorasExtras = horasExtras * valorHoraExtra;

        return valorHorasNormais + valorHorasExtras;
    }

}
```

Um termo que falamos bastante quando queremos especificar um determinado método, juntamente com o seu tipo de retorno e seus parâmetros é "assinatura do método". A assinatura do método do último exemplo é a seguinte:

```
double calcularSalario(int, int, double, double)
```

O método `calcularSalario` recebe 4 argumentos, sendo os dois primeiros do tipo `int` e os dois últimos do tipo `double`, e retorna um valor do tipo `double`.

Os parâmetros de um método devem ter seus nomes logo após a definição de seus tipos. Os nomes não podem repetir e também não podem coincidir com nomes de variáveis locais do método.

Para chamar um método que possui parâmetros, basta informar o nome dele seguido pelos argumentos, que devem ficar entre parênteses e serem separados por vírgulas. Veja:

```
public static void main(String[] args) {  
    FolhaPagamento folha = new FolhaPagamento();  
    double salario = folha.calcularSalario(160, 12, 32.5, 40.2);  
  
    System.out.println("Salário calculado: " + salario);  
}
```

A saída da execução do código acima é:

Salário calculado: 5682.4

Você pode usar qualquer tipo para um parâmetro de um método, incluindo tipos primitivos ou tipos de classes (que estudaremos logo em seguida).

Apesar de, tecnicamente, não existir um número máximo de parâmetros que possa ser usado em métodos, não é recomendado que você defina mais que 2 ou 3 parâmetros, pois ao passar muitos argumentos, a legibilidade do código é afetada e normalmente indica um mal cheiro (*Code smell*).

## 8.13. Argumentos por valor ou por referência

Quando passamos argumentos de tipos primitivos para um método, eles são passados por valor. Isso quer dizer que qualquer alteração nos valores dos argumentos recebidos tem efeito apenas no escopo do método, mas não fora dele. Veja um exemplo:

```
class Produto {  
  
    void definirPreco(double precoCusto) {  
        // adiciona 20% de impostos  
        precoCusto = precoCusto * 1.20;  
  
        // faz várias outras coisas...  
    }  
}
```

O método `definirPreco` do código-fonte acima altera o valor da variável `precoCusto`, que foi recebida como parâmetro do método. O método poderia fazer várias outras coisas para ter algum sentido, mas queremos apenas testar a passagem de parâmetro, por isso ignoramos o restante do código.

O código abaixo instancia um `Produto` e invoca o método `definirPreco`, passando o valor da variável `preco` como argumento.

```
public static void main(String[] args) {  
    double preco = 140;  
  
    Produto p = new Produto();  
    p.definirPreco(preco);  
  
    System.out.println("Preço: " + preco);  
}
```

Ao executar esse exemplo, a saída é:

Preço: 140.0

Veja que o método `definirPreco` alterou o valor do argumento, que teve efeito apenas dentro do método, pois foi passado por valor. Não existem outras possibilidades quando estamos trabalhando com tipos primitivos.

Quando passamos um objeto como argumento para um método, uma referência do objeto é passada. Isso quer dizer que o objeto passado por quem chama o método é o mesmo objeto que é acessado dentro do corpo do método. Sendo assim, qualquer alteração nos valores dos atributos do objeto dentro da execução do método reflete também do lado de fora (de quem chamou), pois na realidade o objeto é o mesmo.

Para exemplificar, criamos uma classe `Preco` que possui vários atributos que fazem a composição de um preço de venda.

```
class Preco {  
  
    double valorCustos;  
    double valorImpostos;  
    double valorLucro;  
    double precoVenda;  
  
}
```

O método `definirPreco` da classe `Produto` receberá um objeto do tipo `Preco` como parâmetro, além do percentual de impostos e margem de lucro a serem usados para calcular o preço de venda.

Veja no código-fonte abaixo que as variáveis de instância de `preco` estão sendo alteradas a partir dos cálculos feitos.

```
class Produto {  
  
    void definirPreco(Preco preco, double percentualImpostos,  
        double margemLucro) {  
        preco.valorImpostos = preco.valorCustos  
            * (percentualImpostos / 100);  
        preco.valorLucro = preco.valorCustos  
            * (margemLucro / 100);  
        preco.precoVenda = preco.valorCustos + preco.valorImpostos  
            + preco.valorLucro;  
  
        // faz várias outras coisas...  
    }  
  
}
```

A classe principal, que possui o método `main`, instancia um objeto do tipo `Preco` e atribui um valor para a variável de instância `valorCustos`. Depois disso, chama o método `definirPreco` da classe `Produto`. O objeto `preco` é passado como parâmetro, e ao executar o método, as variáveis desse objeto são alteradas, refletindo no resultado visualizado pelo usuário.

```
public static void main(String[] args) {  
    Preco preco = new Preco();  
    preco.valorCustos = 140;  
  
    Produto produto = new Produto();  
    produto.definirPreco(preco, 20, 15);  
  
    System.out.println("Valor custos: " + preco.valorCustos);  
    System.out.println("Valor impostos: " + preco.valorImpostos);  
    System.out.println("Valor lucro: " + preco.valorLucro);  
    System.out.println("Preço venda: " + preco.precoVenda);  
}
```

A execução do último exemplo exibe na saída:

```
Valor custos: 140.0  
Valor impostos: 28.0  
Valor lucro: 21.0  
Preço venda: 189.0
```

O método `main` passou uma referência de um objeto do tipo `Preco` para o método `definirPreco`. Esse objeto foi alterado dentro do método, e quando a execução voltou para o `main`, o objeto encontrava-se alterado.

Vamos fazer uma analogia com o mundo real. Imagine que você tenha um carro de cor vermelha, e você o emprestou para um amigo. Se o seu amigo mandar pintar seu carro de preto, mesmo que ele ainda não tenha devolvido para você, na prática, qual é a cor do seu carro? Quando você pegar o carro de volta, qual será a cor dele? Preto, claro! Isso quer dizer que você passou seu carro com todos os atributos para seu amigo, e ele pode fazer o que quiser com ele (quase tudo), e isso refletirá no seu carro quando você o ver novamente.

## 8.14. Métodos que alteram variáveis de instância

Os métodos declarados em uma classe, podem, além de alterar variáveis locais e as recebidas como parâmetro (que também possuem o escopo local), alterar as variáveis de sua própria instância.

Veja a classe `Aeronave`, declarada abaixo:

```
class Aeronave {  
  
    int totalAssentos;  
    int assentosReservados;  
  
    void reservarAssentos(int numeroAssentos) {  
        assentosReservados += numeroAssentos;  
    }  
  
    int calcularAssentosDisponiveis() {  
        return totalAssentos - assentosReservados;  
    }  
}
```

Esta classe representa uma aeronave, e para simplificar, criamos apenas dois atributos, `totalAssentos` e `assentosReservados`. O método `reservarAssentos` recebe como parâmetro o número de assentos que devem ser reservados e altera a variável de instância `assentosReservados`, incrementando o valor dessa variável.

Criamos também um método `calcularAssentosDisponiveis`, que calcula o número de assentos livres na aeronave.

Agora vamos para a parte mais legal: programar um código que instancia aeronaves e usa os métodos que criamos.

```
public static void main(String[] args) {  
    Aeronave aviaoGol = new Aeronave();  
}
```



```

        aviaoGol.totalAssentos = 100;

        Aeronave aviaoTam = new Aeronave();
        aviaoTam.totalAssentos = 130;

        aviaoGol.reservarAssentos(10);
        aviaoTam.reservarAssentos(5);

        int assentosGol = aviaoGol.calcularAssentosDisponiveis();
        int assentosTam = aviaoTam.calcularAssentosDisponiveis();

        System.out.println("Assentos disponíveis - Gol: " + assentosGol);
        System.out.println("Assentos disponíveis - TAM: " + assentosTam);
    }

```

No código acima, instanciamos duas aeronaves, sendo uma da Gol e outra da TAM. A primeira, tem capacidade de 100 passageiros, enquanto a segunda tem capacidade de 130 passageiros. Reservamos 10 assentos na aeronave da Gol e 5 assentos na aeronave da TAM, por isso, ao executarmos o código, a saída na console deve ser a seguinte:

```

Assentos disponíveis - Gol: 90
Assentos disponíveis - TAM: 125

```

Dá pra perceber que, quando invocamos um método que altera as variáveis de sua instância, isso é refletido exatamente no objeto em que chamamos o método, certo? Por isso que métodos, como os que declaramos, são chamados de métodos de instância.

## 8.15. O objeto this

A palavra reservada `this` faz referência ao próprio objeto, quando usado dentro de um método, por exemplo.

No código-fonte da classe `Aeronave`, vamos criar um método chamado `alterarTotalAssentos`, que receberá um argumento com o novo número de passageiros a ser atribuído à variável de instância `totalAssentos`.

```

class Aeronave {

    int totalAssentos;
    int assentosReservados;

    void reservarAssentos(int assentos) {
        assentosReservados += assentos;
    }

    int calcularAssentosDisponiveis() {

```

```

        return totalAssentos - assentosReservados;
    }

    void alterarTotalAssentos(int totalAssentos) {
        totalAssentos = totalAssentos;
    }
}

```

No código acima, existe um erro que pode passar despercebido. O método `alterarTotalAssentos` não tem efeito algum sobre a variável `totalAssentos`. Veja que o nome do argumento é o mesmo da variável de instância. Quando fazemos a atribuição, estamos dizendo que o valor da variável local (recebida como parâmetro) deve ser alterado para ele mesmo!

Claro que, uma forma bem fácil de resolver isso, seria alterar o nome da variável do parâmetro e evitar essa confusão, mas, muitas vezes, usar um nome diferente dificultaria a legibilidade do código.

Vamos resolver esse problema usando a palavra-chave `this`.

```

void alterarTotalAssentos(int totalAssentos) {
    this.totalAssentos = totalAssentos;
}

```

Agora sim! Estamos dizendo que queremos atribuir o valor da variável local à variável de instância. Quando usamos `this`, estamos deixando essa informação explícita.

## 8.16. Sobrecarga de métodos

Sobrecarga de métodos é um conceito simples da orientação a objetos, que permite a criação de vários métodos com o mesmo nome, mas com parâmetros diferentes.

Para exemplificar, vamos alterar a classe `Aeronave` para incluir a possibilidade de reserva de assentos normais e especiais. Para isso, criamos novas variáveis de instância.

```

class Aeronave {

    int totalAssentosNormais;
    int totalAssentosEspeciais;
    int assentosNormaisReservados;
    int assentosEspeciaisReservados;

    void reservarAssentos(int assentos) {
        this.assentosNormaisReservados += assentos;
    }
}

```

```

    }

    int calcularAssentosDisponiveis() {
        return totalAssentosNormais - assentosNormaisReservados
            + totalAssentosEspeciais - assentosEspeciaisReservados;
    }
}

```

Veja que refatoramos os métodos `reservarAssentos` e `calcularAssentosDisponiveis`, para usar as novas variáveis que criamos.

Já temos um método para fazer reserva de assentos, que reserva assentos normais na aeronave, mas, e se quisermos reservar assentos especiais?

Poderíamos criar um método chamado `reservarAssentosEspeciais`, mas não vamos fazer isso. Vamos sobrecarregar o método `reservarAssentos`, incluindo uma nova versão com parâmetros adicionais. Veja:

```

void reservarAssentos(int assentosNormais, int assentosEspeciais) {
    this.assentosNormaisReservados += assentosNormais;
    this.assentosEspeciaisReservados += assentosEspeciais;
}

```

O método acima recebe, além do número de assentos normais, o total de assentos especiais a serem reservados. Isso é sobrecarga de métodos! Temos duas versões de métodos com o nome `reservarAssentos`. Vamos ver as assinaturas desses métodos?

```

void reservarAssentos(int)
void reservarAssentos(int, int)

```

Sobrecarga de métodos é algo simples de ser feito, mas tem uma restrição que a própria linguagem impõe. Não é possível ter duas versões de métodos com a mesma assinatura. Por exemplo, seria impossível ter o método a seguir na classe `Aeronave`:

```

void reservarAssentos(int assentosEspeciais) {
    this.assentosEspeciaisReservados += assentosEspeciais;
}

```

O código acima seria uma tentativa de criar uma versão do método `reservarAssentos`, para reservar assentos especiais, mas não funcionaria (nem compilaria), porque a classe `Aeronave` já possui um método `reservarAssentos` que recebe um `int`.

## 8.17. Crie bons métodos

Existem algumas práticas para você criar bons métodos, e não servem apenas para Java. Vejamos algumas:

- Métodos devem ser pequenos. Não existe uma regra, mas quanto menor, melhor. Métodos de 1 ou 2 linhas fazem todo sentido!
- Métodos devem fazer apenas uma coisa, não mais que isso.
- O nível de indentação não deve ser maior que 2, ou seja, evite os famosos e horríveis *ifs* dentro de *ifs*, blocos dentro de blocos.
- Métodos que não recebem parâmetros são os melhores, mas se precisar de parâmetros, inclua 2, ou no máximo 3. Mais que isso, seu código ficará complicado de ser entendido.
- Não repita você mesmo, ou, como o termo é mais conhecido, "Don't Repeat Yourself" (DRY). Isso quer dizer que, se você precisa de um trecho de código de outro método, você não deve usar *Ctrl+C*. Pense em quebrar seu método em pequenos métodos para reaproveitar o que precisa.
- Coloque nomes em seus métodos e parâmetros que sejam fáceis de entender e que sejam coerentes com o domínio do negócio. Evite abreviações, a não ser que, no negócio, elas sejam muito usadas e entendidas por todos.

## 8.18. Coletor de lixo

O Coletor de lixo, também conhecido como *Garbage collector*, é um mecanismo da JVM para retirar objetos na memória que não estão sendo mais usados. Com esse recurso da máquina virtual, o programador é desobrigado a gerenciar a memória do programa.

```
Aeronave aviao = new Aeronave();  
  
// executa outras coisas demoradas aqui
```

Quando instanciamos um objeto *Aeronave*, ele vai parar na memória da JVM. De tempos em tempos, o Coletor de lixo irá verificar objetos não usados para fazer a limpeza, mas neste caso, o objeto referenciado pela variável *aviao* não será removido da memória.

```
Aeronave aviao = new Aeronave();  
aviao = null;  
  
// executa outras coisas demoradas aqui
```

Agora nós atribuímos `null` à variável `aviao`. Aquele objeto do tipo `Aeronave` continua na memória, porém sem referência alguma apontando para ele. Quando o Coletor de lixo encontrar o objeto, ele será removido da memória, liberando mais espaço.

Não conseguimos forçar a execução do *Garbage collector*, mas podemos recomendar a JVM que rode a coleta de lixo o mais rápido possível pelo comando `System.gc()`. Embora exista esse recurso, não podemos confiar nele. Seu programa nunca deve depender da execução do Coletor de lixo pelo `System.gc()`.

# Wrappers e boxing

## 9.1. O que são classes wrapper?

Em Java, as únicas coisas que não são objetos, são os tipos primitivos. Os tipos `int`, `long`, `char`, `boolean` e etc, não pertencem a nenhuma classe, eles são definidos na própria linguagem. Muitas vezes, precisamos enxergar números, caracteres ou valores booleanos como objetos também, e por isso, precisamos converter esses valores em objetos. Para fazer essa conversão, precisamos usar as classes wrapper.

As classes wrapper "embrulham" tipos primitivos para que eles possam ser tratados como objetos, e fornecem alguns métodos utilitários, normalmente para trabalhar com conversões.

Cada tipo primitivo existente na linguagem Java tem sua classe wrapper correspondente. Os nomes das classes wrappers coincidem com o nome do tipo primitivo, começando com letra maiúscula, exceto para o tipo `char` e `int`, que recebem o nome completo (e não abreviado) para os seus tipos wrapper. Vejamos as classes wrappers existentes e suas correspondências para os tipos primitivos:

- `Boolean` → `boolean`
- `Character` → `char`
- `Byte` → `byte`
- `Short` → `short`
- `Integer` → `int`
- `Long` → `long`
- `Float` → `float`

- Double → double

É interessante usar classes wrappers quando queremos enxergar números, booleanos ou caracteres como objetos. Os tipos primitivos não aceitam valores nulos e também não são suportados por algumas APIs, como a de Collections, que estudaremos mais a frente. Quando esse é o caso, usamos as classes wrapper.

## 9.2. Usando classes wrappers

Todas as classes wrapper, com exceção de Character, fornecem dois construtores: um que recebe o tipo primitivo como argumento e outro uma String. Veja exemplos de uso:

```
// instanciando um wrapper que representa
// um número primitivo do tipo long
Long idadeEmMilisegundos = new Long(933120000000L);

// instanciando um wrapper que transforma
// a String "15" em tipo primitivo
// e embrulha em um objeto wrapper
Integer diasParaPagamento = new Integer("15");

// wrapper que representa um double
Double precoPassagem = new Double(200.10);

// wrapper que representa um float
Float distanciaPercorrida = new Float("100.6");

// representa o valor true
Boolean temPendencias = new Boolean(true);

// representa o valor false
// (qualquer texto diferente de true representa false)
Boolean atrasado = new Boolean("Yes");
```

Os construtores de classes wrappers que representam tipos numéricos e que recebem uma String no construtor como parâmetro, lançam exceções (erros) se a String fornecida não puder ser transformada em um tipo primitivo apropriado. Estudaremos sobre tratamento de exceções em outro capítulo, mas vamos ver alguns exemplos inválidos.

```
// não faz transformação de um número
// por extenso
Integer idade = new Integer("trinta e um");

// os decimais devem ser separados por
```

```
// ponto, e não vírgula
Double precoTotal = new Double("140,30");
```

Quando não for possível efetuar uma conversão de uma String para um tipo específico (Integer, por exemplo), você receberá uma exceção, como pode ver abaixo:

```
Exception in thread "main" java.lang.NumberFormatException:
  For input string: "trinta e um"
  at java.lang.NumberFormatException.forInputString(
    NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:449)
  at java.lang.Integer.<init>(Integer.java:660)
```

## 9.3. Métodos de conversão

Uma das grandes funções de um wrapper é converter coisas. Quando você precisar converter o valor de um objeto wrapper numérico em um tipo primitivo, use um dos muitos métodos xxxValue(). Todos os métodos dessa família não recebem argumentos. Cada classe wrapper numérica possui vários métodos xxxValue(), de forma que todos os objetos wrapper numéricos podem ser convertidos para qualquer tipo primitivo numérico. Por exemplo:

```
Integer idade = new Integer(31);

byte idadeByte = idade.byteValue();
short idadeShort = idade.shortValue();
double idadeDouble = idade.doubleValue();

Float precoProduto = new Float(45.25f);

// converte para short e trunca o valor
short precoShort = precoProduto.shortValue();
```

Para converter strings em tipos primitivos, use um dos seis métodos parseXxx(), dentro de cada tipo wrapper correspondente. Esses métodos lançam uma exceção, caso a string informada não esteja em um formato possível de se transformar em tipo primitivo. Veja alguns exemplos:

```
String numero = "40";
String valor = "12.34";

double precoProduto = Double.parseDouble(valor);
int idade = Integer.parseInt(numero);
```

Agora você já sabe! Sempre que tiver uma string e quiser converter para um tipo primitivo, basta usar um dos métodos parseXxx() correspondentes.



## 9.4. Autoboxing do Java 5

A partir do Java 5, você pode usar os recursos chamados de *autoboxing* e *autounboxing*, que permitem que tipos primitivos sejam usados como objetos, e vice-versa. Esse recurso faz o embrulho de tipos primitivos para objetos e o desembrulho de objetos para tipos primitivos, automaticamente, sempre que for necessário.

Até o Java 1.4, o código abaixo era inválido (não compilava):

```
Double preco = 25.78;
int idade = new Integer(20);
```

Esse código é inválido se for compilado com Java 1.4, pois um valor em tipo primitivo não poderia ser atribuído a uma variável de um tipo wrapper, sem antes ser embrulhado explicitamente. Um tipo wrapper (já embrulhado), também não poderia ser atribuído à uma variável de tipo primitivo, sem antes ser desembrulhada. Vejamos como ficaria para resolver isso no Java 1.4:

```
double preco = 25.78;
Integer idade = new Integer(20);

Double preco2 = new Double(preco);
int idade2 = idade.intValue();
```

Esse tipo de código já foi muito usado antigamente. A legibilidade do código sofria muito quando usávamos essa técnica, mas era necessária em várias situações.

A partir de Java 5, o embrulho e desembrulho são feitos automaticamente, graças ao *autoboxing* e *autounboxing*. Isso quer dizer que, o código abaixo, é totalmente válido em novas versões do Java:

```
Integer quantidade = 10;
Float valorUnitario = 45.35f;

double total = quantidade * valorUnitario;
System.out.println(total);
```

# Trabalhando com arrays

## 10.1. Criando e acessando arrays

Quando precisamos armazenar uma coleção de valores de um mesmo tipo na memória de um programa, podemos trabalhar com arrays, também conhecidos como vetores. Array é um tipo de objeto capaz de armazenar um número fixo de valores de um único tipo.

Para exemplificar, imagine que precisamos guardar a relação de pesos de produtos que uma loja tem no estoque, na memória do programa. Podemos declarar um array do tipo `float`, mas ainda não informamos a capacidade dele.

```
float[] pesosProdutos;
```

Quando declaramos o array `pesosProdutos`, dizemos apenas que a variável deve ser criada, e que nela, teremos um array. Essa variável ainda não referencia um objeto na memória.

Os colchetes indicam que a variável é um tipo de array. Eles podem ser especificados no tipo da variável ou logo após o nome da própria variável. Veja um outro exemplo possível:

```
float pesosProdutos[];
```

A primeira alternativa é a mais usada.

Agora instanciamos um array do tipo `float`, com capacidade para armazenar 10 valores, e atribuímos à variável `pesosProdutos`.

```
float[] pesosProdutos;  
pesosProdutos = new float[10];
```

Opcionalmente, podemos declarar e instanciar o array em uma única linha.

```
float[] pesosProdutos = new float[10];
```

Nesse momento, um objeto de array é criado com 10 posições. Veja que informamos o tamanho do array na instânciação dele.

```
pesosProdutos =
```

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0	1	2	3	4	5	6	7	8	9

Cada item do array é chamado de elemento, e é acessado através de um número de índice. Como você pode ver na imagem acima, o índice começa do número 0, então, em um array de 10 posições, temos elementos do índice 0 ao índice 9.

Quando um array de tipo primitivo é instanciado, cada elemento recebe um valor padrão. No caso de tipos primitivos numéricos, esse valor padrão é 0 (para tipos inteiros) ou 0.0 (para tipos ponto-flutuante).

Para mudar o valor de um determinado elemento, precisamos saber, primeiramente, o índice desse elemento. No exemplo abaixo, alteramos os valores de alguns elementos do array referenciado pela variável pesosProdutos.

```
pesosProdutos[0] = 10.2f;  
pesosProdutos[4] = 98;  
pesosProdutos[9] = 30.5f;
```

Veja a representação do array na memória da máquina virtual:

```
pesosProdutos =
```

10.2	0.0	0.0	0.0	98	0.0	0.0	0.0	0.0	30.5
0	1	2	3	4	5	6	7	8	9

Podemos ainda instanciar um array e inicializar os valores dos elementos, simultaneamente, informando os valores entre chaves.

```
float[] pesosProdutos = { 10.2f, 0, 0, 0, 100, 0, 0, 0, 0, 30.5f };
```

Para acessar qualquer elemento do array e exibir na tela ou efetuar algum cálculo, basta informar o nome da variável, seguido pelo índice do elemento entre colchetes.

```
float[] pesosProdutos = new float[10];
```

```

pesosProdutos[0] = 10.2f;
pesosProdutos[4] = 98;
pesosProdutos[9] = 30.5f;

System.out.println("Somando...");
System.out.println(pesosProdutos[0]);
System.out.println(pesosProdutos[4]);
System.out.println(pesosProdutos[9]);
System.out.println("=");

float total = pesosProdutos[0] + pesosProdutos[4] + pesosProdutos[9];
System.out.println(total);

```

Veja o resultado da execução do código acima:

```

Somando...
10.2
100.0
30.5
=
140.7

```

Se tentarmos acessar um índice fora do intervalo, uma exceção (erro) será gerado em tempo de execução. Por exemplo, podemos tentar executar o código abaixo, mas não teremos sucesso.

```
System.out.println(pesosProdutos[10]);
```

Recebemos o erro abaixo, dizendo que o índice 10 está fora do intervalo do array.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
```

## 10.2. Iterando em arrays

Percorrer um array, também conhecido como iteração, pode ser feito de diversas formas em Java. A mais tradicional, é usando a estrutura de controle for. Veja um exemplo:

```

for (int i = 0; i < pesosProdutos.length; i++) {
    System.out.println("Elemento " + i + " = " + pesosProdutos[i]);
}

```

Usamos a propriedade length do objeto do array para percorrer os elementos dele do início ao fim, e exibimos na tela o índice e o valor do elemento. Agora veja um exemplo mais completo, que também calcula os valores dos elementos do array.

```

float[] pesosProdutos = { 10.2f, 0, 0, 0, 100, 0, 0, 0, 0, 30.5f };
float total = 0;

for (int i = 0; i < pesosProdutos.length; i++) {
    System.out.println("Elemento " + i + " = " + pesosProdutos[i]);
    total += pesosProdutos[i];
}

System.out.println("-----");
System.out.println("Total: " + total);

```

A partir do Java 5, é possível percorrer os arrays usando *enhanced-for*. Trata-se de uma nova sintaxe mais simplificada, que itera entre os elementos do array.

```

float[] pesosProdutos = { 10.2f, 0, 0, 0, 100, 0, 0, 0, 0, 30.5f };
float total = 0;

for (float peso : pesosProdutos) {
    System.out.println(peso);
    total += peso;
}

System.out.println("-----");
System.out.println("Total: " + total);

```

Apesar de *enhanced-for* ter uma implementação mais simples, ele não fornece o índice do elemento atual, enquanto estiver percorrendo o array.

## 10.3. Expandindo arrays

Depois que um array é instanciado, não é possível expandi-lo. A solução, neste caso, é criar um novo array de maior tamanho e copiar todos os elementos do array antigo para o novo.

Felizmente, temos a classe utilitária `java.util.Arrays`, que possui alguns métodos básicos para trabalhar com arrays, inclusive para cópia. Veja um exemplo:

```

// array de 2 elementos
int[] numeros1 = { 5, 8 };

// copiamos para um novo array com 5 elementos
int[] numeros2 = java.util.Arrays.copyOf(numeros1, 5);

for (int numero : numeros2) {
    System.out.println(numero);
}

```

## 10.4. Arrays de objetos

É possível criar arrays de objetos também, e não só de tipos primitivos, como você já aprendeu. Um array de objetos armazena, na verdade, referências de objetos que estão na memória.

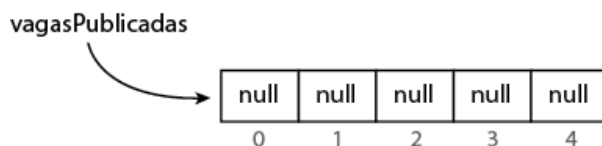
Considere a classe `VagaEmprego` para nosso exemplo.

```
class VagaEmprego {  
    String titulo;  
    double salario;  
}
```

No código abaixo, declaramos e instanciamos um array de `VagaEmprego`, com 5 posições.

```
class ExemploArraysObjetos {  
    public static void main(String[] args) {  
        VagaEmprego[] vagasPublicadas = new VagaEmprego[5];  
    }  
}
```

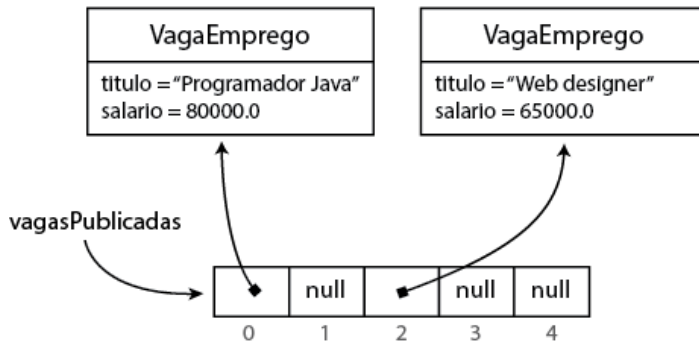
Talvez você esteja imaginando que a instanciamento que fizemos tenha criado 5 instâncias de `VagaEmprego`, mas isso não é verdade. Veja a representação de nosso array na memória.



Quando um array de referências é criado, todos os elementos recebem `null`. Precisamos atribuir referências de objetos para as posições que desejamos. Veja um exemplo.

```
// instancia a vaga e depois atribui ao array  
VagaEmprego vaga = new VagaEmprego();  
vaga.titulo = "Programador Java";  
vaga.salario = 80000;  
vagasPublicadas[0] = vaga;  
  
// instancia e atribui a vaga ao array ao mesmo tempo  
vagasPublicadas[2] = new VagaEmprego();
```

```
vagasPublicadas[2].titulo = "Web designer";
vagasPublicadas[2].salario = 65000;
```



Podemos percorrer o array normalmente, mas temos que tomar cuidado com referências nulas.

```
for (int i = 0; i < vagasPublicadas.length; i++) {
    if (vagasPublicadas[i] != null) {
        System.out.println(vagasPublicadas[i].titulo + " = "
            + vagasPublicadas[i].salario);
    }
}
```

## 10.5. Arrays multidimensionais

Um array multidimensional possui elementos que também são arrays, ou seja, é um array de arrays.

A declaração de arrays multidimensionais é feita usando dois ou mais pares de chaves, como por exemplo `String[][]` para duas dimensões ou `String[][][]` para três dimensões.

Para exemplificar, criaremos um array bidimensional, para armazenar nomes de cidades.

```
String[][] locais = new String[3][];

locais[0] = new String[3];
locais[0][0] = "Uberlândia";
locais[0][1] = "Uberaba";
locais[0][2] = "Belo Horizonte";

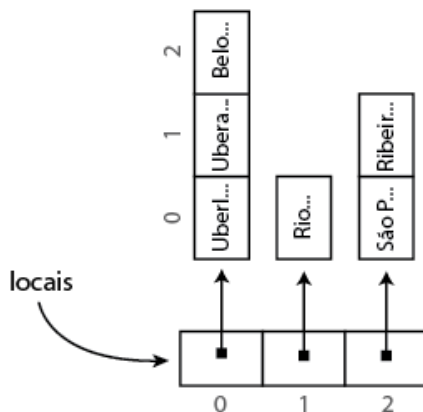
locais[1] = new String[1];
locais[1][0] = "Rio de Janeiro";
```

```
locais[2] = new String[2];
locais[2][0] = "São Paulo";
locais[2][1] = "Ribeirão Preto";
```

Também é possível declarar e inicializar os valores do array de uma vez, usando a sintaxe abaixo:

```
String[][] locais = {
    { "Uberlândia", "Uberaba", "Belo Horizonte" },
    { "Rio de Janeiro" },
    { "São Paulo", "Ribeirão Preto" }
};
```

Para facilitar o entendimento, podemos dizer que a primeira dimensão é representada por linhas, e a segunda por colunas.



O array que criamos não é retangular, pois cada linha tem um número diferente de colunas.

Para iterar em um array multidimensional, precisamos de um for dentro de outro.

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

## 10.6. Lendo os parâmetros da linha de comando

Uma aplicação Java pode aceitar um número variado de argumentos por linha de comando.



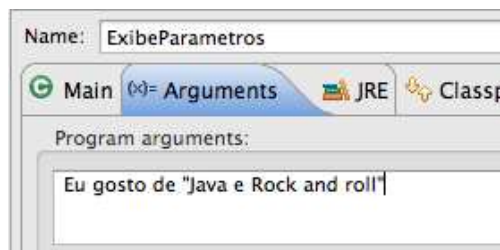
O parâmetro do método `main` de um programa, que é um array do tipo `String`, recebe os argumentos da linha de comando.

```
class ExibeParametros {  
  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            System.out.println("Argumento " + i + ": " + args[i]);  
        }  
    }  
}
```

O código de exemplo acima exibe todos os argumentos recebidos pela aplicação por linha de comando. Veja um exemplo de execução:

```
$ java ExibeParametros Eu gosto de "Java e Rock and roll"  
Argumento 0: Eu  
Argumento 1: gosto  
Argumento 2: de  
Argumento 3: Java e Rock and roll
```

Para executar um programa no Eclipse e passar parâmetros por linha de comando, você deve selecionar a opção **Run Configurations...**, alternar para a aba **Arguments** e digitar os parâmetros no campo **Program arguments**.



## 10.7. Métodos com argumentos variáveis

Suponha que você precise de um método para enviar e-mails para um ou muitos endereços eletrônicos. Você poderia criar um método que recebesse um array.

```
class CorreioEletronico {  
  
    void enviarEmails(String[] emails) {  
        // percorre todos os emails recebidos como parâmetro  
        for (String email : emails) {  
            // envia e-mail  
            System.out.println("E-mail enviado para " + email);  
        }  
    }  
}
```

```

    }
}

```

Para chamar o método `enviarEmails`, você precisaria declarar, instanciar e inicializar um array com e-mails.

```

String[] emails = new String[2];
emails[0] = "joaodascouves@algaworks.com";
emails[1] = "zemane@algaworks.com";

CorreioEletronico correio = new CorreioEletronico();
correio.enviarEmails(emails);

```

Ainda é possível melhorar o código acima, declarando, instanciando e inicializando o array de e-mails em uma única instrução:

```

correio.enviarEmails(new String[]{ "joaodascouves@algaworks.com",
    "zemane@algaworks.com" });

```

Apesar de ter facilitado bastante, o código ainda pode ficar mais simples e legível usando *varargs*.

A versão 5 do Java trouxe um recurso que permite um número variável de argumentos para métodos, que permite maior flexibilidade e simplicidade ao código.

O *varargs*, ou argumentos variáveis, permite que o programador declare um método que pode receber um número variável de parâmetros. O *varargs* deve ser o último argumento na lista de parâmetros.

Usando *varargs*, o método `enviarEmails` poderia ser programado da seguinte forma:

```

void enviarEmails(String... emails) {
    // percorre todos os emails recebidos como parâmetro
    for (String email : emails) {
        // envia e-mail
        System.out.println("E-mail enviado para " + email);
    }
}

```

A variável `emails`, recebida como parâmetro, é um array, mas podemos chamar o método sem precisar declarar e inicializar um array.

```

correio.enviarEmails("joaodascouves@algaworks.com",
    "zemane@algaworks.com");

```

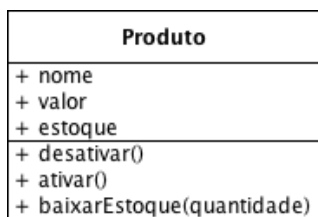
# Diagrama de classes, construtores e encapsulamento

## 11.1. Introdução ao diagrama de classes

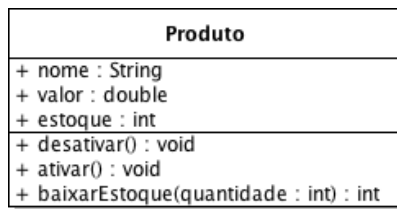
Os sistemas orientados a objetos podem ser modelados com auxílio da UML (Unified Modeling Language), uma linguagem de modelagem para especificar, modelar, visualizar e documentar projetos, baseando em diagramas.

A UML é composta por diagramas de classes, diagramas de sequências, diagramas de casos de uso, e diversos outros. Você aprenderá o básico sobre os diagramas de classes, que tem o objetivo de mostrar a estrutura estática do sistema ou parte dele. Este diagrama exibe as entidades do sistema e seus relacionamentos.

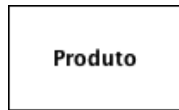
Uma classe é representada por um retângulo, contendo até três compartimentos. O primeiro contém a descrição ou nome da classe, o segundo compartimento lista os atributos da classe e o terceiro, os métodos da classe.



Se for necessário comunicar com mais detalhes através do diagrama de classes, podemos também especificar os tipos das variáveis de instância, parâmetros e retorno de métodos.



Se os detalhes não são importantes, podemos ocultar os compartimentos dos atributos e métodos.



Os relacionamentos entre classes devem ser desenhados também, podendo informar alguns detalhes, como o nome da associação e a multiplicidade.



## 11.2. Construtores

Os construtores fazem a função de iniciação dos objetos instanciados. Toda classe em Java tem pelo menos um construtor.

Quando não declaramos explicitamente um construtor, a classe recebe um padrão, também conhecido como *construtor default*.

```

class Cliente {

    // esta classe tem construtor default

}
  
```

A sintaxe dos construtores é parecida com a de métodos, mas eles não possuem retorno e o nome deve ser **exatamente** igual ao da classe.

```

class Cliente {

    Cliente() {
        // isso é um construtor
    }

}
  
```

```
}
```

Os dois primeiros exemplos possuem construtores que não recebem argumentos e também não fazem nada! A diferença é que, no primeiro código, não declaramos um construtor, e por isso foi usado o construtor padrão. No segundo exemplo, declaramos um construtor padrão (sem parâmetros), que substituiu o construtor implícito, mas que também não faz nada.

Agora, vamos imprimir uma mensagem dentro do construtor.

```
class Cliente {  
  
    Cliente() {  
        System.out.println("Construindo cliente...");  
    }  
  
}
```

Para chamar um construtor, basta instanciarmos um objeto da classe, com a palavra-chave `new`.

```
class TesteConstrutor {  
  
    public static void main(String[] args) {  
        Cliente cliente = new Cliente();  
    }  
  
}
```

A execução da classe `TesteConstrutor` exibirá na saída a mensagem "Construindo cliente...".

Podemos adicionar parâmetros aos construtores, assim como fazemos em métodos.

```
class Cliente {  
  
    String nome;  
  
    Cliente(String nome) {  
        this.nome = nome;  
    }  
  
}
```

Quando criamos um construtor com parâmetros, como o exemplo acima, não conseguimos mais instanciar um objeto da classe usando o construtor sem parâmetros, pois o *construtor default* é anulado.

```
// não compila
Cliente cliente = new Cliente();
```

Agora, para instanciar um objeto do tipo `Cliente`, somos obrigados a passar como parâmetro o nome dele.

```
// compila
Cliente cliente = new Cliente("João das Couves");
```

Podemos ainda criar uma sobrecarga de construtores, adicionando um construtor sem parâmetros e outros com parâmetros diversos.

```
class Cliente {

    String nome;
    String cpf;

    Cliente() {
    }

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

    Cliente(String nome, String cpf) {
        this.nome = nome;
        this.cpf = cpf;
    }

}
```

Agora podemos escolher qual construtor queremos usar na instânciação de objetos do tipo `Cliente`.

```
// usa o construtor padrão (sem parâmetros)
Cliente cliente1 = new Cliente();

// usa o construtor que recebe o nome
Cliente cliente2 = new Cliente("João das Couves");

// usa o construtor que recebe o nome e cpf
Cliente cliente3 = new Cliente("João das Couves", "12312312312");
```

Um construtor pode chamar outro, para aproveitar algum processamento. No código abaixo, o construtor `Cliente(String, String)` chama `Cliente(String)`, que então chama `Cliente()`.

```
class Cliente {

    String nome;
    String cpf;
```

```

    Cliente() {
        System.out.println("Construindo cliente...");
    }

    Cliente(String nome) {
        this();
        System.out.println("Cliente com nome " + nome);
        this.nome = nome;
    }

    Cliente(String nome, String cpf) {
        this(nome);
        System.out.println("Cliente com CPF " + cpf);
        this.cpf = cpf;
    }
}

```

A chamada de outro construtor, através do comando `this()`, deve ser a primeira instrução do construtor.

Para testar, vamos instanciar um novo cliente, usando o construtor que recebe o nome e CPF do cliente.

```

class TesteConstrutor {

    public static void main(String[] args) {
        Cliente cliente = new Cliente("João das Couves",
            "12312312312");
    }
}

```

A execução do último exemplo exibe na saída:

```

Construindo cliente...
Cliente com nome João das Couves
Cliente com CPF 12312312312

```

### 11.3. Modificadores de acesso public e private

Os modificadores de acesso em Java, também conhecidos como modificadores de visibilidade, permitem controlar o acesso a classes, atributos, métodos e construtores. Existem 4 tipos diferentes de modificadores de acesso, mas por enquanto, estudaremos apenas o `public` e `private`.

## Modificadores em classes

Quando o modificador de acesso `public` é usado em uma classe, ela fica visível (acessível) para todas as outras classes. Talvez seja um pouco difícil entender o que isso significa nesse momento, mas ficará mais claro no futuro, quando você aprender sobre os outros modificadores (*default* e *protected*, principalmente).

```
public class Produto {  
  
}
```

A classe acima pode ser instanciada por *qualquer* outra classe, sem restrições.

Talvez você esteja se perguntando: "mas já não era assim?". Não! Quando não especificamos `public` na declaração da classe, ela não pode ser instanciada por qualquer outra classe. Existem restrições!

```
// esta classe não pode ser instanciada por qualquer outra classe  
class Produto {  
  
}
```

Você pode ficar curioso(a), mas ainda não chegou a hora de você aprender mais sobre o modificador de acesso padrão, quando nenhum é especificado. Em um outro capítulo, você descobrirá a diferença.

O modificador `private` não pode ser usado em declarações de classes, com exceção de classes aninhadas, que não abordaremos neste livro.

## Modificadores em atributos, métodos e construtores

O modificador de acesso `public` pode ser usado em atributos, métodos e construtores.

```
public class Produto {  
  
    public int estoque;  
  
    public Produto(int estoque) {  
        this.estoque = estoque;  
    }  
  
    public void zerarEstoque() {  
        this.estoque = 0;  
    }  
  
}
```



No exemplo acima, o atributo `estoque`, o construtor e o método `zerarEstoque` podem ser acessados por qualquer outra classe.

```
Produto produto = new Produto(10);
produto.estoque = 20;
produto.zerarEstoque();
```

Mais uma vez, você entenderá melhor quando estudarmos outros modificadores.

O modificador `private` torna o membro privado, incapaz de ser acessado por outras classes, a não ser a própria classe a qual o membro pertence.

```
public class Produto {

    private int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
    }

    public void zerarEstoque() {
        this.estoque = 0;
    }

}
```

Veja que alteramos o modificador do atributo `estoque` para `private`. A partir de agora, não conseguimos mais acessar o atributo a partir de uma classe externa, mas o acesso pelo construtor e pelo método `zerarEstoque` são válidos, pois pertencem à mesma classe.

```
public class UmaClasseQualquer {

    public static void main(String[] args) {
        Produto produto = new Produto(10);
        produto.estoque = 20; // não compila
    }

}
```

Quando não queremos que um construtor seja usado por classes externas, podemos torná-lo privado, mas temos que deixar pelo menos um construtor acessível. Se não fizermos isso, tornamos impossível a instanciação de um objeto da classe.

```
public class Produto {

    private int estoque;

    public Produto() {
```

```

        this(10);
    }

    private Produto(int estoque) {
        this.estoque = estoque;
    }

    public void zerarEstoque() {
        this.estoque = 0;
    }

    public void imprimirEstoque() {
        System.out.println("Estoque: " + this.estoque);
    }
}

```

No último código, o único construtor público chama um construtor privado, que inicializa a quantidade em estoque com 10 unidades.

```

// chama construtor público
Produto produto = new Produto();

// imprime "Estoque: 10" na saída
produto.imprimirEstoque();

```

O código abaixo não compila, porque o construtor que tentamos usar é privado.

```

// não compila
Produto produto = new Produto(10);

```

Um método privado também só pode ser acessado pela própria classe, e é útil para, principalmente, reutilizar código em mais de um método.

```

public class Produto {

    private int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
        this.imprimirEstoque();
    }

    public void zerarEstoque() {
        this.estoque = 0;
        this.imprimirEstoque();
    }

    private void imprimirEstoque() {
        System.out.println("Estoque: " + this.estoque);
    }
}

```

```
}
```

No exemplo acima, o método `imprimirEstoque` foi definido como privado, e por isso, só pode ser acessado pela própria classe. O construtor e o método `zerarEstoque` chamam o método `imprimirEstoque` depois de alterar a quantidade em estoque.

```
Produto produto = new Produto(20);  
produto.zerarEstoque();
```

A execução do último exemplo exibe na saída:

```
Estoque: 20  
Estoque: 0
```

O código abaixo não compila, pois `imprimirEstoque` é privado.

```
Produto produto = new Produto(20);  
produto.zerarEstoque();  
produto.imprimirEstoque(); // não compila
```

## 11.4. Encapsulamento

Encapsulamento é um dos conceitos fundamentais da orientação a objetos. É uma técnica que torna os atributos da classe privados e fornece acesso a eles através de métodos públicos. Na seção anterior, acabamos usando conceitos de encapsulamento, sem mesmo citar sobre isso.

Como você já sabe, atributos privados só podem ser acessados pela própria classe, portanto, eles ficam "protegidos" de outras classes, e a única forma de acesso é pelos métodos públicos.

```
public class Produto {  
  
    private int estoque;  
  
    public void adicionarEstoque(int estoque) {  
        this.estoque += estoque;  
        this.verificarEstoqueMinimo();  
    }  
  
    public void retirarEstoque(int estoque) {  
        this.estoque -= estoque;  
        this.verificarEstoqueMinimo();  
    }  
  
    private void verificarEstoqueMinimo() {
```

```

        if (this.obterEstoque() < 5) {
            System.out.println("Abaixo do estoque mínimo: "
                               + this.obterEstoque());
        }

        public int obterEstoque() {
            return this.estoque;
        }
    }
}

```

Na classe Produto acima, encapsulamos o atributo estoque e centralizamos o controle de estoque nos métodos adicionarEstoque e retirarEstoque. Essa centralização facilita a manutenção do código, pois outras classes que precisarem adicionar ou retirar itens do estoque, não precisam se preocupar com a lógica por trás disso. Se alguma classe precisar saber a quantidade de itens que um produto possui no estoque, poderá ainda usar o método obterEstoque.

```

Produto produto = new Produto();

produto.adicionarEstoque(10);
System.out.println(produto.obterEstoque());

produto.retirarEstoque(8);
System.out.println(produto.obterEstoque());

```

## Encapsulamento no mundo real

No mundo real, podemos notar o uso de encapsulamento em quase todos os objetos que usamos.

Por exemplo, a TV de sua casa. Existem milhares de componentes dentro de um aparelho de TV, mas nós interagimos apenas com uma interface pública, através do controle remoto ou botões no próprio equipamento.

Se quisermos ligar a TV, não precisamos conhecer os detalhes eletrônicos que estão por trás, não há necessidade de pegarmos uma chave de fenda e abrirmos a TV para ligar os circuitos que ativam a imagem e som. Basta apertarmos o botão "ligar", que é público. Os detalhes técnicos estão encapsulados, o botão "ligar" sabe o que deve ser feito tecnicamente, e dentro do próprio aparelho, existem muitos outros níveis de encapsulamento, para facilitar a manutenção e extensão das funcionalidades do equipamento.

## 11.5. JavaBeans

JavaBeans são classes Java reutilizáveis, que encapsulam as variáveis de instância em um único objeto (o bean).

Para uma classe Java ser considerada um JavaBean, ela deve ter um construtor padrão (sem argumentos) e permitir o acesso às variáveis de instância através de métodos acessores, conhecidos como *getters* e *setters*.

```
public class Produto {  
  
    private String nome;  
    private int estoque;  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public int getEstoque() {  
        return this.estoque;  
    }  
  
    public void setEstoque(int estoque) {  
        this.estoque = estoque;  
    }  
  
}
```

A classe Produto é um JavaBean! Veja que expomos os atributos nome e estoque através dos métodos *getters* e *setters*.

```
Produto produto = new Produto();  
produto.setNome("Mouse sem fio");  
produto.setEstoque(10);
```

```
System.out.println(produto.getNome() + " - " + produto.getEstoque());
```

Embora o padrão diga que os atributos devem ser expostos pelos métodos de acessibilidade, você não é obrigado a expor tudo! Por exemplo, nossa classe Produto poderia expor os métodos de alteração de estoque usando formas mais orientadas a objetos, sem seguir a risca o padrão.

```
public class Produto {  
  
    private String nome;
```

```

private int estoque;

public String getName() {
    return this.nome;
}

public void setName(String nome) {
    this.nome = nome;
}

public int getEstoque() {
    return this.estoque;
}

public void adicionarEstoque(int estoque) {
    this.estoque += estoque;
}

public void retirarEstoque(int estoque) {
    this.estoque -= estoque;
}
}

```

Antes de sair gerando os *getters* e *setters* para todos os atributos, pense bem na necessidade disso.

Diversos *frameworks* de desenvolvimento exigem que os objetos que eles interagem sejam JavaBeans. Por isso, esse padrão é muito importante em Java.

O Eclipse possui uma ferramenta que cria os métodos *getters* e *setters* para os atributos selecionados. Pressione Cmd+Opt+S (Mac) ou Alt+Shift+S (Windows) e clique na opção **Generate Getters and Setters**.

## JavaBeans não são EJBs

Não confunda JavaBeans, padrão explicado nessa seção, com EJBs (Enterprise JavaBeans).

EJB é uma especificação da Java EE, e componentes desse tipo rodam em servidores de aplicação. É um assunto mais avançado, que você pode estudar no futuro.

# Pacotes, outros modificadores e enumerações

## 12.1. Organizando os projetos em pacotes

Quando estamos desenvolvendo projetos reais, geralmente, temos centenas ou milhares de classes que se comunicam para executar o que o software se propõe. Surge então a necessidade de organizarmos nossas classes em diretórios diferentes, para separar sistemas, módulos ou responsabilidades.

Em Java, existe o conceito de *packages* (pacotes), que são diretórios e subdiretórios que organizam as classes.

Além de organizar as classes de projetos, os pacotes são importantes para evitar conflitos de nomes de classes. Por exemplo, imagine se você pegar uma biblioteca de classes de um colega que não organizou em pacotes. Dentro dessa biblioteca de classes, existe uma com o nome *Deducacao*. Se você precisar criar uma classe sua, também com o nome *Deducacao*, será impossível se não colocá-la dentro de um pacote, pois os nomes entrariam em conflito. Por isso, é uma boa prática colocar suas classes **sempre** dentro de pacotes, e não só isso, mas nomear os pacotes de acordo com um padrão usado no mundo todo, que veremos daqui a pouco.

```
package com.algaworks.rh.folha;

public class Deducao {

    private String nome;
    private double valor;

    public String getNome() {
```

```

        return this.nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

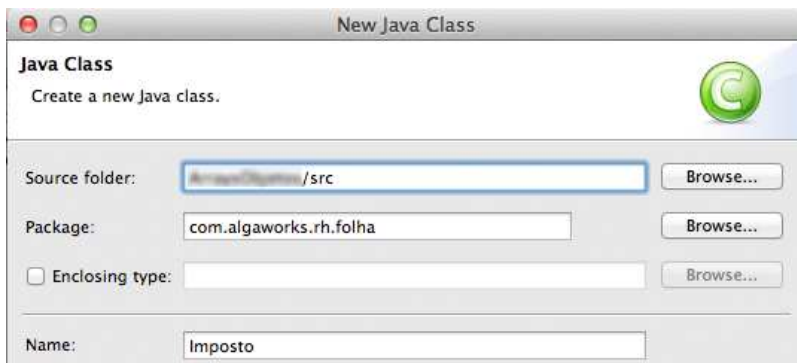
    public double getValor() {
        return this.valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}

```

Colocamos a classe `Deducao` dentro do pacote `com.algaworks.rh.folha` usando a palavra-chave `package` no início do arquivo de código-fonte. Isso quer dizer que a classe deve ficar dentro do diretório `com/algaworks/rh/folha`.

Você não precisa criar os diretórios manualmente. No Eclipse, quando você está criando uma classe, pode informar o nome do pacote no campo **Package**.



A separação de nomes de subpacotes é feita por um ponto, mas o Eclipse criará os diretórios automaticamente para você.

Existe uma convenção que diz que nomes de pacotes devem iniciar com o domínio da empresa na internet ao contrário, e deve ser escrito tudo em letras minúsculas. No caso do pacote `com.algaworks.com.rh.folha`, veja que *algaworks.com* é o domínio da empresa na internet.

É importante seguir esse padrão, pois assim, torna-se praticamente impossível que duas classes com o mesmo nome entrem em conflito.



Depois do nome do domínio ao contrário, geralmente, usa-se o nome do sistema, módulo e submódulo. Por exemplo:

```
com.algaworks.nomedosistema.nomedomodulo.nomedosubmodulo
com.algaworks.rh.folha
com.algaworks.rh.demissao
com.algaworks.rh.admissao
com.algaworks.financeiro.contaspagar
com.algaworks.financeiro.fluxocaixa
com.algaworks.seguranca.web
```

Se sua empresa ou você não tiver um domínio registrado na internet, pense em usar o domínio do site onde você hospeda o código-fonte de seu projeto. Por exemplo, se seu projeto estiver no GitHub, use `com.github.nomeusuario.nomeprojeto`.

Você irá perceber que as classes de APIs Java não usam um nome de domínio ao contrário em nomes de pacotes. Não há nada de errado nisso, pois o padrão foi criado apenas para terceiros usarem.

## Usando classes de pacotes

Uma classe pode usar todas as outras classes do mesmo pacote sem precisar especificar o nome do pacote.

```
package com.algaworks.rh.folha;

public class Holerite {

    private String nomeFuncionario;
    private double salarioBruto;
    private Deducao[] deducoes;

    public String getNomeFuncionario() {
        return nomeFuncionario;
    }

    public void setNomeFuncionario(String nomeFuncionario) {
        this.nomeFuncionario = nomeFuncionario;
    }

    public double getSalarioBruto() {
        return salarioBruto;
    }

    public void setSalarioBruto(double salarioBruto) {
        this.salarioBruto = salarioBruto;
    }
}
```

```

    public Deducao[] getDeducoes() {
        return deducoes;
    }

    public void setDeducoes(Deducao[] deducoes) {
        this.deducoes = deducoes;
    }

    public double getSalarioLiquido() {
        double salarioLiquido = getSalarioBruto();

        for (Deducao deducao : getDeducoes()) {
            salarioLiquido -= deducao.getValor();
        }

        return salarioLiquido;
    }
}

```

A classe `Holerite` referencia a classe `Deducao`, mas não foi necessário informar ao compilador onde ele pode encontrar a classe `Deducao`, pois ela está no mesmo pacote.

Agora, vamos criar uma classe em outro pacote e usar `Deducao` e `Holerite`.

```

package com.algaworks.rh;

public class TesteFolha {

    public static void main(String[] args) {
        com.algaworks.rh.folha.Deducao inss =
            new com.algaworks.rh.folha.Deducao();
        inss.setNome("INSS");
        inss.setValor(64);

        com.algaworks.rh.folha.Deducao valeTransporte =
            new com.algaworks.rh.folha.Deducao();
        valeTransporte.setNome("Vale transporte");
        valeTransporte.setValor(48);

        com.algaworks.rh.folha.Holerite holerite =
            new com.algaworks.rh.folha.Holerite();
        holerite.setNomeFuncionario("João das Couves");
        holerite.setSalarioBruto(800);
        holerite.setDeducoes(new com.algaworks.rh.folha.Deducao[]
            { inss, valeTransporte });

        System.out.println("Salário líquido: "
            + holerite.getSalarioLiquido());
    }
}

```

Note que, para usar as classes de outro pacote, tivemos que informar o *nome do pacote* + *nome da classe* em todas as referências. Isso é chamado de *Fully Qualified Name*, ou *Nome totalmente qualificado*. Quando uma classe está dentro de um pacote, o nome completo dela passa a ser o nome do pacote mais o nome da classe.

Usar nomes totalmente qualificados nas referências de classes pode deixar o código ilegível. Podemos importar as classes do outro pacote, para usar apenas os nomes simples nas referências.

```
package com.algaworks.rh;

import com.algaworks.rh.folha.Deducao;
import com.algaworks.rh.folha.Holerite;

public class TesteFolha {

    public static void main(String[] args) {
        Deducao inss = new Deducao();
        inss.setNome("INSS");
        inss.setValor(64);

        Deducao valeTransporte = new Deducao();
        valeTransporte.setNome("Vale transporte");
        valeTransporte.setValor(48);

        Holerite holerite = new Holerite();
        holerite.setNomeFuncionario("João das Couves");
        holerite.setSalarioBruto(800);
        holerite.setDeducoes(new Deducao[] { inss, valeTransporte });

        System.out.println("Salário líquido: "
            + holerite.getSalarioLiquido());
    }
}
```

Usando a palavra-chave `import`, importamos as classes `Deducao` e `Holerite` para serem usadas por `TesteFolha`.

As declarações de importação devem vir depois da declaração do nome do pacote no arquivo (se houver).

Podemos ainda, importar todas as classes de um pacote.

```
package com.algaworks.rh;

import com.algaworks.rh.folha.*;

public class TesteFolha {
```

```
...  
}
```

Quando usamos o coringa `*`, importamos tudo do pacote. Você não precisa se preocupar com a performance de seu programa por estar importando de um pacote classes que nem serão usadas. Isso não degrada a performance, pois o compilador é inteligente e não carrega classes que você não precisa.

Todavia, é uma boa prática importar as classes uma por uma, pois deixando os nomes explícitos, a legibilidade é melhorada e conflitos de nomes são evitados.

Por padrão, o Eclipse importa as classes uma por uma, mas se você tiver classes importando pacotes inteiros e quiser organizar os *imports*, use o atalho `Cmd+Shift+O` (Mac) ou `Ctrl+Shift+O` (Windows).

O `java.lang` é um pacote fundamental da linguagem Java, por isso, as classes dele não precisam ser importadas explicitamente. Por exemplo, a classe `java.lang.String` pode ser usada sem fazer um `import java.lang.String` antes.

## 12.2. Modificador de acesso default

Quando não especificamos nenhum modificador de acesso em nossas classes, atributos, construtores ou métodos, esses elementos recebem o modificador de acesso *default* (padrão).

```
package com.algaworks.estoque;  
  
class Produto {  
}
```

A classe `Produto` possui o modificador padrão, por isso, fica visível apenas para classes do mesmo pacote.

```
package com.algaworks.comercial;  
  
import com.algaworks.estoque.Produto;  
  
class FechamentoPedido {  
  
    public static void main(String[] args) {  
        // não compila, pois não tem acesso  
        Produto produto = new Produto();  
    }  
}
```

```
}
```

Para conseguirmos compilar a classe `FechamentoPedido`, precisamos incluir o modificador de acesso `public` na classe `Produto` ou mover a classe `FechamentoPedido` para o pacote `com.algaworks.estoque`.

```
package com.algaworks.estoque;

public class Produto {

    private String nome;
    private int estoque;

    String getNome() {
        return this.nome;
    }

    void setNome(String nome) {
        this.nome = nome;
    }

    int getEstoque() {
        return this.estoque;
    }

    void adicionarEstoque(int estoque) {
        this.estoque += estoque;
    }

    void retirarEstoque(int estoque) {
        this.estoque -= estoque;
    }

}
```

Agora a classe `Produto` é pública, mas todos os métodos não possuem modificadores de acesso (ou seja, são *default*).

```
package com.algaworks.comercial;

import com.algaworks.estoque.Produto;

class FechamentoPedido {

    public static void main(String[] args) {
        Produto produto = new Produto();

        // não compila, pois não tem acesso
        produto.setNome("Mouse sem fio");
        produto.adicionarEstoque(10);
    }
}
```

```

        System.out.println(produto.getNome() + " - "
            + produto.getEstoque());
    }
}

```

Continuamos sem conseguir compilar a classe `FechamentoPedido`, pois agora todos os métodos são visíveis apenas para o pacote `com.algaworks.estoque`. Podemos torná-los públicos ou mover a classe `FechamentoPedido` para o mesmo pacote.

```

package com.algaworks.estoque;

class FechamentoPedido {

    public static void main(String[] args) {
        Produto produto = new Produto();

        // Agora sim compila, está no mesmo pacote
        produto.setNome("Mouse sem fio");
        produto.adicionarEstoque(10);

        System.out.println(produto.getNome() + " - "
            + produto.getEstoque());
    }
}

```

Agora que movemos a classe `FechamentoPedido` para o pacote `com.algaworks.estoque`, conseguimos compilar normalmente.

A melhor solução, muitas vezes, não é mover as classes para o mesmo pacote, mas tornar público os membros que necessitam de acesso por classes de outros pacotes.

## 12.3. Membros de classe

Quando diversos objetos são instanciados a partir de uma mesma classe, cada objeto possui suas variáveis de instâncias com valores distintos.

```

public class Usuario {

    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

```

```

    }

}

public class TesteUsuario {

    public static void main(String[] args) {
        Usuario usuario1 = new Usuario("joao");
        Usuario usuario2 = new Usuario("sebastiao");

        System.out.println(usuario1.getNome());
        System.out.println(usuario2.getNome());
    }

}

```

Em determinadas situações, precisamos de variáveis globais, comuns para todas os objetos instanciados. Por exemplo, gostaríamos de contar o número de usuários *logados*.

```

public class Usuario {

    private int totalUsuariosLogados;
    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void logar() {
        this.totalUsuariosLogados++;
    }

    public void deslogar() {
        this.totalUsuariosLogados--;
    }

    public int getTotalUsuariosLogados() {
        return totalUsuariosLogados;
    }

}

```

A variável `totalUsuariosLogados` ainda não é comum a todos os objetos, ela é uma variável de instância!

```

Usuario usuario1 = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");

```

```
usuario1.logar();
usuario2.logar();

System.out.println(usuario1.getTotalUsuariosLogados());
System.out.println(usuario2.getTotalUsuariosLogados());
```

Quando executamos o código acima, a saída é:

```
1
1
```

Isso quer dizer que cada instância de `Usuario` armazenou um total de usuários logados.

Para tornar uma variável comum a todos os objetos, precisamos transformá-la em uma variável de classe, adicionando a palavra-chave `static` na declaração.

```
public class Usuario {

    private static int totalUsuariosLogados;

    ...

}
```

Se executarmos novamente o teste anterior, a saída será:

```
2
2
```

Agora, a variável `totalUsuariosLogados` é da classe, e não mais da instância, por isso, o valor dela é compartilhado com todos os objetos.

Se a variável `totalUsuariosLogados` fosse pública, poderíamos acessá-la diretamente pela classe.

```
public class Usuario {

    public static int totalUsuariosLogados;

    ...

}
```

Veja o exemplo:

```
Usuario usuario1 = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");
```



```

usuariol.logar();
usuario2.logar();

System.out.println(Usuario.totalUsuariosLogados);

```

Note que acessamos a variável usando o nome da classe mais o nome da variável, sem usar uma instância de `Usuario`.

Deixar a variável `totalUsuariosLogados` pública pode não ser muito legal, pois quem controla o incremento e decremento dela são os métodos da classe `Usuario`. Vamos colocar novamente o modificador `private` nela e tornar o método `getTotalUsuariosLogados` estático.

```

public class Usuario {

    private static int totalUsuariosLogados;

    ...

    public static int getTotalUsuariosLogados() {
        return totalUsuariosLogados;
    }

}

```

Agora, para acessarmos o total de usuários logados, basta chamarmos o método estático (da classe).

```

Usuario usuariol = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");

usuariol.logar();
usuario2.logar();

System.out.println(Usuario.getTotalUsuariosLogados());

```

## 12.4. Modificador final e constantes

O modificador `final` pode ser usado em variáveis para indicar que elas não podem ser alteradas, depois da primeira atribuição.

```

public class VariavelLocalFinal {

    public static void main(String[] args) {
        final String nome;
        nome = "Maria";
    }
}

```

```

        // não compila - a variável nome já foi atribuída
        nome = "João";
    }
}

```

Quando usamos o modificador `final` em uma variável de instância, somos obrigados a inicializá-la durante a declaração dela ou no construtor.

```

public class Pessoa {

    final String nome;

    public Pessoa(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

}

```

## Constantes em Java

Quando combinamos os modificadores `static` e `final` na declaração de uma variável, definimos uma variável de classe que não pode ser alterada, ou seja, uma **constante**.

Antes de criarmos nossa primeira constante, veja o código a seguir.

```

public class Carro {

    public void acelerar(int velocidade) {
        if (velocidade > 180) {
            System.out.println("Velocidade acima da permitida.");
        } else {
            System.out.println("Acelerando...");
        }
    }

}

```

O método `acelerar` recebe uma velocidade como parâmetro e restringe a aceleração para até 180 km/h. O número 180 está fixo no meio da lógica de aceleração. Isso é chamado de **número mágico**, pois outros programadores podem não entendê-lo ou precisar replicar esse número em outros métodos. O ideal, neste caso, seria colocar o número 180 em uma constante.

```

public class Carro {

    public static final int VELOCIDADE_MAXIMA = 180;

    public void acelerar(int velocidade) {
        if (velocidade > VELOCIDADE_MAXIMA) {
            System.out.println("Velocidade acima da permitida.");
        } else {
            System.out.println("Acelerando...");
        }
    }
}

```

O funcionamento da classe Carro não irá mudar em nada, mas deixamos o código mais legível, além de podermos reaproveitar a constante VELOCIDADE\_MAXIMA.

De acordo com a convenção de codificação Java, os nomes de constantes devem ser escritos em letras maiúsculas, com as palavras separadas por *underscore* (\_).

## 12.5. Enumerações

Uma enumeração é um tipo de dado abstrato, que possui valores finitos escolhidos pelo programador. Enumerações são usadas normalmente para variáveis absolutas, que identificam "alguma coisa" em um "conjunto de coisas" e não possuem uma ordem numérica. Em tempo de execução, as enumerações são implementadas como inteiros (cada identificador possui um valor inteiro único).

Em versões anteriores ao Java 5, era comum representarmos uma enumeração com valores inteiros, como no exemplo abaixo:

```

public class Cliente {

    public static final int STATUS_ATIVO = 0;
    public static final int STATUS_EM_ANALISE = 1;
    public static final int STATUS_CANCELADO = 2;
    public static final int STATUS_BLOQUEADO = 3;

    private int status;

    public int getStatus() {
        return this.status;
    }

    public void setStatus(int status) {
        this.status = status;
    }
}

```

```
}
```

No exemplo acima, representamos status de clientes. Esse padrão gera muitos problemas:

- Não é *typesafe*: como o status é apenas um inteiro, você pode informar qualquer outro inteiro onde um status é requerido.
- Pode causar problemas de ambiguidade: você precisa utilizar prefixos nos nomes das constantes (neste caso, utilizamos STATUS\_) para evitar colisões com outra enumeração inteira.
- Fragilidade: como as enumerações de inteiros são constantes, elas são compiladas com valores fixos. Se uma nova constante é adicionada entre duas outras constantes ou o número de alguma constante já usada é alterado, as classes que as utilizam precisam ser revisadas e recompiladas, ou o comportamento do programa ficará indefinido.
- Valores impressos não são informativos: como são apenas inteiros, se você imprimir uma enumeração, você verá apenas um número, que não diz nada sobre o que ela representa.

```
public class TesteClienteSemEnumeracao {  
  
    public static void main(String[] args) {  
        Cliente cliente = new Cliente();  
  
        cliente.setStatus(Cliente.STATUS_ATIVO);  
        System.out.println(cliente.getStatus());  
  
        // isso deveria ser inválido, mas compila normalmente!  
        cliente.setStatus(10);  
        System.out.println(cliente.getStatus());  
    }  
}
```

A partir do Java 5, a linguagem suporta tipos enumerados.

```
public class Cliente {  
  
    public enum Status { ATIVO, EM_ANALISE, CANCELADO, BLOQUEADO }  
  
    private Status status;  
  
    public Status getStatus() {  
        return this.status;  
    }  
}
```

```

    }

    public void setStatus(Status status) {
        this.status = status;
    }
}

```

Agora que o status do cliente é de um tipo enumerado, fica impossível passar valores não definidos na enumeração Status. Além disso, as instruções de impressão na tela exibem o nome da constante, e não um número.

```

public class TesteClienteComEnumeracao {

    public static void main(String[] args) {
        Cliente cliente = new Cliente();

        cliente.setStatus(Cliente.Status.ATIVO);
        System.out.println(cliente.getStatus());

        cliente.setStatus(Cliente.Status.EM_ANALISE);
        System.out.println(cliente.getStatus());
    }
}

```

As enumerações em Java são muito mais poderosas que se parecem. A nova declaração de enum define um novo tipo enum. Além de resolver todos os problemas mencionados acima, as enumerações permitem que você adicione métodos e atributos, implemente interfaces, além de serem comparáveis e serializáveis. Não abordaremos esses recursos mais avançados neste livro, mas se tiver curiosidade, pesquise na internet e faça alguns exemplos.

# Orientação a objetos avançada

## 13.1. Herança e sobrescrita

Em um sistema de banco, temos uma classe que representa uma transferência entre contas.

```
public class Transferencia {

    private String descricao;
    private double valor;
    private String data;
    private String contaOrigem;
    private String contaDestino;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters
}
```

Declaramos o atributo `data` com o tipo `String` porque ainda não estudamos um tipo específico para datas, mas não se preocupe!

Agora, precisamos também de algo que represente uma transação de pagamento de boleto. Podemos tentar adicionar alguns atributos na classe Transferencia e até mesmo alterar o nome dela para Transacao, para ficar mais genérico.

```
public class Transacao {

    // atributos comuns
    private String descricao;
    private double valor;
    private String data;

    // atributos de uma transferência
    private String contaOrigem;
    private String contaDestino;

    // atributos de um pagamento de boleto
    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());

        if (this.getContaOrigem() != null) {
            System.out.println("Conta de origem: "
                + this.getContaOrigem());
            System.out.println("Conta de destino: "
                + this.getContaDestino());
        } else {
            System.out.println("Linha digitável: "
                + this.getLinhaDigitavel());
            System.out.println("Data de vencimento: "
                + this.getDataVencimento());
            System.out.println("Cedente: " + this.getCedente());
        }

        System.out.println();
    }

    // getters e setters
}
```

A classe Transacao que acabamos de criar ficou muito feia! Fizemos uma gambiarra! Uma classe que representa duas coisas ao mesmo tempo só poderia resultar nisso.

Vamos esquecer essa tentativa e tentar separar em duas classes, Transferencia (a primeira versão que criamos) e PagamentoBoleto.

```

public class PagamentoBoleto {

    private String descricao;
    private double valor;
    private String data;
    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters

}

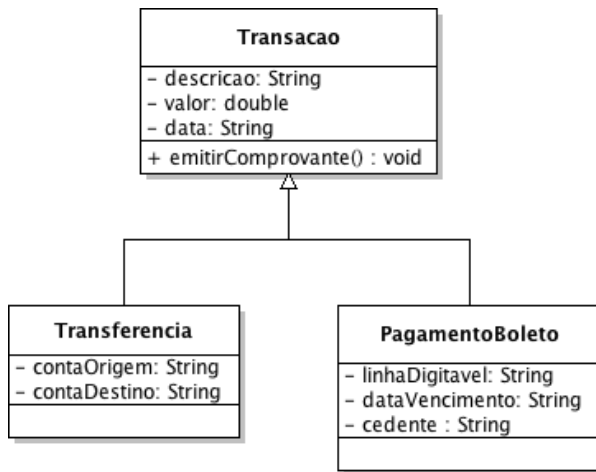
```

Conseguimos melhorar um pouco a estrutura de nossas classes, eliminando a bagunça que teríamos que tratar com tudo junto em uma única classe, porém, estamos repetindo bastante código.

Podemos deixar nossas classes muito mais reutilizáveis com orientação a objetos. Para isso, usaremos o conceito de herança. Herança adiciona a capacidade de uma classe filha estender/herdar atributos e métodos de uma classe mãe.

Na UML, o relacionamento de herança é simbolizado por uma linha sólida com uma seta não preenchida, que aponta da classe filha para a classe mãe.





A classe `Transacao` possuirá apenas atributos e métodos que representam uma transação genérica.

```

public class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }

    // getters e setters

}
  
```

As classes `Transferencia` e `PagamentoBoleto` herdam a classe `Transacao`. A herança é declarada através da palavra-chave `extends`.

```

public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    // getters e setters

}

public class PagamentoBoleto extends Transacao {
  
```

```

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    // getters e setters

}

```

Agora, instâncias de objetos do tipo `Transferencia` e `PagamentoBoleto` possuem os membros de `Transacao` também.

```

public class TesteTransacao {

    public static void main(String[] args) {
        PagamentoBoleto pagtoBoleto = new PagamentoBoleto();
        pagtoBoleto.setDescricao("Condomínio");
        pagtoBoleto.setValor(450);
        pagtoBoleto.setData("26/08/2013");
        pagtoBoleto.setLinhaDigitavel("1234 1234 1234");
        pagtoBoleto.setDataVencimento("27/08/2013");
        pagtoBoleto.setCedente("Edifício da Praça Redonda");
        pagtoBoleto.emitirComprovante();

        Transferencia transferencia = new Transferencia();
        transferencia.setDescricao("Aluguel");
        transferencia.setValor(1500);
        transferencia.setData("10/08/2013");
        transferencia.setContaOrigem("0001000123");
        transferencia.setContaDestino("0001000965");
        transferencia.emitirComprovante();
    }

}

```

Podemos dizer que `PagamentoBoleto` é **uma** `Transacao`, assim como `Transferencia` também é.

Também é comum dizer que `Transacao` é uma **superclasse** e `Transferencia` e `PagamentoBoleto` são **subclasses** de `Transacao`.

Java não permite herança múltipla, ou seja, uma classe pode herdar de apenas uma outra classe.

O problema agora é que o método `emitirComprovante` não tem acesso aos atributos e métodos das classes filhas, para imprimir na saída algumas informações específicas dessas subclasses. Por exemplo, a execução da classe acima exibe na saída comprovantes genéricos.

Comprovante da transação

=====

Descrição: Condomínio

Data: 26/08/2013

Valor: 450.0

Comprovante da transação

=====

Descrição: Aluguel

Data: 10/08/2013

Valor: 1500.0

Podemos usar o conceito de **sobrescrita** e declarar o método `emitirComprovante` nas classes filhas, substituindo totalmente o método da classe mãe. Por enquanto, vamos fazer isso apenas na classe `Transferencia`.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters
}
```

Agora a execução de nossa classe de teste exibe na saída o comprovante de transferência que implementamos no método sobrescrito na classe filha.

Comprovante da transação

=====

Descrição: Condomínio

Data: 26/08/2013

Valor: 450.0

Comprovante da transação

=====

Descrição: Aluguel

Data: 10/08/2013

Valor: 1500.0

Conta de origem: 0001000123

Conta de destino: 0001000965

Em nosso exemplo, os cabeçalhos dos comprovantes são idênticos para todos os tipos de transações, por isso, seria desnecessário escrever esse código novamente em todas as classes filhas. Mesmo substituindo um método da classe mãe, podemos chamá-lo se acharmos necessário, usando a palavra-chave `super`.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters
}
```

Pronto! Agora vamos fazer a mesma coisa com a classe `PagamentoBoleto`.

```
public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters
}
```

A chamada `super.emitirComprovante()` irá invocar o método da superclasse e depois imprimir na saída os detalhes específicos da transação, como uma transferência ou pagamento de boleto.

Comprovante da transação

=====

Descrição: Condomínio

Data: 26/08/2013

Valor: 450.0

Linha digitável: 1234 1234 1234

Data de vencimento: 27/08/2013

Cedente: Edifício da Praça Redonda

Comprovante da transação

=====

Descrição: Aluguel

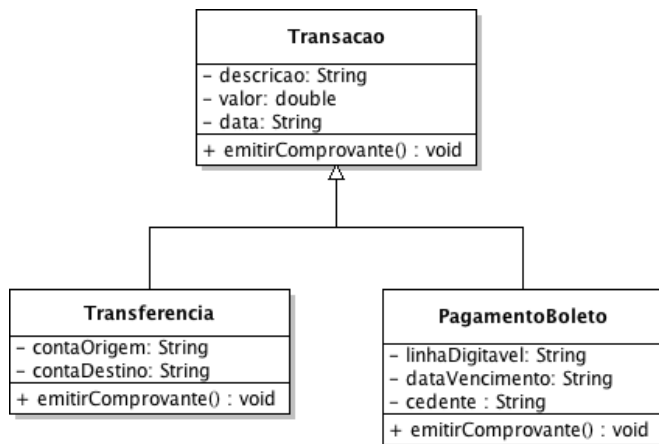
Data: 10/08/2013

Valor: 1500.0

Conta de origem: 0001000123

Conta de destino: 0001000965

Nosso diagrama de classes final ficou da seguinte forma:



Como você pôde ver, herança é um recurso poderoso da orientação a objetos, mas você deve usar com muito cuidado.

É comum ver programadores iniciantes em orientação a objetos usar herança apenas para reaproveitar código, e isso é muito errado.

Para evitar este erro, sempre que pensar em usar herança, faça uma pequena análise. Suponhamos que você queira que uma classe `Radio` herde `Televisor` para reaproveitar membros comuns. Pergunte-se a si mesmo: "*rádio é um televisor?*". Claro

que a resposta é **não**, rádio não é um tipo específico de televisor, por isso, Radio não deve estender Televisor.

## java.lang.Object

Todas as classes em Java, inclusive as que você mesmo criou, herdam de `java.lang.Object`.

```
// herda Object implicitamente
public class MinhaClasse {

}

// herda Object explicitamente
public class MinhaClasse extends Object {
}
```

Se uma classe herdar outra, apenas a classe pai que pode herdar `Object` diretamente.

```
// herda Object implicitamente
public class MinhaClasse {
}

public class OutraClasse extends MinhaClasse {
}
```

A classe `OutraClasse` não herda `Object` diretamente, mas ainda podemos dizer que `OutraClasse` é um `Object`, pois toda a hierarquia de classes é herdada por `OutraClasse`.

## 13.2. Modificador de acesso `protected`

O modificador de acesso `protected` diz que o membro só pode ser acessado pelo próprio pacote (igual o nível *default*) e por subclasses, mesmo estando em outros pacotes. Este modificador pode ser usado em atributos, métodos e construtores de uma classe.

```
package com.algaworks.comum;

public class Equipamento {

    protected boolean ligado;

    public boolean isLigado() {
        return this.ligado;
    }
}
```

```
    }  
}
```

Se uma classe de outro pacote tentar atribuir um valor para o atributo `ligado`, o código não compilará.

```
package com.algaworks.simulacao;  
  
import com.algaworks.comum.Equipamento;  
  
public class TesteProtected {  
  
    public static void main(String[] args) {  
        Equipamento equipamento = new Equipamento();  
  
        // não compila  
        equipamento.ligado = true;  
    }  
}
```

O código acima não compila! A classe `TesteProtected` está em um pacote diferente da classe `Equipamento`, e o atributo `ligado` não está visível.

Criaremos agora uma classe que estende `Equipamento`, em um pacote diferente, mas usa o atributo `ligado`.

```
package com.algaworks.audio;  
  
import com.algaworks.comum.Equipamento;  
  
public class Radio extends Equipamento {  
  
    public void ligar() {  
        // compila!  
        this.ligado = true;  
    }  
}
```

A classe `Radio` compila normalmente, pois um atributo protegido é visível para subclasses.

```
package com.algaworks.simulacao;  
  
import com.algaworks.audio.Radio;  
  
public class TesteProtected {  
  
    public static void main(String[] args) {
```

```

        Radio radio = new Radio();

        radio.ligar();
        System.out.println(radio.isLigado());
    }

}

```

Deixar atributos protegidos para serem acessados diretamente por outras classes pode influenciar você e outros programadores a alterarem valores de atributos sem passar por um método, "roubando" a responsabilidade da classe de gerenciar o seu estado. Isso não é uma boa prática, por isso, tome bastante cuidado.

### 13.3. Polimorfismo

Polimorfismo é a capacidade de um objeto tomar várias formas, que decorre diretamente do mecanismo de herança. Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.

Voltando ao código do tópico sobre herança, vamos instanciar um objeto do tipo *Transferencia*, mas atribuir a referência a uma variável do tipo *Transacao*.

```
Transacao transferencia = new Transferencia();
```

Quando fazemos isso, o objeto que instanciamos não é convertido para um novo tipo. Ele continua sendo uma transferência, mas **enxergamos** ele apenas como uma transação, por isso, não conseguimos invocar os métodos específicos de uma transferência.

```
Transacao transferencia = new Transferencia();
transferencia.setDescricao("Aluguel");
transferencia.setValor(1500);
transferencia.setData("10/08/2013");
```

```
// não compila
transferencia.setContaOrigem("0001000123");
transferencia.setContaDestino("0001000965");
```

```
transferencia.emitirComprovante();
```

Talvez você ache que não faz sentido um objeto de um tipo ser enxergado pelo supertipo, pois restringe os métodos que podem ser chamados. Esse é um pensamento comum em pessoas que estão iniciando com orientação a objetos, mas não se engane, o poder do polimorfismo é fantástico!



Vamos criar um construtor na classe Transferencia, para ficar mais fácil nosso exemplo.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public Transferencia(String descricao, double valor, String data,
        String contaOrigem, String contaDestino) {
        super(descricao, valor, data);

        this.contaOrigem = contaOrigem;
        this.contaDestino = contaDestino;
    }

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters
}
```

Veja uma novidade aí! Chamamos o construtor da classe mãe pela instrução `super(descricao, valor, data)`. Esse construtor ainda não existe, portanto, precisamos criá-lo também.

```
public class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }
}
```

```

    }

    // getters e setters

}

```

Voltando ao polimorfismo, vamos instanciar novamente uma *Transferencia*, passando os dados pelo construtor criado, e invocar o método *emitirComprovante*.

```

Transacao transferencia = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");
transferencia.emitirComprovante();

```

O código acima exibe na saída:

Comprovante da transação

=====

Descrição: Aluguel

Data: 10/08/2013

Valor: 1500.0

Conta de origem: 0001000123

Conta de destino: 0001000965

Veja que o método chamado foi da classe *Transferencia*.

Precisamos criar um construtor na classe *PagamentoBoleto* para continuar nossos exemplos.

```

public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public PagamentoBoleto(String descricao, double valor,
        String data, String linhaDigitavel, String dataVencimento,
        String cedente) {
        super(descricao, valor, data);

        this.linhaDigitavel = linhaDigitavel;
        this.dataVencimento = dataVencimento;
        this.cedente = cedente;
    }

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
    }
}

```

```

        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters
}

```

Agora, vamos criar uma classe emissora de comprovantes, que desconhece qualquer tipo específico de transação.

```

public class EmissorDeComprovantes {

    public void emitirComprovantes(Transacao... transacoes) {
        for (Transacao transacao : transacoes) {
            transacao.emitirComprovante();
        }
    }
}

```

Repare que o programador da classe EmissorDeComprovantes não precisa conhecer os tipos de transações que existem. Ele só precisa conhecer o que uma transação é capaz de fazer. Dessa forma, deixamos o código mais desacoplado e facilitamos a manutenção do software.

Instanciamos nosso emissor de comprovantes e invocamos o método emitirComprovantes.

```

Transferencia transferencia = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");

PagamentoBoleto pagtoBoleto = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");

EmissorDeComprovantes emissor = new EmissorDeComprovantes();
emissor.emitirComprovantes(transferencia, pagtoBoleto);

```

A JVM chama o método apropriado para cada objeto, de acordo com o tipo real dele, e não do tipo da variável.

A execução do último código exibe na saída:

```

Comprovante da transação
=====
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0

```

Conta de origem: 0001000123  
Conta de destino: 0001000965

Comprovante da transação

=====

Descrição: Condomínio  
Data: 26/08/2013  
Valor: 450.0  
Linha digitável: 1234 1234 1234  
Data de vencimento: 27/08/2013  
Cedente: Edifício da Praça Redonda

## 13.4. Casting de objetos e instanceof

Um objeto tem o tipo da classe usada para instanciá-lo.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,  
    "26/08/2013", "1234 1234 1234", "27/08/2013",  
    "Edifício da Praça Redonda");  
  
Object obj = pagtoBoleto;
```

No código acima, o objeto referenciado pelas variáveis `transacao` e `obj` é do tipo `PagamentoBoleto`.

A classe `PagamentoBoleto` é filha de `Transacao`, que é filha de `Object`. Podemos dizer que `PagamentoBoleto` é uma `Transacao` e também um `Object`. Dessa forma, sempre que você precisar de uma `Transacao` ou `Object`, poderá passar um objeto do tipo `PagamentoBoleto`.

Pense no mundo real: se o gerente de sua conta pedir para você falar qual transação você quer gerar uma segunda via do comprovante, fica subentendido que você pode dizer os dados de um pagamento de boleto ou de uma transferência, além das diversas outras transações. Se um pedinte na rua pedir para você qualquer coisa, por exemplo, "pode me dar qualquer coisa?", você poderia simplesmente entregar um pagamento de boleto para ele, uma caneta, seu carro, uma nota fiscal, uma moeda ou qualquer outra coisa, pois o que foi solicitado é um tipo muito genérico, um `Object` do mundo real.

Até agora, estudamos o caminho mais simples, onde temos um tipo específico sendo atribuído a um tipo mais genérico.

O reverso exige um pouco de cuidado! Uma Transacao pode ser um PagamentoBoleto, mas também pode não ser.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");
```

```
// não compila
PagamentoBoleto pagtoBoleto = transacao;
```

Sabemos que no código acima, instanciamos um PagamentoBoleto, mas o compilador não sabe. Por isso, não conseguimos compilar. Precisamos dizer ao compilador que queremos fazer um *casting* do objeto! Quando fazemo isso, nada é convertido e nada no objeto é alterado, apenas assumimos todos os riscos de tentar enxergar um objeto mais genérico como um tipo mais específico.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");
```

```
// compila!
PagamentoBoleto pagtoBoleto = (PagamentoBoleto) transacao;
System.out.println(pagtoBoleto.getLinhaDigitavel());
```

Conseguimos fazer o *casting* escrevendo (PagamentoBoleto) na frente do nome da variável.

Agora, vamos instanciar uma Transferencia, declarando a variável com o tipo Transacao, e depois tentar atribuir a uma variável do tipo PagamentoBoleto.

```
Transacao transacao = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");
```

```
// não compila
PagamentoBoleto pagtoBoleto = transacao;
```

Precisamos deixar explícito o *casting* que queremos fazer, ou o código não irá compilar.

```
Transacao transacao = new Transferencia("Aluguel", 1500, "10/08/2013",
    "0001000123", "0001000965");
```

```
// compila, mas experimente rodar (hehehe)
PagamentoBoleto pagtoBoleto = (PagamentoBoleto) transacao;
```

Fizemos o *casting* do objeto para o tipo PagamentoBoleto, e o código compila, pois é possível que o objeto referenciado pela variável transacao seja do tipo PagamentoBoleto, já que declaramos essa variável com o tipo Transacao. Nós

sabemos que, na verdade, é uma transferência, mas o compilador não sabe disso, e acredite, muitas vezes nem mesmo você saberá, por exemplo, quando você recebe objetos como parâmetro de um método, e não é você que passa esses parâmetros (mas outro programador).

Apesar do código compilar, quando executamos, recebemos uma mensagem de erro em tempo de execução.

```
Exception in thread "main" java.lang.ClassCastException:
    Transferencia cannot be cast to PagamentoBoleto
    at TesteCasting.main(TesteCasting.java:8)
```

## Operador de comparação instanceof

Você pode fazer uma comparação lógica para verificar se um objeto é de um tipo específico, usando o operador instanceof.

```
Object obj = new Transferencia("Aluguel", 1500, "10/08/2013",
    "0001000123", "0001000965");

if (obj instanceof PagamentoBoleto) {
    PagamentoBoleto pagtoBoleto = (PagamentoBoleto) obj;
    System.out.println(pagtoBoleto.getLinhaDigitavel());
} else if (obj instanceof Transacao) {
    Transacao transacao = (Transacao) obj;
    System.out.println(transacao.getDescricao());
}
```

Com essa verificação, você fica livre de erros em tempo de execução pelo motivo de *castings* inválidos.

## 13.5. Classes e métodos abstratos

No exemplo de transações de uma conta em um banco, que fizemos nas últimas seções, criamos a classe *Transacao* apenas para reaproveitar código e tomar vantagem do polimorfismo. Não faz sentido algum instanciar uma *Transacao*.

```
Transacao transacao = Transacao("Convênio médico", 400, "05/08/2013");
```

A transação que instanciamos é muito genérica! Não é um pagamento de boleto, uma transferência ou qualquer outra coisa. É apenas uma transação.

Se analisarmos o mundo real, realmente, não existe o conceito concreto do que é uma transação em um banco. Sempre existe algo mais específico para definir melhor, neste caso.

Podemos pensar em outro exemplo do mundo real: uma fabricante de carros fabrica apenas veículos, ou carros? Se alguém disser que quer oferecer um veículo para você, com certeza você deve imaginar que pode ser uma moto, um carro, um avião, mas nunca apenas um veículo, sem um tipo específico. Sabe porque? Veículo é muito abstrato! Não existem objetos simplesmente do tipo Veículo no mundo real. Usamos a palavra "veículo" para conceituar um meio de transporte.

Continuando o raciocínio sobre nosso exemplo, seria interessante dizer que Transacao é uma classe abstrata, portanto, não pode ser instanciada. Para isso, usamos a palavra-chave `abstract` na declaração da classe.

```
public abstract class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("=====");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }

    // getters e setters

}
```

Agora, não conseguimos instanciar uma Transacao.

```
// não compila
Transacao transacao = Transacao("Convênio médico", 400, "05/08/2013");
```

Você quer saber qual é a utilidade da classe Transacao, já que não podemos instanciá-la? Polimorfismo, principalmente, e reaproveitamento de código pelas subclasses.

## Métodos abstratos

Suponha que exista a necessidade de emitir comprovantes com conteúdos totalmente diferentes. Por exemplo, um comprovante de transferência pode incluir um cabeçalho com os números das contas de origem e destino, antes mesmo de imprimir o valor, etc.

Neste caso, podemos remover o método `emitirComprovante` da classe `Transacao`, pois não conseguiremos reaproveitá-lo.

```
public abstract class Transacao {  
  
    private String descricao;  
    private double valor;  
    private String data;  
  
    public Transacao(String descricao, double valor, String data) {  
        this.descricao = descricao;  
        this.valor = valor;  
        this.data = data;  
    }  
  
    // getters e setters  
  
}
```

Agora, alteramos a implementação dos métodos de emissão de comprovante nas classes `Transferencia` e `PagamentoBoleto`.

```
public class PagamentoBoleto extends Transacao {  
  
    private String linhaDigitavel;  
    private String dataVencimento;  
    private String cedente;  
  
    public PagamentoBoleto(String descricao, double valor,  
        String data, String linhaDigitavel,  
        String dataVencimento, String cedente) {  
        super(descricao, valor, data);  
  
        this.linhaDigitavel = linhaDigitavel;  
        this.dataVencimento = dataVencimento;  
        this.cedente = cedente;  
    }  
  
    public void emitirComprovante() {  
        System.out.println("Comprovante da pagamento de boleto");  
        System.out.println("=====");  
        System.out.println("Linha digitável: "  
            + this.getLinhaDigitavel());  
    }  
}
```



```

        System.out.println("-----");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Data do pagamento: " + this.getData());
        System.out.println();
    }

    // getters e setters
}

public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public Transferencia(String descricao, double valor, String data,
        String contaOrigem, String contaDestino) {
        super(descricao, valor, data);

        this.contaOrigem = contaOrigem;
        this.contaDestino = contaDestino;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante de transferência");
        System.out.println("=====");
        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println("-----");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Data da operação: " + this.getData());
        System.out.println();
    }

    // getters e setters
}

```

Você se lembra da classe EmissorDeComprovantes? Será que temos algum problema nela, depois dessas alterações?

```

public class EmissorDeComprovantes {

    public void emitirComprovantes(Transacao... transacoes) {
        for (Transacao transacao : transacoes) {

            // não compila

```

```

        transacao.emitirComprovante();
    }
}

```

Puxa! A classe EmissorDeComprovantes não está mais compilando, pois ela utiliza polimorfismo, e tenta chamar o método `emitirComprovante` em um tipo `Transacao`, mas nós removemos o método.

Para resolver isso, poderíamos criar um método `emitirComprovante` na classe `Transacao`, que não implementa nada, mas isso não é uma boa ideia, já que poderíamos esquecer de sobrescrever o método quando criarmos novas subclasses.

Felizmente, podemos resolver isso criando um método abstrato!

Em Java, uma classe abstrata pode ter métodos abstratos. Métodos abstratos devem ser, obrigatoriamente, implementados por uma subclasse. Para dizer que um método é abstrato, basta declararmos ele usando a palavra-chave `abstract`.

```

public abstract class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public abstract void emitirComprovante();

    // getters e setters
}

```

Repare que não incluímos corpo no método `emitirComprovante`. Métodos abstratos não possuem implementação, por isso, finalizamos a declaração dele com um ponto e vírgula.

Agora a classe `EmissorDeComprovantes` continua compilando, pois o compilador tem certeza que o método `emitirComprovante` será implementado por classes concretas que herdarem `Transacao`.

## 13.6. Interfaces

Existem diversas situações na vida real que precisamos definir padrões de como as coisas devem funcionar, e os fabricantes assinam ou concordam com um contrato que diz como os componentes se interagem. Cada fabricante é capaz de fornecer seus componentes/peças sem conhecer como as outras partes funcionam.

Para exemplificar, vamos pensar em como os carros funcionam, sem entrar em muitos detalhes técnicos, pois este autor não conhece a fundo sobre esse assunto, e acredito que você também não (a não ser que seja um mecânico ou apaixonado pela mecânica dos carros).

Para termos um carro funcionando, primeiramente, as peças devem ser fabricadas. Somente depois da fabricação de todas as peças que o carro é montado.

Durante a fabricação das peças, os diversos fabricantes conhecem as especificações das outras partes que irão interagir com as peças de sua responsabilidade, em termos do que elas devem ter, e não como elas funcionam exatamente.

Por exemplo, o fabricante de pneus deve conhecer o aro da roda e também que ela deve girar, mas não faz a mínima ideia de como funciona a combustão do motor ou se o carro tem freios ABS ou não.

O fabricante do painel do carro sabe que em determinado lugar, deve existir o espaço para o painel do computador de bordo, mas não tem mais nenhum detalhe a mais sobre isso. O computador de bordo pode ser fabricado por dezenas de outros fabricantes, inclusive os paralelos, e funcionar normalmente no carro, desde que siga o padrão.

Essas especificações ou padrões, nós chamamos em Java de interfaces. Uma interface especifica o que alguma coisa deve ter, mas não define como deve fazer para alcançar o que é necessário.

Pense nos diversos componentes do mundo real que seguem interfaces: TVs e seus componentes, computadores e as placas e acessórios que se encaixam perfeitamente, independente do fabricante, funcionários em sua empresa, que possuem cargos bem definidos e interagem com outros departamentos e pessoas, etc. O mundo todo é baseado em interfaces!

Interfaces é o recurso da orientação a objetos que potencializa o polimorfismo em nível máximo. É fantástico o que podemos fazer com a combinação desses conceitos!

Agora vamos implementar algo em Java usando o conceito de interfaces. Precisamos programar um checkout (fechamento de compra), que aceita pagamentos por cartão de crédito de uma operadora qualquer e imprime os dados da compra em uma impressora qualquer. Veja bem a palavra "qualquer". Neste momento, não importa muito qual é a operadora de cartão e qual é a impressora.

Vamos começar definindo o contrato de uma impressora.

```
package com.algaworks.impressao;

public interface Impressora {

    void imprimir(Imprimivel imprimivel);

}
```

Declaramos uma interface usando a palavra-chave interface.

Uma interface possui métodos públicos abstratos (sem implementação). Quando não declaramos que o método é público e abstrato, ele recebe implicitamente `public abstract`.

Na interface `Impressora`, dizemos que qualquer classe que tenha interesse em se tornar uma impressora, deve implementar o método `imprimir`. Note que não especificamos como deve ser feita essa impressão. Pode ser jato de tinta, laser, matricial, etc.

O método `imprimir` recebe um objeto do tipo `Imprimivel`.

```
package com.algaworks.impressao;

public interface Imprimivel {

    String getCabecalhoPagina();
    String getCorpoPagina();

}
```

Um imprimível é algo capaz de ser impresso, e para isso, definimos como requisitos mínimos que ele deve ter os dois métodos declarados, um que retorna o cabeçalho da página e outro que retorna o corpo da página.

Até agora, não temos nenhuma impressora e nenhum imprimível. Temos apenas contratos!

Deixaremos para implementar classes concretas mais pra frente, para estimular a programação baseada em interfaces, sem conhecer o que está por trás da implementação, que é uma grande dificuldade para quem está iniciando em programação orientada a objetos.

Agora, criaremos a interface *Operadora*, que define o que uma operadora de cartão de crédito deve ter.

```
package com.algaworks.pagamento;

public interface Operadora {

    boolean autorizar(Autorizavel autorizavel, Cartao cartao);

}
```

Veja que uma operadora deve ter o método *autorizar*, que recebe um objeto do tipo *Autorizavel* e um *Cartao*.

```
package com.algaworks.pagamento;

public interface Autorizavel {

    double getValorTotal();

}

package com.algaworks.pagamento;

public class Cartao {

    private String nomeTitular;
    private String numeroCartao;

    public String getNomeTitular() {
        return nomeTitular;
    }

    public void setNomeTitular(String nomeTitular) {
        this.nomeTitular = nomeTitular;
    }

    public String getNumeroCartao() {
        return numeroCartao;
    }

    public void setNumeroCartao(String numeroCartao) {
        this.numeroCartao = numeroCartao;
    }

}
```

Autorizavel é uma interface, que define o que uma classe deve implementar para ser autorizável, ou seja, para ser processada por alguma operadora de cartão de crédito.

A classe Cartao possui os dados básicos do cartão de crédito a ser autorizado/cobrado.

Agora chegou a hora de criar a primeira classe que implementa Autorizavel. Programaremos a classe Compra, que representa a compra que está sendo feita por um cliente. Uma compra é autorizável, ou seja, pode ser cobrada por cartão de crédito.

```
package com.algaworks.caixa;

import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel {

    private String nomeCliente;
    private double valorTotal;
    private String produto;

    public String getNomeCliente() {
        return nomeCliente;
    }

    public void setNomeCliente(String nomeCliente) {
        this.nomeCliente = nomeCliente;
    }

    public double getValorTotal() {
        return valorTotal;
    }

    public void setValorTotal(double valorTotal) {
        this.valorTotal = valorTotal;
    }

    public String getProduto() {
        return produto;
    }

    public void setProduto(String produto) {
        this.produto = produto;
    }

}
```

Uma classe implementa uma interface a partir da palavra-chave implements. A classe é obrigada a implementar todos os métodos declarados na interface. Neste caso, implementamos o método getValorTotal, que é o único método exigido pela interface Autorizavel.

Classes podem implementar diversas interfaces. Incluiremos a interface `Imprimivel` na declaração da classe `Compra`.

```
package com.algaworks.caixa;

import com.algaworks.impressao.Imprimivel;
import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel, Imprimivel {

    // atributos e métodos

}
```

Quando implementamos a interface `Imprimivel`, nosso código deixa de compilar, pois somos obrigados a implementar os métodos definidos pela nova interface que adicionamos.

```
package com.algaworks.caixa;

import com.algaworks.impressao.Imprimivel;
import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel, Imprimivel {

    // atributos e métodos

    public String getCorpoPagina() {
        return this.getProduto() + " = " + this.getValorTotal();
    }

    public String getCabecalhoPagina() {
        return this.getNomeCliente();
    }

}
```

A próxima classe será de nome `Checkout`. Esta classe receberá uma `Operadora` e uma `Impressora` no construtor, e terá um método `fecharCompra`, que receberá uma `Compra` e um `Cartao` para fechar a compra (autorizar e imprimir o cupom).

```
package com.algaworks.caixa;

import com.algaworks.impressao.Impressora;
import com.algaworks.pagamento.Cartao;
import com.algaworks.pagamento.Operadora;

public class Checkout {

    private Operadora operadora;
    private Impressora impressora;
```

```

public Checkout(Operadora operadora, Impressora impressora) {
    this.operadora = operadora;
    this.impressora = impressora;
}

public void fecharCompra(Compra compra, Cartao cartao) {
    boolean autorizado = this.operadora.autorizar(compra, cartao);

    if (autorizado) {
        this.impressora.imprimir(compra);
    } else {
        System.out.println("Pagamento negado!");
    }
}
}

```

Até aqui, este poderia ser o seu trabalho em um projeto em Java. As implementações de operadoras de cartões e impressoras, poderiam ser terceirizadas ou de responsabilidade de outro programador de sua equipe.

Ainda não conseguimos executar nosso projeto, pois precisamos de pelo menos uma classe que implementa Impressora e outra que implementa Operadora.

```

public class TesteCheckout {

    public static void main(String[] args) {
        // precisamos de implementações de Operadora e Impressora
        Operadora operadora = ...
        Impressora impressora = ...

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }
}

```

Criaremos uma classe ImpressoraEpson, que implementa a interface Impressora.

```

package com.algaworks.impressao;

public class ImpressoraEpson implements Impressora {

```



```

public void imprimir(Imprimivel imprimivel) {
    System.out.println("* * * * *");
    System.out.println(imprimivel.getCabecalhoPagina());
    System.out.println("* * * * *");
    System.out.println(imprimivel.getCorpoPagina());
    System.out.println("- - - - -");
    System.out.println("==          EPSON          ==");
    System.out.println("- - - - -");
}
}

```

E agora a classe Cielo, que implementa a interface Operadora e fornece a integração com a processadora de cartões e crédito Cielo.

```

package com.algaworks.pagamento;

public class Cielo implements Operadora {

    public boolean autorizar(Autorizavel autorizavel, Cartao cartao) {
        return cartao.getNumeroCartao().startsWith("123");
    }

}

```

Para efeito de testes, a implementação da Cielo autorizará apenas cartões com números que comecem com "123".

Podemos agora instanciar um Checkout, passando como parâmetro do construtor os objetos das classes que implementam Impressora e Operadora.

```

public class TesteCheckout {

    public static void main(String[] args) {
        Operadora operadora = new Cielo();
        Impressora impressora = new ImpressoraEpson();

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }

}

```

Tente executar a classe `TesteCheckout`, alterando o número do cartão.

Até agora, parece que o uso de interfaces não trouxe nenhum benefício.

Imagine que sua empresa tenha necessidade de instalar uma nova impressora, e também fechou um contrato com a Redecard, para ter uma alternativa no processamento de pagamentos com cartões de crédito. Você pode ficar responsável pela implementação da nova impressora e passar para outro programador a responsabilidade da integração com a Redecard. Basta usar as interfaces!

```
package com.algaworks.impressao;

public class ImpressoraXingling implements Impressora {

    public void imprimir(Imprimivel imprimivel) {
        System.out.println(imprimivel.getCabecalhoPagina());
        System.out.println("-----");
        System.out.println(imprimivel.getCorpoPagina());
        System.out.println("=====");
        System.out.println("**Xingling Printer**");
    }

}
```

A classe `ImpressoraXingling` exibe na saída os mesmos dados do cabeçalho e corpo da página, mas em um formato diferente.

```
package com.algaworks.pagamento;

public class Redecard implements Operadora {

    public boolean autorizar(Autorizavel autorizavel, Cartao cartao) {
        return cartao.getNumeroCartao().startsWith("456")
            && autorizavel.getValorTotal() < 200;
    }

}
```

A classe `Redecard` autoriza apenas cartões que os números iniciem com "456" e o valor total seja menor que R\$200.

Para usar as novas classes, basta substituir a instanciação em `TesteCheckout`.

```
public class TesteCheckout {

    public static void main(String[] args) {
        Operadora operadora = new Redecard();
        Impressora impressora = new ImpressoraXingling();
    }

}
```

```

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }
}

```

Faça diversos testes, alternando as implementações de impressoras e operadoras e mudando o valor total da compra e número do cartão.

O uso de interfaces deixou nosso código muito mais extensível e desacoplado. A classe Checkout não conhece as implementações de impressoras e operadoras, mas apenas os contratos (as interfaces).

Da mesma forma, as implementações de impressoras não conhecem o que está sendo impresso, e as classes que implementam operadoras não conhecem o que está sendo autorizado. Tudo é baseado em interfaces.

## 13.7. Tratando e lançando exceções

Exceções são situações excepcionais e geralmente indesejáveis que podem ocorrer durante a execução de um programa. Exceções podem ser tratadas incluindo código adequado no programa.

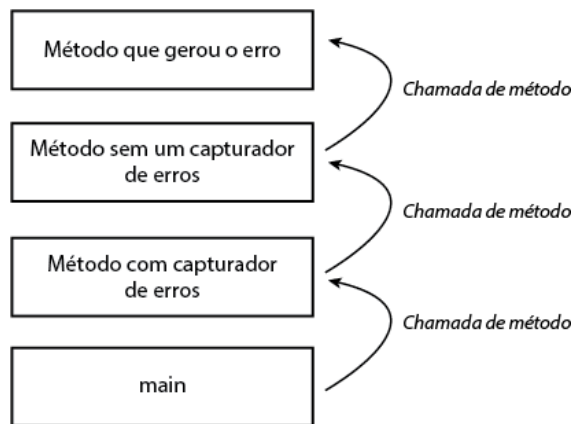
Exemplos típicos de exceções são:

- Índice de uma lista (array) fora do intervalo permitido.
- Problemas em operações aritméticas, tais como *overflows* e divisões por zero.
- Argumentos inválidos numa chamada a um método.
- Uso de uma referência que não aponta para nenhum objeto.
- Falta de memória (relativamente improvável em Java, graças ao coletor de lixo).

O modelo de tratamento de exceções em Java permite tratar uma exceção num escopo (bloco de código) diferente daquele que gerou a exceção. Isso permite uma melhor organização do código.

Quando ocorre um erro em um método, o método cria um objeto de exceção que contém informações sobre o erro, incluindo seu tipo e o estado do programa quando o mesmo foi gerado. O ato de criar objetos de exceção e colocá-los no sistema é chamado de "lançamento de exceção".

Quando um método lança uma exceção, o sistema tenta encontrar um código na pilha de execução capaz de capturá-la. A pilha de execução, também conhecida como *call stack*, é uma lista de métodos ordenados, que foram chamados até chegar ao método que lançou a exceção.



O sistema procura na pilha de execução por um método que contém um bloco de código que pode capturar a exceção. Esse bloco de código é chamado de "capturador de exceção" (*exception handler*). A pesquisa começa no método que ocorreu o erro e continua na pilha, na ordem inversa em que o método foi chamado. Quando um capturador apropriado é encontrado, o sistema passa a exceção para ele. O capturador de exceção é considerado apropriado se o tipo do objeto da exceção lançada for compatível com o tipo que o capturador pode tratar.

Se o sistema não encontrar um capturador em nenhum método, a execução do programa é encerrada.

## Captura de exceções

Para capturar uma exceção, é necessário montar uma estrutura de código.

O código que pode lançar a exceção deve ser inserido em um bloco precedido da palavra-chave `try`. O processador então tentará executar o bloco, até que eventualmente uma exceção seja lançada, seja por um comando contido dentro do bloco, seja por um método chamado dentro do bloco.

O bloco `try` pode ser seguido de um bloco que será executado, caso houver de fato o lançamento de uma exceção do tipo especificado. Este bloco deve ser anunciado pela palavra-chave `catch`, seguida (entre parênteses) do tipo de exceção em questão.

Se vários tipos de exceções puderem ser lançadas no bloco `try`, deve-se fornecer um bloco `catch` para cada tipo de exceção, podendo especificar tipos mais abrangentes da hierarquia das classes de exceções, ou ainda, a partir do Java 7, incluir vários nomes de exceções seguidas por `|` (pipe).

Se uma exceção for lançada no bloco `try`, o bloco é encerrado e a execução salta para o bloco `catch` apropriado. Os blocos `catch` são chamados manipuladores de exceções.

Ainda é possível acrescentar, após os blocos `catch`, um bloco precedido da palavra-chave `finally`, que será executado em todos os casos, após a execução completa do bloco `try` ou após a execução de um bloco `catch`, conforme o caso.

Resumindo, a estrutura é:

```
try {  
    // aqui vai o código que pode gerar exceções dos tipos  
  
} catch (TipoUmException ex1) {  
    // aqui vai o código para lidar com uma exceção do tipo  
    // TipoUmException  
  
} catch (TipoDoisException ex2) {  
    // aqui vai o código para lidar com uma exceção do tipo  
    // TipoDoisException  
  
} catch (TipoTresException | TipoQuatroException ex3) {  
    // aqui vai o código para lidar com exceções do tipo  
    // TipoTresException e TipoQuatroException (apenas Java 7)  
  
} finally {  
    // aqui vai o código que deve ser executado em qualquer caso  
  
}
```

Quando ocorre uma exceção, os blocos `catch` são examinados sucessivamente até que o argumento corresponda ao tipo da exceção. Note que, no exemplo acima, se `TipoDoisException` for uma subclasse de `TipoUmException`, o segundo bloco `catch` nunca será alcançado. Neste caso, deve-se inverter a ordem dos blocos `catch`.

Um bloco `catch` pode também lançar novas exceções e relançar a exceção que ele manipulou. Neste caso, todo o conjunto de blocos acima deve estar inserido num bloco `try` mais externo, e exceção relançada deve ser manipulada por um bloco `catch` seguindo este bloco `try` externo.

Antes de testar a captura de exceção, crie uma classe que executa o seguinte código no método `main`:

```
double x = 4 / 0;
```

Você já deve saber que essa divisão retornará um erro, pois estamos tentando dividir por zero:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Se quisermos tratar esse erro para tentar corrigi-lo ou informar ao usuário com uma mensagem não técnica, utilizamos o bloco `try/catch`:

```
try {  
    double x = 4 / 0;  
} catch (ArithmeticException ae) {  
    System.out.println("Erro na divisão");  
}
```

O código acima captura a exceção causada por divisão por zero e exibe a mensagem "Erro na divisão".

Opcionalmente, podemos exibir a pilha de execução da exceção.

```
try {  
    double x = 4 / 0;  
} catch (ArithmeticException ae) {  
    ae.printStackTrace();  
}
```

Você não é obrigado a tratar a exceção `ArithmeticException`, pois ela é uma *unchecked exception*, mais especificamente, uma *runtime exception*.

Exceções não checadas não obrigam a captura, pois, normalmente, é um caso impossível de recuperar, e continuar executando apenas pioraria a situação. Portanto,

se o programa não incluir um bloco `catch` para tratar esta exceção, o processo em que ela acontece é interrompido, mostrando uma mensagem de erro.

Algumas exceções são conhecidas como *errors*, também do tipo *unchecked exception*, ou seja, não é necessário captura. Os erros são utilizados pela JVM para indicar uma deficiência de recursos, falhas ou outras condições que tornam impossível que a execução continue, por exemplo `OutOfMemoryError`, quando a memória da JVM é esgotada.

Existem também exceções do tipo *checked exception*, usadas para condições em que se espera que a classe que chamou o método possa se recuperar. O lançamento da exceção é um indício ao usuário da API, que aquela determinada condição é um dos retornos possíveis da chamada do método, por isso, é obrigatório que seja inserido um código para tratar a exceção ou propagá-la. Veremos um exemplo de exceção checada mais a frente.

## 13.8. Lançamento de exceções

Naturalmente, exceções são objetos em Java. A classe `Exception` é a superclasse de todas as exceções. Existem vários tipos de exceções já definidas nas bibliotecas. O programador pode também construir as suas próprias exceções.

Muitas classes de bibliotecas possuem métodos que podem lançar exceções. Estas exceções podem ser tratadas (capturadas) por código escrito pelo programador. Além disto, métodos escritos pelo programador também podem lançar exceções, tanto de tipos já definidos em bibliotecas, como de novos tipos construídos pelo programador.

Uma exceção é lançada usando-se a palavra-chave `throw`, seguida pela referência da exceção.

```
Exception zebra = new Exception("Deu zebra!");  
throw zebra;
```

Para lançar a exceção dentro de um método, você deve incluir na declaração do método a palavra-chave `throws`, seguida do tipo da exceção.

```
public class TesteLancamentoExcecao1 {  
  
    public static void main(String[] args) {  
        try {  
            metodoQuePodeLancarExcecao();  
        } catch (Exception e) {
```

```

        e.printStackTrace();
    }
}

private static void metodoQuePodeLancarExcecao()
    throws Exception {
    Exception zebra = new Exception("Deu Zebra!");
    throw zebra;
}
}

```

Uma vez que a exceção foi lançada, a execução do método é interrompida e o controle volta para quem chamou este método, que pode relançar a exceção para que ela seja capturada em outro nível da pilha de chamadas, ou tratá-la.

```

public class TesteLancamentoExcecao2 {

    private static void metodoUm() throws Exception {
        Exception zebra = new Exception("Deu Zebra!");
        throw zebra;
    }

    private static void metodoDois() throws Exception {
        metodoUm();
    }

    private static void metodoTres() throws Exception {
        try {
            metodoDois();
        } catch (Exception e) {
            System.out.println("Erro no método três.");

            // relança a exceção
            throw e;
        }
    }

    public static void main(String[] args) {
        try {
            metodoTres();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

As exceções não checadas não precisam ser listadas explicitamente após a palavra-chave `throws`. Correspondentemente, não é obrigatório incluir código para manipular tais exceções.



```

public class TesteLancamentoExcecao3 {

    private static void metodoQuePodeLancarExcecao() {
        // RuntimeException é uma exceção não checada
        RuntimeException zebra = new RuntimeException("Deu Zebra!");
        throw zebra;
    }

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

}

```

## Exceções customizadas

Você pode estender a classe `Exception` ou uma das suas subclasses para construir as suas próprias exceções.

```

public class EstoqueInsuficienteException extends Exception {

    public EstoqueInsuficienteException(String msg) {
        super(msg);
    }

}

```

Para lançar uma exceção customizada, basta instanciar a classe da exceção e disparar com `throw`.

```

public class Produto {

    private int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
    }

    public void baixarEstoque(int quantidade)
        throws EstoqueInsuficienteException {
        if (quantidade > this.estoque) {
            throw new EstoqueInsuficienteException("Estoque "
                + "insuficiente. " + "Estoque atual: "
                + this.estoque);
        }

        this.estoque -= quantidade;
    }

}

```

A chamada do método `baixarEstoque` deve tratar a possível exceção, que pode ser lançada quando o estoque for insuficiente.

```
public class TesteEstoque {  
  
    public static void main(String[] args) {  
        Produto produto = new Produto(10);  
        try {  
            produto.baixarEstoque(8);  
            produto.baixarEstoque(4);  
            produto.baixarEstoque(20);  
        } catch (EstoqueInsuficienteException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

A segunda invocação do método `baixarEstoque` lança uma exceção. Veja a saída do programa.

```
EstoqueInsuficienteException: Estoque insuficiente. Estoque atual: 2  
    at Produto.baixarEstoque(Produto.java:13)  
    at TesteEstoque.main(TesteEstoque.java:9)
```

# Mais sobre a API Java

## 14.1. Classe java.lang.Math

No dia a dia como desenvolvedores, uma hora ou outra precisamos utilizar alguma operação matemática básica, como raiz quadrada, exponencial, logarítmo e até funções de trigonometria. *Tá bom*, não é todo dia que usamos trigonometria.

A boa notícia é que a API do Java tem uma classe que realiza muitas dessas operações. É a classe `Math` do pacote `java.lang`.

Nela temos vários métodos estáticos para os mais variados tipos de operações matemáticas.

Vamos a um primeiro exemplo simples, calcular a raiz quadrada.

### Método `sqrt`

A assinatura do método da classe `Math` para realizar o cálculo da raiz quadrada é:

```
public static double sqrt(double a)
```

Como podemos observar, ele recebe um `double` como argumento e devolve também um `double` com o resultado da operação.

```
double a = 9;
double r = Math.sqrt(a);
System.out.println("Raiz quadrada de " + a + " = " + r);
```

Veja o resultado da execução do código acima:

Raiz quadrada de  $9.0 = 3.0$

## Método abs

Existem muitos outros métodos na classe `Math`, alguns com várias sobrecargas para diferentes tipos de dados, como é o caso do método `abs`, que retorna o valor absoluto (o valor do número sem sinal). Ele é sobrecarregado para aceitar `double`, `float`, `int` e `long`.

```
double a = 9.0;
double b = -15.0;

double ra = Math.abs(a);
double rb = Math.abs(b);

int x = 200;
int y = -8;

int rx = Math.abs(x);
int ry = Math.abs(y);

System.out.println("O valor absoluto de " + a + " = " + ra);
System.out.println("O valor absoluto de " + b + " = " + rb);
System.out.println("O valor absoluto de " + x + " = " + rx);
System.out.println("O valor absoluto de " + y + " = " + ry);
```

Saída:

```
O valor absoluto de 9.0 = 9.0
O valor absoluto de -15.0 = 15.0
O valor absoluto de 200 = 200
O valor absoluto de -8 = 8
```

## Método ceil

Veja assinatura do método:

```
public static double ceil(double a)
```

Este método irá arredondar o valor recebido como argumento para o próximo maior valor inteiro. Parece simples, certo? Mas vamos fazer um teste para não deixar nenhuma dúvida.

```
double a = 8.05;
double b = -8.75;
```

```
double ra = Math.ceil(a);  
double rb = Math.ceil(b);  
  
System.out.println("Próximo maior inteiro para: " + a + " = " + ra);  
System.out.println("Próximo maior inteiro para: " + b + " = " + rb);
```

Com números positivos é simples deduzir o próximo maior inteiro. E com números negativos? Qual o próximo maior inteiro para -8.75? A resposta certa é -8.0. Veja abaixo a saída da execução do código acima:

```
Próximo maior inteiro para: 8.05 = 9.0  
Próximo maior inteiro para: -8.75 = -8.0
```

## Método floor

Veja a assinatura do método:

```
public static double floor(double a)
```

A função deste método é o inverso do método ceil. Ele irá arredondar para o próximo menor valor inteiro. E também devemos ter cuidado ao pensar no resultado para valores negativos.

```
double a = 8.05;  
double b = -8.75;  
  
double ra = Math.floor(a);  
double rb = Math.floor(b);  
  
System.out.println("Próximo menor inteiro para: " + a + " = " + ra);  
System.out.println("Próximo menor inteiro para: " + b + " = " + rb);
```

Veja a saída abaixo:

```
Próximo menor inteiro para: 8.05 = 8.0  
Próximo menor inteiro para: -8.75 = -9.0
```

## Método max

Este método retorna o maior entre dois valores passados como argumento. Isso é bem útil para melhorarmos a legibilidade do nosso código.

```
double despesasFixas = 400.00;  
double despesasVariaveis = 300.0;
```

```
double maiorDespesaDoMes = Math.max(despesasFixas, despesasVariaveis);
System.out.println("Maior despesa do mês: R$ " + maiorDespesaDoMes);
```

Veja a saída abaixo:

Maior despesa do mês: R\$ 400.0

Repare que este código fica mais legível do que utilizando um if ou até um operador ternário.

Usando if:

```
double maiorDespesaDoMes = 0.0;
if (despesasFixas >= despesasVariaveis) {
    maiorDespesaDoMes = despesasFixas;
} else {
    maiorDespesaDoMes = despesasVariaveis;
}
```

Usando operador ternário:

```
double maiorDespesaDoMes = despesasFixas >= despesasVariaveis
    ? despesasFixas : despesasVariaveis;
```

Este método é sobrecarregado quatro vezes para receber float, double, int e long. Veja as assinaturas:

```
public static float max(float a, float b)
public static double max(double a, double b)
public static int max(int a, int b)
```

## Método min

O método min faz o inverso do método max. Também é sobrecarregado quatro vezes para float, double, int e long.

```
public static float min(float a, float b)
public static double min(double a, double b)
public static int min(int a, int b)
```

O exemplo é basicamente o mesmo do max.

```
double despesasFixas = 400.00;
double despesasVariaveis = 300.0;

double menorDespesaDoMes = Math.min(despesasFixas, despesasVariaveis);
System.out.println("Menor despesa do mês: R$ " + menorDespesaDoMes);
```

Veja a saída abaixo:

Menor despesa do mês: R\$ 300.0

## Método round

Este é um método bastante útil para fazermos arredondamento de float ou double.

Para os valores onde a parte decimal for menor que 0.5, o arredondamento é para baixo, caso contrário é para cima.

```
float a = 3.4f;
int ra = Math.round(a);
System.out.println("ra=" + ra);

float b = 3.7f;
int rb = Math.round(b);
System.out.println("rb=" + rb);

float c = 3.5f;
int rc = Math.round(c);
System.out.println("rc=" + rc);
```

A saída será:

```
ra=3
rb=4
rc=4
```

Para este método, temos duas assinaturas.

```
public static int round(float a)
public static long round(double a)
```

Estes são alguns dos vários métodos estáticos da classe `java.lang.Math`. Consulte a documentação para conhecer outros em: <http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>.

## 14.2. Static import

Na seção anterior, não precisamos importar a classe `Math`, pois ela é do pacote `java.lang`, mas podemos usar um recurso especial chamado de `import static`, que facilita e melhora a leitura de nosso código.

Logo abaixo da declaração do pacote da classe, podemos adicionar `import static` e importar qualquer membro estático de uma classe. Por exemplo, vamos importar o método estático `sqrt` da classe `Math`.

```
import static java.lang.Math.sqrt;
```

Agora, para utilizarmos o método `sqrt`, não precisamos mais escrever o nome da classe `Math`.

```
double r = sqrt(a);
```

Se quisermos importar método por método, basta adicionarmos mais uma linha de `import static`, por exemplo, vamos importar também o `abs`.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.abs;
```

Se estivermos utilizando vários métodos estáticos e quisermos importar todos ao mesmo tempo, podemos utilizar o *wildcard* `*`.

```
import static java.lang.Math.*;
```

## 14.3. Classes `String`, `StringBuffer` e `StringBuilder`

### Classe `String`

Já trabalhamos com `String` em capítulos anteriores, mas agora é hora de vermos mais detalhes dessa poderosa classe do Java.

Por `String` ser uma classe, ela vem com vários métodos extremamente úteis, que você com certeza utilizará no seu dia a dia.

A classe `String` é **imutável**. Isso quer dizer que, depois de instanciada, não conseguimos alterar seu valor.

Vamos usar o método `toUpperCase` da classe `String` para converter todos os caracteres para caixa alta (maiúscula). O método `toLowerCase` faz o contrário.

```
String nome = "Pedro";
System.out.println(nome.toUpperCase());
System.out.println(nome);
```

Veja a saída abaixo:



PEDRO  
Pedro

Parece confuso? Isso acontece porque o método `toUpperCase` retorna uma nova instância de `String`, e não altera o valor da variável `nome`. Podemos substituir a referência da variável `nome`.

```
String nome = "Pedro";  
System.out.println(nome);  
nome = nome.toUpperCase();  
System.out.println(nome);
```

Um ponto de atenção que devemos ter é na comparação de objetos do tipo `String`. Vamos ver a saída do código abaixo e entender o resultado.

```
String s1 = "Pedro";  
String s2 = "Pedro";  
  
System.out.println("s1 == s2? " + (s1 == s2));  
  
String s3 = new String("Pedro");  
System.out.println("s3 == s1? " + (s3 == s1));
```

Qual o resultado você espera ver? Tem certeza?

Veja a saída abaixo:

```
s1 == s2? true  
s3 == s1? false
```

A primeira conclusão que podemos tirar é que não se compara o conteúdo de `String` com `==`. O operador de comparação `==` irá comparar a referência dos objetos, ou seja, para qual endereço na memória a variável está "apontando". No caso, `s1` e `s2` referenciam o mesmo objeto, já `s3` referencia um outro objeto, pois instanciamos usando `new String`.

Ainda assim, gostaríamos que todas as comparações do exemplo resultassem em `true`, pois o conteúdo é o mesmo: *Pedro*. Vamos então utilizar o método correto para comparação de **conteúdos** de `String`, o `equals`.

```
String s1 = "Pedro";  
String s2 = "Pedro";  
  
System.out.println("s1.equals(s2)? " + (s1.equals(s2)));  
  
String s3 = new String("Pedro");  
System.out.println("s3.equals(s1)? " + (s3.equals(s1)));
```

Veja que a saída agora é como gostaríamos que fosse.

```
s1.equals(s2)? true  
s3.equals(s1)? true
```

Uma String é uma sequência de caracteres, isso significa que cada posição tem um índice que inicia com zero e termina com o tamanho da String - 1.

Veja a figura abaixo. A palavra "ALGAWORKS" tem 9 caracteres, a primeira posição tem índice 0 e a última 8.

A	L	G	A	W	O	R	K	S
0	1	2	3	4	5	6	7	8

Através do método `charAt`, conseguimos recuperar o caractere, passando o índice como parâmetro.

```
String s = "ALGAWORKS";  
  
System.out.println(s.charAt(0));  
System.out.println(s.charAt(2));  
System.out.println(s.charAt(8));  
System.out.println(s.charAt(9));
```

Veja a saída abaixo:

```
A  
G  
S  
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:  
    String index out of range: 9  
    at java.lang.String.charAt(String.java:658)  
    at teste.Teste.main(Teste.java:11)
```

Se utilizarmos um índice negativo ou maior que o tamanho da String, irá ser lançada uma exceção.

Outro método muito importante é o `length()`, que retorna o número de caracteres da string.

```
String s = "Cursos online";  
  
System.out.println(s.length());
```

Veja a saída abaixo e repare que espaços também são considerados para determinar o tamanho da string.

13

Uma outra forma de criar strings e que merece destaque, é através do construtor, que recebe um array de char.

```
char[] array = {'A', 'L', 'G', 'A', 'W', 'O', 'R', 'K', 'S'};

String s = new String(array);

System.out.println(s);
```

Saída:

ALGAWORKS

Para recuperar trechos dentro de uma String, podemos usar o método substring.

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

Na primeira versão, é gerada uma nova String a partir do índice informado e se prolonga até o fim do objeto.

```
String s = "Desenvolvimento Java";
System.out.println(s.substring(16));
```

Saída:

Java

A segunda versão também irá gerar uma nova String, começando no beginIndex e terminando em endIndex, porém sem adicionar o último caractere.

```
String s = "Cursos online de desenvolvimento de software";

System.out.println(s.substring(7, 12));
```

onlin

Repare que o índice na posição 12, a letra "e", não foi incluída na nova String.

O método trim é útil para remover espaços do início e fim da String.

```
String s = "  AlgaWorks  ";
System.out.println "\"" + s.trim() + "\"");
```

"AlgaWorks"

Um método que você vai usar muito no seu dia a dia é o `split`.

```
public String[] split(String regex)
```

A `String` será "quebrada" nos pontos que o *regex* for encontrada e um novo array será gerado, com as partes da string.

```
String s = "Cursos_online_de_desenvolvimento_de_software";
```

```
String[] array = s.split("_");
```

```
for (String a : array) {  
    System.out.println(a);  
}
```

Cursos  
online  
de  
desenvolvimento  
de  
software

O método `indexOf` devolve a posição da primeira ocorrência na `String`.

```
String s = "Java";  
int indice = s.indexOf('a');  
System.out.println(indice);
```

Resultado:

1

Se usássemos `lastIndexOf`, a busca seria feita de trás pra frente.

```
String s = "Java";  
int indice = s.lastIndexOf('a');  
System.out.println(indice);
```

Saída:

3

O método `replaceAll` é usado para substituir trechos da `String` por outra `String`.  
Veja a assinatura abaixo:

```
public String replaceAll(String regex, String replacement)
```

O primeiro argumento é um *regex* que será usada na pesquisa da string. O segundo é o conteúdo que será substituído onde ocorrências da *regex* forem encontradas.

```
String s = "Desenvolvimento Java";  
System.out.println(s.replaceAll("Java", "de software"));
```

Resultado:

Desenvolvimento de software

Como você deve ter percebido, essa é uma classe com vários métodos que realizam as mais diversas operações em uma string. Para conhecer mais métodos e detalhes da classe String, acesse a documentação em <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>.

## Classe StringBuffer

Um objeto do tipo StringBuffer é uma String que pode ser alterada. Isso significa que, se você tiver um código que irá precisar concatenar várias strings, é melhor usar StringBuffer.

Outro detalhe importante é que StringBuffer é *thread-safe*, ou seja, seus métodos são sincronizados e somente uma thread consegue alterar seu conteúdo por vez.

Os principais métodos de StringBuffer são append e insert. Eles são sobrecarregados para aceitar vários tipos de dados e servem para adicionar caracteres no final ou em um ponto específico, respectivamente.

```
// Não podemos iniciar StringBuffer diretamente com aspas duplas  
StringBuffer sb = new StringBuffer();  
  
sb.append("Bem vindo ao curso de Java.");  
sb.append("Fique a vontade para tirar suas dúvidas.");  
sb.append("Estamos aqui para te ajudar a aprender.");  
  
System.out.println(sb);
```

Repare na saída que cada String foi concatenada e o objeto sb foi alterado.

Bem vindo ao curso de Java.Fique a vontade para tirar suas dúvidas. Estamos aqui para te ajudar a aprender.

Com o método insert dizemos em qual posição gostaríamos de adicionar o conteúdo. Por exemplo:

```
StringBuffer sb = new StringBuffer();

sb.append("Bem vindo ao curso de Java.");
sb.append("Fique a vontade para tirar suas dúvidas.");
sb.append("Estamos aqui para te ajudar a aprender.");

sb.insert(26, " da Algaworks");

System.out.println(sb);
```

Saída:

Bem vindo ao curso de Java da Algaworks.Fique a vontade para tirar suas dúvidas.Estamos aqui para te ajudar a aprender.

Veja mais detalhes da classe StringBuffer em <http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html>.

## Classe StringBuilder

Objetos do tipo StringBuilder também são strings que podem ser alteradas, assim como StringBuffer, e os métodos são praticamente os mesmos.

Quer saber qual a diferença? StringBuilder não é *thread-safe* ([http://en.wikipedia.org/wiki/Thread\\_\(computing\)](http://en.wikipedia.org/wiki/Thread_(computing))).

Isso significa que objetos deste tipo são mais rápidos que do tipo StringBuffer.

A não ser que você esteja realmente tentando compartilhar uma string em diferentes threads, use sempre StringBuilder.

```
// Não podemos iniciar StringBuilder diretamente com aspas duplas
StringBuilder sb = new StringBuilder();

sb.append("Bem vindo ao curso de Java.");
sb.append("Fique a vontade para tirar suas dúvidas.");
sb.append("Estamos aqui para te ajudar a aprender.");

sb.insert(26, " da Algaworks");

System.out.println(sb);
```

Saída:

Bem vindo ao curso de Java da Algaworks.Fique a vontade para tirar suas dúvidas. Estamos aqui para te ajudar a aprender.

## 14.4. Trabalhando com datas

Conhecer a API de data do Java é essencial para trabalhar em quase todos os projetos. Você pode usar as classes dessa API para saber a data que um pedido foi emitido ou a hora que um usuário alterou seu perfil no sistema, por exemplo. Também é possível fazer operações com datas, como comparações, adicionar ou remover dias, meses, anos, horas, minutos, segundos e milissegundos.

As principais classes para se trabalhar com datas são `java.util.Calendar` e `java.util.Date`. A primeira é mais completa e recomendada, mas a segunda também é importante e deve ser conhecida, já que muitos frameworks e APIs a utilizam.

Para começar, vamos recuperar a data atual do sistema e imprimir na tela.

```
Calendar agoraComCalendar = Calendar.getInstance();
Date agoraComDate = new Date();

System.out.println("agora com calendar: " + agoraComCalendar);
System.out.println();
System.out.println("agora com date: " + agoraComDate);
```

Veja a saída:

```
agora com calendar: java.util.GregorianCalendar[time=1377600758868,
areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=
sun.util.calendar.ZoneInfo[id="America/Sao_Paulo",
offset=-10800000,dstSavings=3600000,useDaylight=true,
transitions=129,lastRule=java.util.SimpleTimeZone[id=
America/Sao_Paulo,offset=-10800000,dstSavings=3600000,useDaylight=
true,startYear=0,startMode=3,startMonth=9,startDay=15,
startDayOfWeek=1,startTime=0,startTimeMode=0,endMode=3,endMonth=1,
endDay=15,endDayOfWeek=1,endTime=0,endTimeMode=0]],firstDayOfWeek=1,
minimalDaysInFirstWeek=1,ERA=1,YEAR=2013,MONTH=7,WEEK_OF_YEAR=35,
WEEK_OF_MONTH=5,DAY_OF_MONTH=27,DAY_OF_YEAR=239,DAY_OF_WEEK=3,
DAY_OF_WEEK_IN_MONTH=4,AM_PM=0,HOUR=7,HOUR_OF_DAY=7,MINUTE=52,
SECOND=38,MILLISECOND=868,ZONE_OFFSET=-10800000,DST_OFFSET=0]
```

```
agora com date: Tue Aug 27 07:52:38 BRT 2013
```

Não ficou muito bonito essa saída, não é verdade? Vamos aprender a formatar a saída usando a classe `SimpleDateFormat`, do pacote `java.text`.

Com essa classe, conseguimos formatar objetos em um texto apresentável, ou seja, mostrar ao usuário do seu sistema da maneira que ele está mais acostumado a ler datas. Vamos analisar o código abaixo:

```

Calendar agoraComCalendar = Calendar.getInstance();
Date agoraComDate = new Date();

DateFormat formatador = new SimpleDateFormat("dd/MM/yyyy");

String agoraComCalendarFormatado = formatador.format(
    agoraComCalendar.getTime());
String agoraComDateFormatado = formatador.format(agoraComDate);

System.out.println("agora com calendar: " + agoraComCalendarFormatado);
System.out.println("agora com date: " + agoraComDateFormatado);

```

A classe `java.text.SimpleDateFormat` implementa a interface `java.text.DateFormat`. Repare que no construtor da classe `java.text.SimpleDateFormat`, passamos uma string `"dd/MM/yyyy"`, que é o padrão do formatador que estamos criando.

Analise a saída abaixo:

```

agora com calendar: 27/08/2013
agora com date: 27/08/2013

```

Veja que a data agora foi impressa na tela seguindo o padrão que definimos. Talvez você se pergunte, porque o `"MM"` está em maiúsculo e `"dd"` e `"yyyy"` em minúsculo? A resposta é simples, o pessoal que criou o padrão de formatação definiu assim. :)

Se você acessar a documentação da classe `SimpleDateFormat` em <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>, verá todos os códigos para formatar a data e hora.

Seguem alguns deles:

- `y` → ano. `yy` formata com dois dígitos e `yyyy` com quatro dígitos.
- `M` → mês do ano
- `d` → dia do mês
- `a` → marcador para horário em formato 12hs: am/pm
- `H` → hora no dia. Começa em 0 e termina em 23
- `K` → hora no formato 12hs. Começa em 0 termina em 11
- `m` → minutos
- `s` → segundos



Outro detalhe importante que precisamos destacar no exemplo acima, é a conversão de Calendar para Date, através do método getTime(). Precisamos dessa operação pois format de SimpleDateFormat só aceita objetos do tipo Date.

Vamos fazer a operação contrária, simular uma entrada de usuário através de uma String e transformar em um objeto de data.

```
String dataNascimentoUsuario = "15/05/1990";

DateFormat formatador = new SimpleDateFormat("dd/MM/yyyy");

try {
    Date dataNascimento = formatador.parse(dataNascimentoUsuario);
    System.out.println("Data é: " + dataNascimento);
} catch (ParseException e) {
    System.err.println("Formatao inválido. " + e.getMessage());
    System.exit(1);
}
```

O método que utilizamos agora é o parse. Ele recebe uma String e devolve um Date. Veja a saída abaixo:

Data é: Tue May 15 00:00:00 BRT 1990

Repare que tivemos que tratar a exceção ParseException, pois caso a entrada estivesse em um formato inválido, ela seria lançada. Se trocarmos a data de nascimento para um valor incorreto, por exemplo "15051990", receberíamos a seguinte mensagem na tela:

Formatao inválido. Unparseable date: "15151990"

Isso é muito útil para solicitarmos ao usuário a data no formato correto.

A classe Calendar possui vários métodos para manipulação de datas. Entre eles está o add, que permite acrescentar e remover unidades da data.

Para exemplificar, vamos supor que precisamos gerar as datas de vencimento do financiamento de um veículo. O cliente escolheu o dia 10 para pagamento e fez o financiamento em 12 vezes.

```
DateFormat formatador = new SimpleDateFormat("dd/MM/yyyy");

try {
    Date dataVencimento = formatador.parse("20/10/2013");

    Calendar calendar = Calendar.getInstance();
    calendar.setTime(dataVencimento);

    for (int i = 1; i <= 12; i++) {
```

```

        System.out.println("Parcela \"" + i + "\" vence em: "
            + formatador.format(calendar.getTime()));
        calendar.add(Calendar.MONTH, 1);
    }
} catch (ParseException e) {
    System.err.println("Formatao inválido. " + e.getMessage());
    System.exit(1);
}

```

A assinatura do método é muito simples, como primeiro argumento, informamos qual unidade queremos adicionar e com o segundo, o valor que queremos somar. Se passarmos um valor negativo, será subtraído, e um valor positivo, somado. Veja a saída da execução do código:

```

Parcela "1" vence em: 20/10/2013
Parcela "2" vence em: 20/11/2013
Parcela "3" vence em: 20/12/2013
Parcela "4" vence em: 20/01/2014
Parcela "5" vence em: 20/02/2014
Parcela "6" vence em: 20/03/2014
Parcela "7" vence em: 20/04/2014
Parcela "8" vence em: 20/05/2014
Parcela "9" vence em: 20/06/2014
Parcela "10" vence em: 20/07/2014
Parcela "11" vence em: 20/08/2014
Parcela "12" vence em: 20/09/2014

```

Repare que `Calendar` já cuida de avançar o ano quando mudamos de dezembro para janeiro. Existem várias outras constantes que podemos utilizar para adicionar em outras unidades, como:

```

Calendar.DAY_OF_MONTH
Calendar.YEAR
Calendar.HOUR_OF_DAY
Calendar.MINUTE

```

Vamos criar um exemplo usando configurações com tempo (horas, minutos e segundos) e aprender o método `roll`. Ele é bem parecido com o método `add`, também funciona adicionando ou removendo unidades do valor representado, mas com a grande diferença de não alterar os campos maiores. Analise o código abaixo:

```

DateFormat formatador = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

try {
    Date dataExpiracao = formatador.parse("20/10/2013 14:30:00");

    Calendar calendar = Calendar.getInstance();
    calendar.setTime(dataExpiracao);
}

```

```

        calendar.roll(Calendar.DAY_OF_MONTH, -25);

        System.out.println("Nova data de expiração: "
            + formatador.format(calendar.getTime()));
    } catch (ParseException e) {
        System.err.println("Formatao inválido. " + e.getMessage());
        System.exit(1);
    }
}

```

Saída:

Nova data de expiração: 26/10/2013 14:30:00

Veja que agora removemos 25 dias e chegamos ao dia 26 do mesmo mês. O método `roll` **não** altera nada além da unidade que estamos informando, neste caso o dia do mês. A conta é simples, são 20 dias até chegar em 31 de outubro e remover mais 5 dias até 26.

A API de `Calendar` é bem completa, com vários detalhes. Veja mais em <http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html>.

## 14.5. Trabalhando com números

Antes de entendermos essa API tão importante do Java, vamos executar um exemplo, onde a regra pode ser simples, mas muito comum durante o desenvolvimento de sistemas.

A ideia é simples, vamos imaginar um caixa eletrônico para pagamento das compras em um supermercado. Só que nosso sistema tem que estar preparado para receber em reais, a partir de um valor em dólar.

O código abaixo simula com valores literais o fluxo de funcionamento do nosso sistema, fazendo as contas com a conversão de moedas e cálculo do troco, mas existe uma pegadinha que vamos descobrir agora. Leia o código abaixo:

```

double valorEmDolar = 10.00;
double cotacaoDolar = 2.43;
double valorEmReais = valorEmDolar * cotacaoDolar;

// Valor em reais aqui é: R$24,30
// O cliente deposita R$25,00 para pagar.
// Temos que devolver R$0,70 de troco.

double troco = 25.0 - valorEmReais;

```

```

if (troco == 0.70) {
    System.out.println("Tudo certo, isso é matemática!");
} else {
    System.out.println("Impossível de acontecer. Será? Valor do troco: "
        + troco);
}

```

Você apostaria que esse código entra no primeiro if? Acha que é impossível entrar no else? Então veja a saída abaixo:

Impossível de acontecer. Será? Valor do troco: 0.6999999999999993

Surpresa! E agora vai um conselho, trabalhar com double fazendo esse tipo de conta é muito propenso a erros, devido a aproximações feitas pelo processador. O ideal é utilizarmos um tipo de dados mais apropriado, preparado para lidar com essa situação, o `java.math.BigDecimal`.

Veja abaixo a versão do mesmo sistema usando `BigDecimal`, mas dessa vez irá funcionar como esperado.

```

BigDecimal valorEmDolar = new BigDecimal("10.00");
BigDecimal cotacaoDolar = new BigDecimal("2.43");
BigDecimal valorEmReais = valorEmDolar.multiply(cotacaoDolar);

// Valor em reais aqui é: R$24,30
// O cliente deposita R$25,00 para pagar.
// Temos que devolver R$0,70 de troco.

BigDecimal pagamento = new BigDecimal("25.00");

BigDecimal troco = pagamento.subtract(valorEmReais);

if (troco.compareTo(new BigDecimal("0.70")) == 0) {
    System.out.println("Tudo certo, isso é matemática! "
        + "Valor do troco: " + troco);
} else {
    System.out.println("Impossível de acontecer. Será? "
        + "Valor do troco: " + troco);
}

```

Veja a saída abaixo do código funcionando de acordo com a matemática tradicional.

Tudo certo, isso é matemática! Valor do troco: 0.7000

Vamos analisar o código para entendermos como funciona a classe `BigDecimal`. Para criar um valor, usamos o construtor que recebe uma `String` como parâmetro, contendo o valor que queremos iniciar.

```
BigDecimal valorEmDolar = new BigDecimal("10.00");
```

Quando queremos multiplicar valores, usamos o método `multiply` e, uma parte importante de se aprender aqui, é que `BigDecimal` é **imutável**, ou seja, seu valor não se altera nunca! Veja o exemplo abaixo:

```
BigDecimal quantidade = new BigDecimal("5");
BigDecimal valor = new BigDecimal("2.50");

valor.multiply(quantidade); // Não irá alterar "valor"

System.out.println("Valor é: " + valor);

valor = valor.multiply(quantidade);

System.out.println("Valor agora é: " + valor);
```

Somente na segunda parte a variável `valor` será alterada. Veja a saída abaixo:

```
Valor é: 2.50
Valor agora é: 12.50
```

Repare que usamos strings no construtor de `BigDecimal`. Ele é sobrecarregado para receber vários tipos de dados, mas se ainda for um valor de ponto flutuante, como `double`, a representação em `String` funciona melhor. Veja no exemplo abaixo o mesmo problema da aproximação que tivemos anteriormente.

```
BigDecimal bd = new BigDecimal(1.52);
System.out.println(bd.toString());

1.520000000000000017763568394002504646778106689453125
```

O mesmo não irá acontecer usando `String`.

```
BigDecimal bd = new BigDecimal("1.52");
System.out.println(bd.toString());
```

Agora a saída sairá corretamente:

```
1.52
```

A comparação com `BigDecimal` é feita com o método `compareTo`, veja a assinatura abaixo:

```
public int compareTo(BigDecimal val)
```

Esse método compara o objeto atual com o `BigDecimal` informado e retorna um `int` como resultado. Com ele, conseguimos fazer todas as operações de comparação (<, ==,

>, >=, !=, <=), pois o método retorna -1, 0 ou 1, caso o objeto seja menor, igual ou maior ao parâmetro, respectivamente.

```
BigDecimal a = new BigDecimal(10);

if (a.compareTo(new BigDecimal(10)) == 0) {
    System.out.println("O valor de \"a\" é igual a 10");
}

if (a.compareTo(new BigDecimal(15)) < 0) {
    System.out.println("O valor de \"a\" é menor que 15");
}

if (a.compareTo(new BigDecimal(5)) > 0) {
    System.out.println("O valor de \"a\" é maior que 5");
}
```

Saída:

```
O valor de "a" é igual a 10
O valor de "a" é menor que 15
O valor de "a" é maior que 5
```

Veja mais detalhes da classe `BigDecimal` em <http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>.

## Formatação numérica

Vamos analisar a classe `java.text.DecimalFormat`, que nos auxiliará na formatação de valores decimais para uma apresentação mais elegante ao usuário.

Utilizamos um padrão formado por alguns códigos que guiam como deve ser a formatação. Vamos criar um sistema para um posto de combustível. Como sabemos, o valor do litro da gasolina é dado sempre com 3 casas decimais, apesar de pagarmos apenas com duas, então vamos mostrar ao cliente exatamente como ele irá pagar.

```
double litrosAbastecidos = 38.42;
double valorDoLitro = 2.299;

// totalAPagar terá o valor de 88.32758
double totalAPagar = litrosAbastecidos * valorDoLitro;

// para mostrar ao usuário, queremos formatar com apenas
// 2 casas decimais
DecimalFormat formatador = new DecimalFormat("##0.00");
```

```
String valorAPagar = formatador.format(totalAPagar);
System.out.println("O valor da conta foi: " + valorAPagar);
```

A saída ficou um pouco mais apresentável, veja abaixo:

O valor da conta foi: 88.33

Repare que no código usamos `"##0.00"` para criar nosso padrão. O zero serve para forçar sempre ter um caractere naquela posição, ou seja, se o valor tivesse sido redondo, por exemplo, 15 reais, seria apresentado ao usuário 15.00, e não apenas 15. O cerquilha (#) é um código de formatação opcional, caso não tenha valor ali, nada será mostrado.

Veja que o separador das casas decimais usada foi `"."` e não `","` como é no Brasil. Para alterarmos isso, podemos criar o `formatador` usando `java.text.NumberFormat` e passarmos a localidade como parâmetro.

```
DecimalFormat formatador =
    (DecimalFormat) NumberFormat.getInstance(new Locale("pt", "BR"));
formatador.applyPattern("##0.00");

String valorAPagar = formatador.format(totalAPagar);
System.out.println("O valor da conta foi: " + valorAPagar);
```

Saída:

O valor da conta foi: 88,33

Melhorou ainda mais usando `","` como separador decimal, mas ainda não temos o `"R$"` na frente do valor. Vamos então simplificar nosso código, usando apenas `NumberFormat` para recuperar um formatador de moedas do Brasil.

```
NumberFormat formatador =
    NumberFormat.getCurrencyInstance(new Locale("pt", "BR"));

String valorAPagar = formatador.format(totalAPagar);
System.out.println("O valor da conta foi: " + valorAPagar);
```

Veja que agora nem precisamos informar o padrão, a API já sabe que moedas são apresentadas com duas casas decimais e que o símbolo monetário do Brasil é o `"R$"`.

O valor da conta foi: R\$ 88,33

Com `NumberFormat`, podemos fazer o *parse* e transformar uma `String` em um número, para então fazermos operações matemáticas. Isso é bastante útil quando estamos recebendo entrada do usuário ou lendo de algum arquivo, por exemplo.

```

NumberFormat formatador = NumberFormat.getInstance(
    new Locale("pt", "BR"));

try {
    Double total = (Double) formatador.parse("40,20");

    // podemos fazer contas agora, como somar mais 5, por exemplo
    total += 5.0;

    System.out.println("Novo valor: " + total);
} catch (ParseException e) {
    System.err.println("Não conseguiu fazer o parse da String. "
        + e.getMessage());
    System.exit(1);
}

```

Repare que, como criamos o `NumberFormat` com a localidade brasileira, o caractere do separador decimal usado foi a vírgula. Veja a saída abaixo:

Novo valor: 45.2

Veja mais detalhes de `NumberFormat` em <http://docs.oracle.com/javase/7/docs/api/java/text/NumberFormat.html> e de `DecimalFormat` em <http://docs.oracle.com/javase/7/docs/api/java/text/DecimalFormat.html>.



# Collections Framework

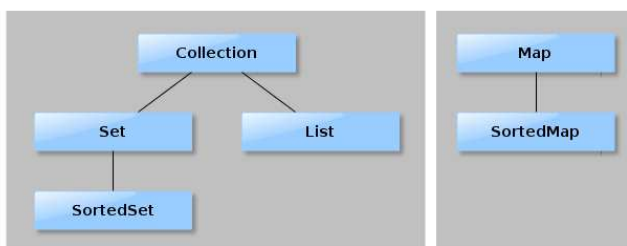
## 15.1. Introdução a coleções

Quando falamos em coleções, logo pensamos em vários objetos, de um determinado tipo, agrupados. Podemos colecionar livros, figurinhas e até carros (para os mais afortunados).

E é justamente essa ideia da API de coleções do Java, agrupar elementos em uma unidade só. As coleções são usadas para armazenar, recuperar e manipular objetos.

O *Collections Framework* é composto por várias interfaces e classes para os mais diversos tipos de coleções, trazendo vários benefícios, como reduzir esforço de programação e melhorar a velocidade e qualidade do código.

Cada tipo de coleção é derivado de uma interface, e conhecer a diferença entre cada uma delas é objetivo deste capítulo. Veja no diagrama abaixo a hierarquia dessas interfaces.



Tanto Set e List são um tipo especial de Collection e SortedSet é um tipo de Set. Veja que a interface Map não é uma Collection.

- **Collection:** é a raiz das coleções. A plataforma Java não fornece nenhuma implementação direta dessa interface, normalmente ela é usada quando queremos buscar o máximo de generalização possível, ou seja, não importa se o objeto será do tipo `Set` ou `List`.
- **Set:** é uma coleção que não irá aceitar elementos duplicados, conhecida como conjuntos de itens. Por exemplo, clientes de uma loja, carros de uma locadora de veículos, etc. Tudo que você pensar que não podem existir dois ou mais iguais em um mesmo conjunto, são candidatos a pertencerem a um `Set`.
- **List:** coleções deste tipo aceitam elementos duplicados e são ordenados em uma sequência, normalmente a de inserção. Cada objeto na lista recebe um índice (um inteiro) por posição. Usamos `List` para criar listas de itens de um pedido de compra, por exemplo.
- **Map:** são itens que possuem uma identificação exclusiva, ou seja, possuem chaves para acessar cada valor do mapa. A chave não pode ser repetida, tem que ser única. Podemos usar `Map` para criar uma lista telefônica, onde cada chave é o número do telefone e o nome é o valor.
- **SortedSet:** é um `Set` que mantém a ordem crescente dos elementos dentro dela. São usadas quando precisamos tirar proveito da ordenação, como armazenar os estados brasileiros em ordem alfabética.
- **SortedMap:** um `Map` que mantém as chaves ordenadas. É útil para mantermos dicionários, por exemplo.

*Collections Framework* fornece implementações para todas essas interfaces, com exceção de `Collection`, que é apenas um supertipo mais genérico. Nas próximas seções, estudaremos mais detalhes de cada classe e como usá-las.

## 15.2. Listas - Interface `List`

As listas do tipo `java.util.List` são ordenadas por um índice (um número inteiro que representa a posição) e também aceitam objetos duplicados armazenados na coleção.

Podemos pensar que listas são como *arrays* que pode crescer e diminuir, ou seja, no momento que quisermos, podemos adicionar ou remover elementos sem nos preocupar se irá "caber" na lista ou não.

Existem 3 implementações da interface `List`, `java.util.ArrayList`, `java.util.LinkedList` e a que foi refatorada para entrar na API de coleções, `java.util.Vector`. As duas primeiras não são *thread-safe*, mas são as de melhor performance e as mais usadas na maioria das aplicações.

## ArrayList

`ArrayList` é uma implementação de lista que proporciona iteração e acesso aleatório com rapidez.

Vamos ver um exemplo onde adicionaremos nomes em uma lista e depois varreremos a coleção para imprimir os elementos na tela. Não se assuste com o código abaixo, iremos detalhar cada linha para não ficar dúvidas.

```
// Instancia lista - não precisamos dizer o tamanho
List<String> listaDeNomes = new ArrayList<String>();

// Adiciona elemento na lista de strings
listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

// Imprime o elemento na posição 1
String nomePosicao1 = listaDeNomes.get(1);
System.out.println("Elemento na posição 1: " + nomePosicao1);
System.out.println();

// Percorre a lista
for (int i = 0; i < listaDeNomes.size(); i++) {
    System.out.println("listaDeNomes.get(" + i + ") = "
        + listaDeNomes.get(i));
}
```

Veja a saída:

Elemento na posição 1: João

```
listaDeNomes.get(0) = Pedro
listaDeNomes.get(1) = João
listaDeNomes.get(2) = Maria
```

Na primeira linha, instanciamos uma `ArrayList` e atribuímos a uma variável `listaDeNomes` do tipo `List`. É muito bom declararmos nossas variáveis onde o tipo é a interface, isso ajuda no desacoplamento do código, principalmente quando algum outro framework for utilizado, portanto, sempre que possível, tente programar dessa forma.

Outro detalhe importante dessa instanciação é o fato de não precisarmos dizer o tamanho da lista. Poderíamos até dizer esse tamanho no construtor, existe outra versão que aceita um inteiro como parâmetro, para dizer a capacidade inicial. Veja a assinatura desse construtor abaixo:

```
public ArrayList(int initialCapacity)
```

Continuando com a instanciação, vemos que existe *generics* envolvido através de `List` e `ArrayList`. Esse tipo usado na declaração da variável, define que a lista aceita somente aquele tipo de objeto, ela é avaliada em tempo de compilação e isso é muito útil, pois se tentarmos passar algum elemento inválido, não irá nem compilar. Veja o código abaixo, ele **não compila**.

```
List<String> listaDeNomes = new ArrayList<String>();  
listaDeNomes.add(new Integer(10));
```

O compilador reclama, informando que o método `add` aceita somente `String`. Veja abaixo a mensagem original:

The method `add(String)` in the type `List` is not applicable for the arguments `(Integer)`

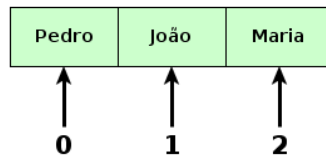
Agora, se você já usa o Java 7, tem uma novidade que irá economizar a digitação: são os chamados **diamantes**, que na prática, faz com que não precisemos dizer o tipo da classe na instanciação da lista. Podemos deixar apenas `<>`. Veja abaixo o mesmo código.

```
List<String> listaDeNomes = new ArrayList<>();  
listaDeNomes.add("Pedro");
```

Continuando a análise do código, vemos que através do método `add(String)`, conseguimos adicionar elementos na lista.

```
listaDeNomes.add("Pedro");
```

Como essa é uma coleção ordenada por um índice, cada elemento é inserido em uma posição e um inteiro representa aquela posição, iniciando com zero. Veja a imagem abaixo de como está a `listaDeNomes` depois da inserção dos três nomes nela.



E é por iniciar com zero, que ao executarmos o código `listaDeNomes.get(1)`, é impresso "João" na tela. O método `get` recebe um inteiro como argumento e retorna o elemento encontrado naquela posição.

Um detalhe muito importante é que, devido ao *generics*, o método retornará um objeto exatamente do tipo da lista, no nosso caso, `String`. Não precisamos fazer nenhum `cast` chato e feio. Veja abaixo um exemplo com duas listas, uma de `String` e outra de `Integer`.

```
List<String> nomes = new ArrayList<>();
List<Integer> senhas = new ArrayList<>();

nomes.add("Ricardo");
senhas.add(new Integer(1010));

String nome = nomes.get(0);
Integer senha = senhas.get(0);
```

Para percorrer a lista, usamos o método `size()`, que devolve um inteiro contendo a quantidade de elementos na lista, neste exemplo, 3. Existem ainda outras formas de percorrermos os elementos da coleção, usando `java.util.Iterator` ou o *for avançado*.

O `Iterator` permite percorrer os elementos da lista na ordem em que eles foram inseridos. Uma versão mais avançada, chamada `ListIterator`, pode ser usada para iterar no sentido contrário, mas para isso, precisamos colocar o *cursor* na última posição.

```
Iterator<String> iterator = listaDeNomes.iterator();

while (iterator.hasNext()) {
    String nome = iterator.next();
    System.out.println("Nome: " + nome);
}

System.out.println();
System.out.println("De trás pra frente...");

// para percorrer de trás pra frente, o cursor deve estar
// no fim da lista
ListIterator<String> listIterator =
    listaDeNomes.listIterator(listaDeNomes.size());
```

```
while (listIterator.hasPrevious()) {
    String nome = listIterator.previous();
    System.out.println("Nome: " + nome);
}
```

Saída:

Nome: Pedro  
 Nome: João  
 Nome: Maria

De trás pra frente...

Nome: Maria  
 Nome: João  
 Nome: Pedro

Os métodos `hasNext()` e `hasPrevious()` retornam um boolean para informar se existem mais elementos a frente ou atrás. Com os métodos `next()` e `previous()`, recuperamos o elemento da lista. Mais uma vez, graças ao *generics*, não precisamos de nenhum *cast*.

O *for avançado* é usado para percorrer elementos em uma coleção. Apesar do nome, ele é bem simples de se usar e aprender. Vamos usar o mesmo exemplo anterior para ilustrar sua utilização. Veja o código abaixo:

```
List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}
```

Sabemos que cada elemento na lista é do tipo `String`, graças ao *generics*, e *for avançado* utiliza dessa facilidade. Sua estrutura é definida da seguinte forma:

```
for (TipoDoElementoDaLista elemento : lista)
```

Veja que `TipoDoElementoDaLista` será o tipo definido usando *generics*, e que `elemento` será o nome da variável que receberá cada objeto da lista durante a iteração.

Agora que entendemos como criar e percorrer uma lista, vamos ver mais detalhes de outros métodos úteis dessa poderosa API do Java.

## Adicionando elementos

Não é só o método `add(E element)` que insere elementos em uma lista. Temos também os métodos abaixo:

```
public void add(int index, E element)
public boolean addAll(Collection< ? extends E > c)
public boolean addAll(int index, Collection< ? extends E > c)
```

O primeiro deles facilita a inserção em uma determinada posição.

```
List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

listaDeNomes.add(1, "José");

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}
```

Neste exemplo, "José" será inserido na posição 1, no lugar de "João", que irá subir uma posição, e assim sucessivamente a todos os elementos da lista. Veja a saída abaixo:

```
Nome: Pedro
Nome: José
Nome: João
Nome: Maria
```

Já o método `addAll`, insere elementos de uma outra coleção na lista, sendo que a primeira versão insere no final da lista e a segunda na posição especificada pelo índice.

```
List<String> listaDeNomesPrincipal = new ArrayList<>();

listaDeNomesPrincipal.add("Pedro");
listaDeNomesPrincipal.add("João");
listaDeNomesPrincipal.add("Maria");

// Essa lista pode criada de um sistema externo,
// ou de um arquivo, por exemplo
List<String> listaDeNomesAdicional = new ArrayList<>();
listaDeNomesAdicional.add("Ricardo");
listaDeNomesAdicional.add("Beatriz");

// Adiciona a lista adicional, na lista principal
listaDeNomesPrincipal.addAll(listaDeNomesAdicional);

for (String nome : listaDeNomesPrincipal) {
```

```
        System.out.println("Nome: " + nome);
    }
```

Veja a saída:

```
Nome: Pedro
Nome: João
Nome: Maria
Nome: Ricardo
Nome: Beatriz
```

## Limpendo a lista

O método `clear()` remove todos os elementos da coleção. Vamos aproveitar e ver o que acontece quando tentamos acessar uma posição da lista depois de ter removido todos eles.

```
List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}

// Não deixa nenhum elemento na lista
listaDeNomes.clear();

String nome = listaDeNomes.get(0);

// Não chega aqui, lança uma exception antes
System.out.println("Nome: " + nome);
```

Veja que a exceção `IndexOutOfBoundsException` é lançada, informando que o índice 0 não existe, pois o tamanho da lista é zero. Essa exceção também é lançada por vários outros métodos, quando tentamos acessar alguma posição inexistente.

```
Nome: Pedro
Nome: João
Nome: Maria
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index: 0, Size: 0
    at java.util.ArrayList.rangeCheck(ArrayList.java:604)
    at java.util.ArrayList.get(ArrayList.java:382)
    at teste.Teste.main(Teste.java:22)
```



Para evitar esse tipo de erro, podemos perguntar a lista se ela está vazia, antes de tentar acessar o elemento, utilizando o método `isEmpty()`, que irá retornar `true` caso nenhum objeto for encontrado na coleção.

```
if (!listaDeNomes.isEmpty()) {  
    // só irá executar caso tenha algum elemento na lista  
    // repare o "!" no início do if  
    String nome = listaDeNomes.get(0);  
    System.out.println("Nome: " + nome);  
}
```

## Buscando elementos

Também podemos descobrir se determinado objeto está dentro da lista. Para isso, temos à disposição os métodos abaixo:

```
public boolean contains(Object o)  
public int indexOf(Object o)  
public int lastIndexOf(Object o)
```

O `boolean contains(Object o)` irá retornar `true`, caso encontre o objeto na coleção, e `false`, caso contrário.

```
List<String> listaDeNomes = new ArrayList<>();  
  
listaDeNomes.add("Pedro");  
listaDeNomes.add("João");  
listaDeNomes.add("Maria");  
  
boolean encontrouJoao = listaDeNomes.contains("João");  
System.out.println("Encontrou João na lista? " + encontrouJoao);  
  
boolean encontrouRicardo = listaDeNomes.contains("Ricardo");  
System.out.println("E Ricardo? " + encontrouRicardo);
```

Veja que ele consegue encontrar "João", que está na lista, mas não encontra "Ricardo".

```
Encontrou João na lista? true  
E Ricardo? false
```

Os métodos `int indexOf(Object o)` e `int lastIndexOf(Object o)` devolvem o índice do elemento encontrado. Já que objetos do tipo `List` podem ter repetições, a primeira versão retorna a posição do primeiro objeto encontrado, já a segunda versão, o último índice. Os dois métodos retornam `-1`, caso não encontrem nada. O código abaixo ilustra a utilização desses dois métodos.

```

List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");
listaDeNomes.add("Ricardo");
listaDeNomes.add("Maria");
listaDeNomes.add("José");

int indiceDaPrimeiraMaria = listaDeNomes.indexOf("Maria");
System.out.println("Índice da primeira Maria: "
    + indiceDaPrimeiraMaria);

int indiceDaUltimaMaria = listaDeNomes.lastIndexOf("Maria");
System.out.println("Índice da última Maria: "
    + indiceDaUltimaMaria);

```

Saída (lembre-se que a primeira posição é zero):

```

Índice da primeira Maria: 2
Índice da última Maria: 4

```

## Removendo elementos

Uma situação muito comum é precisarmos remover elementos de dentro da coleção, e é claro que temos mais de uma forma de fazer isso. Veja abaixo os métodos que vamos aprender.

```

public E remove(int index)
public boolean remove(Object o)
public boolean removeAll(Collection< ? > c)

```

Agora que você já está mais familiarizado com a API, consegue deduzir como esses métodos funcionam, certo?

- `E remove(int index)`: recebe o índice, remove o elemento e o retorna
- `boolean remove(Object o)`: recebe o objeto que será removido, retornando `true` caso consiga removê-lo
- `boolean removeAll(Collection< ? > c)`: recebe a coleção, procura na lista, removendo os elementos que coincidem, e retorna `true` caso consiga remover pelo menos um elemento.

Exemplo com `public E remove(int index)`:

```

List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");

```

```

listaDeNomes.add("João");
listaDeNomes.add("Maria");

String nomeRemovido = listaDeNomes.remove(1);
System.out.println("Nome removido: " + nomeRemovido);
System.out.println();

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}

```

Saída:

Nome removido: João

Nome: Pedro

Nome: Maria

Agora com public E remove(int index):

```

List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

boolean foiRemovido = listaDeNomes.remove("João");
System.out.println("Conseguiu remover? " + foiRemovido);
System.out.println();

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}

```

Saída:

Conseguiu remover? true

Nome: Pedro

Nome: Maria

E por fim, public boolean removeAll(Collection< ? > c):

```

List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

List<String> nomesAApagar = new ArrayList<>();
nomesAApagar.add("Pedro");
nomesAApagar.add("Maria");

```

```
listaDeNomes.removeAll(nomesAApagar);

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}
```

Saída:

Nome: João

## Substituindo um elemento

Se precisarmos substituir um elemento em uma posição, podemos utilizar o método:

```
public E set(int index, E element)
```

Este método irá substituir o elemento e retornar o antigo.

```
List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

String nomeAnteriorNaLista = listaDeNomes.set(1, "Ricardo");
System.out.println("Nome que estava na lista: "
    + nomeAnteriorNaLista);
System.out.println();

for (String nome : listaDeNomes) {
    System.out.println("Nome: " + nome);
}
```

Saída:

Nome que estava na lista: João

Nome: Pedro  
Nome: Ricardo  
Nome: Maria

## Convertendo para arrays

Se um dia você se deparar com alguma API que espera um `String[]` e você tiver um `List`, você precisará transformar a lista em um array. Isso é simples utilizando o método:

```
public <T> T[] toArray(T[] a)
```

Parece complicado de usar? Mas não é! O método recebe o array que irá receber os elementos da lista e o devolve com os elementos adicionados. Caso o tamanho do array não seja suficiente, um novo será criado. Veja o exemplo abaixo.

```
List<String> listaDeNomes = new ArrayList<>();

listaDeNomes.add("Pedro");
listaDeNomes.add("João");
listaDeNomes.add("Maria");

// neste caso, o tamanho do array é do tamanho da lista,
// mas funcionaria se instanciássemos assim: new String[1]
String[] array = new String[listaDeNomes.size()];
array = listaDeNomes.toArray(array);

for (String nome : array) {
    System.out.println("Nome: " + nome);
}
```

Saída:

```
Nome: Pedro
Nome: João
Nome: Maria
```

## 15.3. Conjuntos - Interface Set

Os conjuntos são representados pela classe `java.util.Set`. Sua principal característica é não aceitar elementos duplicados. Esta interface não contém vários métodos como `List`, pois seus elementos não são acessados através de índices.

Seu uso é comum em situações que queremos manter vários elementos únicos, sem repetição. Por exemplo, um sistema irá ler de um arquivo texto vários e-mails. Nosso programa deve enviar um e-mail para cada um na lista, mas não queremos que um cliente receba o mesmo texto duas ou mais vezes, pode ser que alguém tenha errado e digitado nesse arquivo um e-mail duas ou até mais vezes.

Vamos apenas simular essa leitura de arquivo usando uma String com vários e-mails, separados por espaço. Iremos também fingir que estamos enviando o e-mail e mostrar uma mensagem na tela, quando encontrarmos elementos duplicados.

```
// Imagine que essa linha tenha vindo do arquivo
String emailsNoArquivo = "joao@joao.com jose@jose.com joao@joao.com "
    + "maria@maria.com ricardo@ricardo.com maria@maria.com";

// Cada e-mail é separado em uma posição do array
String[] arrayDeEmails = emailsNoArquivo.split(" ");

// Instanciamos um conjunto usando java.util.HashSet
Set<String> listaEmailsUnicos = new HashSet<>();

for (String email : arrayDeEmails) {
    if (listaEmailsUnicos.add(email)) {
        System.out.println("Enviando e-mail para: " + email);
    } else {
        System.out.println("Encontrado e-mail duplicado. "
            + "Não irá enviar e-mail: " + email);
    }
}

System.out.println("\nLista de emails: " + listaEmailsUnicos);
```

Veja que a classe que usamos para instanciar o conjunto foi `java.util.HashSet`. Um detalhe muito importante sobre ela é que não existe garantia da ordem que você irá ler os elementos, ou seja, se você percorrê-la, poderá ter um resultado diferente a cada iteração.

Veja a saída do código abaixo:

```
Enviando e-mail para: joao@joao.com
Enviando e-mail para: jose@jose.com
Encontrado e-mail duplicado. Não irá enviar e-mail: joao@joao.com
Enviando e-mail para: maria@maria.com
Enviando e-mail para: ricardo@ricardo.com
Encontrado e-mail duplicado. Não irá enviar e-mail: maria@maria.com
```

```
Lista de emails: [ricardo@ricardo.com, maria@maria.com, jose@jose.com,
joao@joao.com]
```

Repare que, quando um e-mail que já foi adicionado na lista é encontrado, o método `add` retorna `false`, assim conseguimos saber se o elemento foi ou não adicionado no conjunto.

Por fim, veja o conjunto impresso, com apenas e-mails únicos, sem repetição.

Talvez você esteja se perguntando: se não tem o índice para acessar os elementos, como conseguimos acessar um elemento específico? A resposta é simples, não tem. Você deve percorrer todos até encontrar quem deseja, mas não consegue acessar uma posição específica, mesmo porque não existe ordem, lembra?

Para percorrer o conjunto, podemos usar o `Iterator` ou o *for avançado*.

```
Set<String> emails = new HashSet<>();
emails.add("joao@joao.com");
emails.add("maria@maria.com");
emails.add("ricardo@ricardo.com");
emails.add("jose@jose.com");

System.out.println("for avançado");
for (String email : emails) {
    System.out.println("Email: " + email);
}

System.out.println("\niterator");
Iterator<String> iterator = emails.iterator();
while (iterator.hasNext()) {
    System.out.println("Email: " + iterator.next());
}
```

Saída:

```
for avançado
Email: ricardo@ricardo.com
Email: maria@maria.com
Email: jose@jose.com
Email: joao@joao.com
```

```
iterator
Email: ricardo@ricardo.com
Email: maria@maria.com
Email: jose@jose.com
Email: joao@joao.com
```

Um detalhe muito importante sobre o `HashSet` é justamente o *Hash* no nome dele. Vamos entender como sobrescrever o método `int hashCode()` pode tornar nosso software melhor na próxima seção.

## 15.4. O `equals()` e o `hashCode()`

Já vimos que podemos definir o tipo genérico com `String` tanto em `ArrayList` como em `HashSet`. Agora vamos expandir nossos exemplos e entender que podemos

também usar qualquer outro tipo de classe, inclusive as nossas próprias, e entender um papel muito importante de dois métodos, equals e hashCode.

Vamos começar criando uma classe que será usada para o tipo genérico das coleções. A classe abaixo representa um aluno, que tem um número de matrícula, nome e idade.

```
public class Aluno {

    private Integer matricula;
    private String nome;
    private Integer idade;

    public Aluno(Integer matricula, String nome, Integer idade) {
        this.matricula = matricula;
        this.nome = nome;
        this.idade = idade;
    }

    // getters e setters

}
```

Nosso sistema irá simular uma consulta a um banco de dados para buscar todos os alunos cadastrados. Para isso, criaremos uma classe chamada SimuladorBancoDeDados, com um método static que retorna a lista de alunos.

```
public class SimuladorBancoDeDados {

    public static Collection<Aluno> buscarAlunos() {
        Collection<Aluno> alunos = new ArrayList<>();
        alunos.add(new Aluno(1111, "João Costa", 20));
        alunos.add(new Aluno(1112, "Maria Rita", 21));
        alunos.add(new Aluno(1113, "José de Castro", 22));
        alunos.add(new Aluno(1114, "Ricardo Caetano", 20));

        return alunos;
    }

}
```

Na classe de teste você pode pensar que é simples, basta selecionar um número aleatório do tamanho da lista e, já que estamos com um ArrayList, podemos acessar através do índice.

Mas, repare um detalhe importante, o retorno do método buscarAlunos() é do tipo Collection. Isso significa que temos apenas os métodos comuns a todas as coleções, e o get() não é um desses métodos disponíveis. Aproveitando o tópico, esse tipo



de situação é muito comum quando queremos alcançar o máximo de generalização possível. Veja que `Collection` é a interface mais alta na hierarquia da API de coleções.

Vamos supor que um objeto do tipo `Aluno` fosse entregue ao nosso programa para verificar se ele existe na coleção, e caso verdadeiro, ele é o ganhador. Veja abaixo como ficará nosso código.

```
Collection<Aluno> alunos = SimuladorBancoDeDados.buscarAlunos();

// seleciona um aluno para ganhar um brinde
Aluno alunoSorteado = new Aluno(1111, "João Costa", 20);

// verifica se a lista contém o aluno
if (alunos.contains(alunoSorteado)) {
    System.out.println("Parabéns " + alunoSorteado.getNome()
        + ", você ganhou um brinde especial!");
} else {
    System.out.println("Esse aluno não está cadastrado no "
        + "banco de dados.");
}
```

Através do método `contains`, conseguimos verificar se um determinado objeto está dentro da lista. Nesse caso, o aluno com número de matrícula 1111, de nome "João Costa", de 20 anos, deve ser o ganhador, e se olharmos na classe que simula o banco de dados, vemos que um aluno assim existe lá. Logo, conseguimos deduzir qual será a saída, certo? Ou não? Veja a saída abaixo:

Esse aluno não está cadastrado no banco de dados.

Essa é a saída e não há nada de errado com o `if`. O "problema" está na maneira que o Java compara objetos, que é através do método `equals()`. Todas as classes possuem esse método, pois ele está presente em `Object`, a superclasse de todas as classes.

O que precisamos fazer é sobrescrever este método e através dele, ensinar como deve ser feita a comparação entre objetos do tipo `Aluno`. Como dois alunos devem ser comparados? Podemos escolher o número de matrícula, já que deve ser único no nosso sistema. Assim, a implementação do `equals` ficará como abaixo:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Aluno other = (Aluno) obj;
```

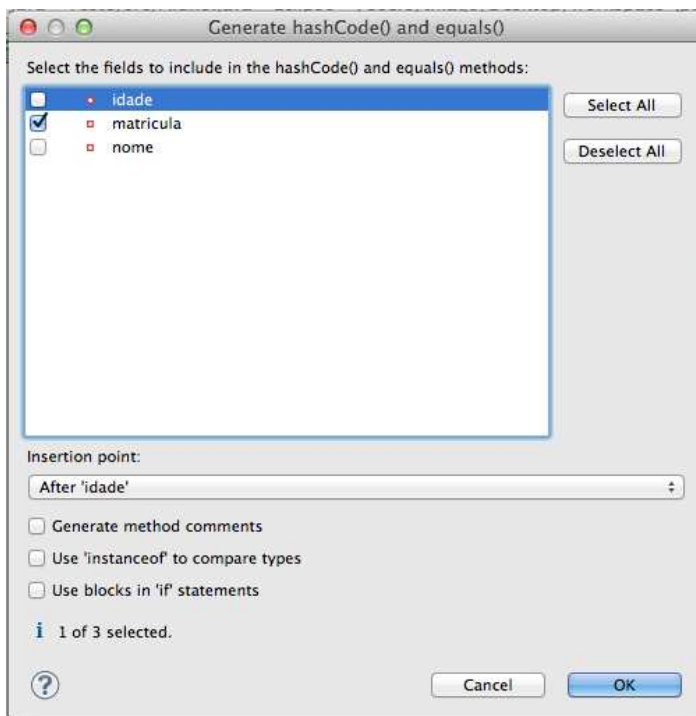
```

    if (matricula == null) {
        if (other.matricula != null)
            return false;
    } else if (!matricula.equals(other.matricula))
        return false;
    return true;
}

```

A boa notícia é que você não precisa digitar todo esse código, o Eclipse consegue gerar para você. Encontre o menu do Eclipse **Source** e clique em **Generate hashCode() and equals()**.

Selecione apenas a matrícula (veja imagem abaixo), pois apenas com ela, conseguimos identificar o aluno. Se fosse algum outro objeto que precisássemos de mais de um atributo para identificá-lo, selecionaríamos todos os necessários. Será gerado também o método hashCode, além do equals, que explicaremos mais adiante.



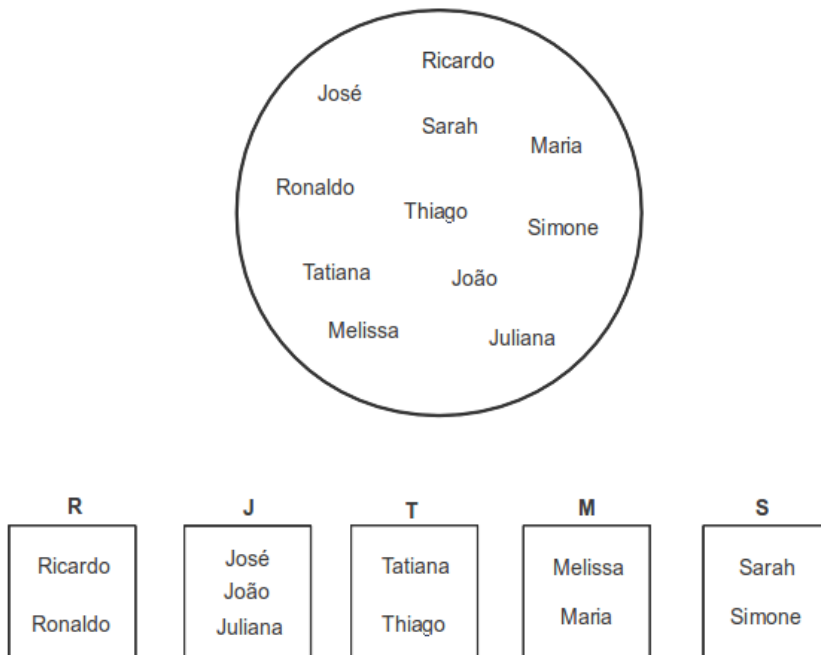
Se executarmos o código novamente, veremos o resultado que gostaríamos, já que ensinamos ao Java como comparar dois alunos. Veja a saída abaixo:

Parabéns João Costa, você ganhou um brinde especial!

Fica a dica: é através do método equals que um HashSet irá comparar os objetos para descobrir se existe ou não elementos repetidos.

O método `hashCode`, por sua vez, também é importante para facilitar e agilizar o acesso em coleções que utilizam *hash*, ou seja, um código que é usado para organizar a coleção em grupos menores, executando pesquisas mais rápidas.

Por exemplo, na imagem abaixo, existem vários nomes em um conjunto. Se pesquisarmos o nome "Tatiana", teremos que olhar um por um até encontrar, a não ser que separemos em subconjuntos menores, para facilitar a pesquisa. Essa é a função do *hash*.



Por isso, o método `hashCode` é implementado usando o identificador do objeto, assim teremos muitos códigos *hash* diferentes, agrupando os objetos usando alguma lógica, e encontrar cada objeto será muito mais rápido. Veja abaixo a implementação gerada pelo Eclipse.

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
        + ((matricula == null) ? 0 : matricula.hashCode());
    return result;
}
```

## 15.5. Mapas - Interface Map

Com a interface `java.util.Map`, conseguimos montar estruturas onde uma chave mapeia para um valor, por exemplo, através da matrícula, mapeamos um aluno. Outro detalhe importante é que não podemos ter chaves duplicadas, neste exemplo, não podemos ter duas matrículas iguais, o que faz muito sentido.

Vamos criar um exemplo usando a implementação da classe `java.util.HashMap`. Neste primeiro exemplo a matrícula é do tipo `Integer` e o aluno, a classe criada na seção anterior. No código abaixo, estamos instanciando o mapa e adicionando os alunos.

```
Map<Integer, Aluno> alunos = new HashMap<>();

alunos.put(1111, new Aluno(1111, "João", 20));
alunos.put(1112, new Aluno(1112, "Maria", 21));
alunos.put(1113, new Aluno(1113, "Ricardo", 20));
```

O método `put` recebe dois parâmetros, a chave e o valor. Veja a assinatura:

```
public V put(K key, V value)
```

Como ele também utiliza *generics*, recebe "K" para chave, que neste exemplo é um `Integer` e "V" como valor, que aqui é do tipo `Aluno`. Repare que este método retorna "V", que é o valor que estava armazenado antes para essa chave. Neste caso será retornado `null`, pois não havia nada no mapa para a chave informada.

Para recuperar um objeto a partir da chave, utilizamos o método `get`.

```
public V get(Object key)
```

Ele recebe a chave como argumento e retorna o valor. Continuando a implementação do exemplo anterior, veja o código abaixo.

```
// instancia e insere alunos no mapa aqui...

Aluno a1 = alunos.get(1111);
System.out.println("Nome: " + a1.getNome());
```

Irá imprimir:

Nome: João

Se a chave não for encontrada, será retornado `null`. Por isso, tome cuidado e faça validações antes de utilizar o objeto retornado do mapa.

Para percorrer os elementos do mapa, podemos usar `keySet`, que retorna um objeto do tipo `Set`, com todas as chaves do mapa, ou usamos `values`, que retorna uma coleção com todos os valores. Veja o exemplo abaixo.

```
Set<Integer> matriculas = alunos.keySet();

for (Integer matricula : matriculas) {
    System.out.println("Matricula: " + matricula + ". Aluno: "
        + alunos.get(matricula).getNome());
}
```

A saída desse trecho é:

```
Matricula: 1113. Aluno: Ricardo
Matricula: 1112. Aluno: Maria
Matricula: 1111. Aluno: João
```

Veja que na impressão do nome do aluno, colocamos vários métodos em cascata, mas poderíamos deixar mais organizado, veja abaixo a mesma funcionalidade, escrita de uma maneira diferente.

```
Set<Integer> matriculas = alunos.keySet();

for (Integer matricula : matriculas) {
    Aluno a = alunos.get(matricula);
    String nome = a.getNome();
    System.out.println("Matricula: " + matricula + ". Aluno: "
        + nome);
}
```

Se quisermos percorrer apenas os valores, e não nos importar com a chave, podemos utilizar o método `values`. Veja o código abaixo, que imprime o nome e a idade do aluno.

```
Collection<Aluno> todosAlunos = alunos.values();

for (Aluno aluno : todosAlunos) {
    System.out.println("Nome: " + aluno.getNome()
        + ". Idade: " + aluno.getIdade());
}
```

Saída:

```
Nome: Ricardo. Idade: 20
Nome: Maria. Idade: 21
Nome: João. Idade: 20
```

Repare que, neste caso, `values` retornou uma `Collection`, e não `Set`, como no caso anterior com as chaves, o que deixa mais claro que chaves não podem repetir, mas os valores sim. Poderíamos ter o mesmo aluno com duas matrículas diferentes, o que pode fazer sentido em várias situações.

## 15.6. Conjunto ordenado - Interface `SortedSet`

Um `java.util.SortedSet` será bem fácil de entender, pois ele é um `Set` ordenado. Isso quer dizer que se o tipo for numérico, a ordem será crescente, se for `String`, em ordem alfabética, e se for um `Date`, em ordem cronológica.

Vamos criar uma lista de nomes, inserir de maneira aleatória e no momento da impressão, verificar que a ordem alfabética foi seguida. A implementação que vamos usar é a classe `java.util.TreeSet`. Veja o exemplo abaixo.

```
SortedSet<String> nomesOrdenados = new TreeSet<>();

nomesOrdenados.add("Ricardo");
nomesOrdenados.add("João");
nomesOrdenados.add("Ana");
nomesOrdenados.add("Fabrício");

for (String nome : nomesOrdenados) {
    System.out.println(nome);
}
```

Como combinado, a saída será impressa em ordem alfabética. Veja:

```
Ana
Fabrício
João
Ricardo
```

Na verdade, quem implementa a ordenação é a classe `TreeSet`, e ela também implementa a interface `Set`, ou seja, poderíamos declarar o conjunto como no código abaixo, que o resultado seria o mesmo.

```
Set<String> nomesOrdenados = new TreeSet<>();
```

A diferença é que a interface `SortedSet` adiciona alguns métodos a mais que `Set` não tem. Como por exemplo:

```
public E first();
public E last();
```

Esses métodos irão retornar o primeiro e o último elemento na coleção ordenada.

```
SortedSet<String> nomesOrdenados = new TreeSet<>();

nomesOrdenados.add("Ricardo");
nomesOrdenados.add("João");
nomesOrdenados.add("Ana");
nomesOrdenados.add("Fabrício");

String primeiroNome = nomesOrdenados.first();
String ultimoNome = nomesOrdenados.last();

System.out.println("Primeiro: " + primeiroNome);
System.out.println("Último: " + ultimoNome);
```

Saída:

```
Primeiro: Ana
Último: Ricardo
```

E se quisermos adicionar um objeto Aluno? Como deve ser a ordenação? A resposta é simples, fazer Aluno implementar a interface Comparable e codificar o método compareTo.

Quando implementamos essa interface, temos que informar qual é a classe que queremos fazer a comparação, neste caso, Aluno. Veja o código abaixo.

```
public class Aluno implements Comparable<Aluno> {
    // ...
}
```

Quando implementamos Comparable, somos obrigados a implementar apenas um método, o compareTo. Na implementação dele, iremos dizer como que dois alunos são comparados para fins de **ordenação**. Analise o código abaixo e veja a explicação de como devemos implementá-lo.

```
@Override
public int compareTo(Aluno o) {
    int resultado = this.nome.compareTo(o.getNome());

    // se os nomes são iguais, podemos ordenar pela matrícula
    if (resultado == 0) {
        resultado = this.matricula > o.getMatricula() ? 1 : -1;
    }

    return resultado;
}
```

Esse método recebe um aluno como parâmetro para comparar com o objeto atual. A ideia é ordenar pelo nome e, caso os nomes sejam iguais, ordenar pelo número de matrícula.

Retornarmos um inteiro menor que zero (normalmente usamos -1) se o objeto atual for menor que o objeto passado, 0 se eles forem iguais e igual ou maior que 1 se ele for maior.

Um objeto "menor" que outro significa que ele deve ser inserido antes do outro na coleção, e um objeto "maior", que sua posição é depois do outro, e zero não faz diferença.

Com a classe Aluno usando essa implementação, se executarmos o código abaixo, veremos a saída ordenada.

```
SortedSet<Aluno> alunos = new TreeSet<>();

alunos.add(new Aluno(1111, "João Costa", 20));
alunos.add(new Aluno(1112, "Maria Rita", 21));
alunos.add(new Aluno(1113, "José de Castro", 22));
alunos.add(new Aluno(1114, "Ricardo Caetano", 20));
alunos.add(new Aluno(1115, "Maria Rita", 23));

for (Aluno aluno : alunos) {
    System.out.println("Nome: " + aluno.getNome()
        + ". Matrícula: " + aluno.getMatricula());
}
```

Saída:

```
Nome: José de Castro. Matrícula: 1113
Nome: João Costa. Matrícula: 1111
Nome: Maria Rita. Matrícula: 1112
Nome: Maria Rita. Matrícula: 1115
Nome: Ricardo Caetano. Matrícula: 1114
```

## 15.7. Mapa ordenado - Interface SortedMap

O SortedMap é um Map com as chaves ordenadas. A classe que podemos utilizar como implementação é a `java.util.TreeMap`.

Assim como vimos em SortedSet, temos alguns métodos úteis em SortedMap, devido a ordenação. Veja a assinatura deles abaixo:



```

public K firstKey()
public K lastKey()
public SortedMap<K,V> headMap(K toKey)
public SortedMap<K,V> tailMap(K fromKey)

```

Os dois primeiros são bem simples, retornarm a primeira e a última chave, respectivamente. Vamos ver um exemplo com os dois últimos e entender como eles funcionam.

```

SortedMap<Integer, Aluno> alunos = new TreeMap<>();

alunos.put(1111, new Aluno(1111, "João", 20));
alunos.put(1112, new Aluno(1112, "Maria", 21));
alunos.put(1113, new Aluno(1113, "Ricardo", 20));
alunos.put(1114, new Aluno(1114, "José", 22));
alunos.put(1115, new Aluno(1115, "Sarah", 23));
alunos.put(1116, new Aluno(1116, "Thiago", 21));
alunos.put(1117, new Aluno(1117, "Mateus", 19));
alunos.put(1118, new Aluno(1118, "Eduardo", 23));

System.out.println("Primeiros três alunos:");
SortedMap<Integer, Aluno> primeirosTresAlunos = alunos.headMap(1114);

for (Aluno aluno : primeirosTresAlunos.values()) {
    System.out.println("Matrícula: " + aluno.getMatricula()
        + ". Nome: " + aluno.getNome());
}

System.out.println("\nÚltimos três alunos:");
SortedMap<Integer, Aluno> ultimosTresAlunos = alunos.tailMap(1116);

for (Aluno aluno : ultimosTresAlunos.values()) {
    System.out.println("Matrícula: " + aluno.getMatricula()
        + ". Nome: " + aluno.getNome());
}

```

Veja a saída abaixo e logo depois a explicação do código.

```

Primeiros três alunos:
Matrícula: 1111. Nome: João
Matrícula: 1112. Nome: Maria
Matrícula: 1113. Nome: Ricardo

```

```

Últimos três alunos:
Matrícula: 1116. Nome: Thiago
Matrícula: 1117. Nome: Mateus
Matrícula: 1118. Nome: Eduardo

```

O método `headMap` retorna um novo `SortedMap` com elementos da primeira chave até a chave informada no argumento, mas sem incluí-la, por isso a impressão foi de 1111 à 1113, mesmo informando a chave 1114.

Já o método `tailMap` retorna um novo `SortedMap`, porém iniciando da chave informada até o fim do objeto.

Nesse exemplo usamos `Integer` como tipo da chave, mas se uma classe do tipo `Aluno` fosse usada, a ordenação também seria feita usando o método `compareTo`.

## 15.8. Algoritmos - Classe Collections

A classe `java.util.Collections` tem vários métodos estáticos úteis para se trabalhar com coleções, a maioria dos métodos utiliza `List` como argumento.

Vamos descrever alguns destes métodos nessa seção, mas para mais informações, consulte a documentação da classe em <http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>.

### Ordenação

Existirão situações que teremos um `ArrayList`, e gostaríamos que ele fosse ordenado por ordem alfabética, ou pela ordem definida no nosso objeto, como é o caso da classe `Aluno`. Se não existisse a classe `Collections`, teríamos que implementar esse algoritmo. Veja o exemplo abaixo.

```
List<String> nomes = new ArrayList<>();

nomes.add("Pedro");
nomes.add("José");
nomes.add("Maria");
nomes.add("Ana");

System.out.println("Antes de ordenar:");

// ArrayList mantém a ordem de inserção dos elementos
for (String nome : nomes) {
    System.out.println(nome);
}

Collections.sort(nomes);

System.out.println("\nDepois de ordenar:");

// agora a lista foi ordenada
for (String nome : nomes) {
    System.out.println(nome);
}
```

Veja que a saída foi ordenada na segunda iteração:

Antes de ordenar:

Pedro  
José  
Maria  
Ana

Depois de ordenar:

Ana  
José  
Maria  
Pedro

Para ordenar em ordem decrescente, existe um passo a mais. Cuidado e atenção, não é o método `reverse` que usaremos, continua sendo o `sort`. Veja o código abaixo.

```
List<String> nomes = new ArrayList<>();

nomes.add("Pedro");
nomes.add("José");
nomes.add("Maria");
nomes.add("Ana");

System.out.println("Antes de ordenar:");

// ArrayList mantém a ordem de inserção dos elementos
for (String nome : nomes) {
    System.out.println(nome);
}

Comparator<String> reverseComparator = Collections.reverseOrder();
Collections.sort(nomes, reverseComparator);

System.out.println("\nDepois de ordenar decrescente:");

// agora a lista foi ordenada
for (String nome : nomes) {
    System.out.println(nome);
}
```

Precisamos primeiro construir um objeto do tipo `Comparator` com o mesmo tipo genérico da lista que queremos ordenar, neste caso, `String`. Depois usamos o método `sort`, sendo que o primeiro argumento é a lista a ser ordenada e o segundo o comparador. Veja a saída abaixo.

Antes de ordenar:

Pedro  
José  
Maria

Ana

Depois de ordenar decrescente:

Pedro

Maria

José

Ana

## Método reverse

O método reverse simplesmente altera a ordem dos elementos, trazendo os últimos para as primeiras posições. Ele não faz ordenação alfabética decrescente, mas inverte os elementos na lista. Veja o exemplo abaixo:

```
List<String> nomes = new ArrayList<>();

nomes.add("Pedro");
nomes.add("José");
nomes.add("Maria");
nomes.add("Ana");

System.out.println("Antes de inverter:");

// ArrayList mantém a ordem de inserção dos elementos
for (String nome : nomes) {
    System.out.println(nome);
}

Collections.reverse(nomes);

System.out.println("\nDepois de inverter:");

// agora a lista foi alterada
for (String nome : nomes) {
    System.out.println(nome);
}
```

Repare que o último elemento agora é o primeiro, o penúltimo o segundo e assim sucessivamente. Veja a saída abaixo:

Antes de inverter:

Pedro

José

Maria

Ana

Depois de inverter:

Ana

Maria

## Bloqueando modificações

Vamos pensar em um exemplo para entender quando é necessário ter uma lista sem modificações. Suponha um sistema de cursos que, depois de iniciado, não é possível adicionar ou remover alunos. Na implementação de `ArrayList`, podemos sempre que quisermos, adicionar ou remover elementos. Mas, através do método `unmodifiableList`, conseguimos criar uma cópia que não se pode alterar, veja o exemplo abaixo.

```
List<String> nomes = new ArrayList<>();

nomes.add("Pedro");
nomes.add("José");
nomes.add("Maria");
nomes.add("Ana");

// Cópia da lista "nomes", mas essa não conseguiremos alterar
List<String> nomesNaoModificaveis = Collections
    .unmodifiableList(nomes);

// Não permitirá adicionar ou remover elementos da lista
nomesNaoModificaveis.add("Ricardo");
```

Como `nomesNaoModificaveis` é uma lista não modificável, chamar qualquer método que tente alterá-la, irá ser um problema e uma exceção será lançada, veja a saída abaixo.

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(
Collections.java:1075)
    at com.algaworks.Teste.main(Teste.java:21)
```

# Entrada e saída

## 16.1. O que é I/O Streams?

*I/O* é a abreviação em inglês de *Input* e *Output*, que significam *Entrada* e *Saída*, respectivamente. *Entrada* significa tudo que o programa que estamos desenvolvendo possa ler ou receber, seja de um arquivo, da rede, de outros programas, etc.



Já *sSaída*, é o que nosso software estiver escrevendo ou produzindo para estes *streams*, ou seja, toda fonte ou destino dos dados.



Em Java existe uma vasta API para se trabalhar com entrada e saída. Você deve se lembrar que, para ler de uma fonte de dados, utiliza-se *input stream*, e para escrever, *output stream*. Na próxima seção, iremos aos exemplos.

## 16.2. Gravando arquivos

Para demonstrar como gravar arquivos no disco, vamos criar uma lista de e-mails.

A forma mais eficiente de se escrever em arquivos ou mesmo enviar dados através da rede, é ir armazenando em uma área da memória, e quando essa área estiver cheia, completamos a operação. Essa área é chamada de *buffer*.

Por isso, vamos usar a classe `java.io.BufferedWriter` para escrever nossa lista de e-mails em um arquivo. Veja o código abaixo.

```
String[] emails = { "joao@joao.com", "maria@maria.com",  
    "ricardo@ricardo.com" };  
  
BufferedWriter outputStream = null;  
  
try {  
    outputStream = new BufferedWriter(new FileWriter("emails.txt"));  
  
    for (String email : emails) {  
        outputStream.write(email);  
        outputStream.newLine();  
    }  
} catch (IOException e) {  
    System.err.println("Não conseguiu gravar o arquivo. Erro: "  
        + e.getMessage());  
    System.exit(1);  
} finally {  
    try {  
        outputStream.close();  
    } catch (IOException e) { }  
}
```

Criamos um array de `String` com nossa lista de e-mails. Logo abaixo, instanciamos o `outputStream` com as classes `BufferedWriter` e `FileWriter`. A primeira delas é nosso *buffer stream*, com ela que conseguimos escrever dados para um destino, não importa qual é o destino, queremos apenas escrever dados na saída.

A classe `FileWriter` que determina que o *stream* seja direcionado para um arquivo de texto local no computador. Seu construtor recebe uma `String` com o nome do arquivo. Como não informamos o caminho completo, esse arquivo será gerado na raiz do projeto, mas poderíamos dizer, por exemplo, para gravar em outro diretório, veja abaixo:

```
// No Linux ou Mac  
outputStream = new BufferedWriter(new FileWriter("/tmp/emails.txt"));
```

```
// No Windows
outputStream = new BufferedWriter(new FileWriter("C:\\emails.txt"));
```

O trecho abaixo é fácil de entender:

```
for (String email : emails) {
    outputStream.write(email);
    outputStream.newLine();
}
```

Para cada e-mail no array, estamos escrevendo no *buffer* através do método `write`. Já o método `newLine` cria uma nova linha, para que cada e-mail fique em uma linha separada.

Repare que fomos obrigados a capturar `IOException`, pois `FileWriter` pode lançar caso não consiga criar o arquivo.

E por fim, fechamos o `BufferedWriter`, para não deixar nenhum recurso aberto, o que deixa nosso código feio, não agradável de ler, veja o `finally` abaixo:

```
finally {
    try {
        outputStream.close();
    } catch (IOException e) { }
}
```

A boa notícia é que a partir do Java 7, foi criada uma nova sintaxe, chamada de *try-with-resources*. Assim, vários recursos que precisam ser fechados e que implementem a interface `AutoCloseable` podem usar esse recurso e deixar o código mais limpo. Veja a nova versão do código para Java 7.

```
String[] emails = { "joao@joao.com", "maria@maria.com",
    "ricardo@ricardo.com" };

try (BufferedWriter outputStream
    = new BufferedWriter(new FileWriter("emails.txt"))) {

    for (String email : emails) {
        outputStream.write(email);
        outputStream.newLine();
    }
} catch (IOException e) {
    System.err.println("Não conseguiu gravar o arquivo. Erro: "
        + e.getMessage());
    System.exit(1);
}
```

Se você executar esse código mais de uma vez, verá que o arquivo é sobrescrito sempre, ou seja, não conseguimos adicionar apenas um e-mail no fim. Isso porque o



comportamento padrão de `FileWriter` sempre irá criar um novo arquivo. Para que seja possível incluir novos emails sem a necessidade de apagar o arquivo atual, no construtor, coloque `true` como segundo argumento. Veja abaixo:

```
try (BufferedWriter outputStream
    = new BufferedWriter(new FileWriter("emails.txt", true))) {
```

Outra classe que podemos utilizar para escrever arquivos é a `java.io.PrintStream`. Com ela, temos vários métodos úteis, que facilitam a escrita com strings. Veja o exemplo abaixo:

```
String[] emails = { "joao@joao.com", "maria@maria.com",
    "ricardo@ricardo.com" };

try (PrintStream outputStream = new PrintStream("emails.txt")) {
    for (String email : emails) {
        outputStream.append(email);
        outputStream.println();
    }

} catch (FileNotFoundException e) {
    System.err.println("Não conseguiu criar o arquivo. Erro: "
        + e.getMessage());
    System.exit(1);
}
```

Repare que usamos o método `append` para adicionar a `String` ao arquivo e `println` para criar uma nova linha.

Veja mais detalhes da classe em <http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html>.

## 16.3. Lendo arquivos

Agora que já temos o arquivo salvo no computador, podemos aprender como ler esse arquivo. Para leitura, também é bom usarmos um *buffer*, assim como na escrita. Isso porque é lido uma quantidade maior de dados de uma vez no arquivo e depois entregue ao nosso programa.

As classes que utilizaremos para leitura são `BufferedReader` e `FileReader`. Veja o exemplo abaixo.

```
try (BufferedReader inputStream
    = new BufferedReader(new FileReader("emails.txt"))) {

    String email = inputStream.readLine();
```

```

    while (email != null) {
        System.out.println(email);
        email = inputStream.readLine();
    }

} catch (IOException e) {
    System.err.println("Não conseguiu ler o arquivo. Erro: "
        + e.getMessage());
    System.exit(1);
}

```

Repare o uso do método `readLine`, que retorna uma `String` para cada linha do arquivo ou `null`, quando terminar, ou seja, imagine um ponteiro em cada linha do arquivo, que vai descendo até terminá-lo.

Uma outra forma mais enxuta de escrever esse `while` é ler a linha do *buffer* e verificar se está diferente de `null` em um trecho de código único. Veja essa versão abaixo:

```

String email = null;
while ((email = inputStream.readLine()) != null) {
    System.out.println(email);
}

```

## 16.4. A classe Scanner

A classe `java.util.Scanner` é útil para ler entradas que seguem padrões, quebrando em *tokens* de acordo com o tipo de formatação.

Por exemplo, vamos criar um programa que processa um arquivo de itens de pedido. Iremos gerar na saída o nome do produto, a quantidade e o valor do item.

O arquivo segue um padrão, em cada linha temos as informações de um produto, separadas por ";", sendo que o primeiro *token* é a descrição, o segundo a quantidade e o terceiro o valor do produto. Veja abaixo o conteúdo do arquivo *itens\_pedido.txt*, que será processado no nosso programa.

```

Arroz;3;9,8;
Feijão;2;6,3;
Extrato de tomate;1;3,25;
Óleo de Soja;2;4;

```

Podemos observar que a descrição é uma `String`, a quantidade um inteiro e o valor um `double`, formatado para o Brasil, pois o separador decimal é a vírgula. Por isso, vamos instanciar o `Locale` para o Brasil.

```
Locale localeBrasil = new Locale("pt", "BR");
```

Para instanciar o Scanner, passaremos um leitor no seu construtor. Vamos utilizar o `BufferedReader`, que também recebe um `FileReader`, que sabe ler o conteúdo do arquivo.

```
try (Scanner scanner = new Scanner(  
    new BufferedReader(new FileReader("itens_pedido.txt")))) {
```

O Scanner também precisa ser fechado quando não formos mais utilizá-lo. Usando *try-with-resources* do Java 7, conseguimos fechá-lo juntamente com o `BufferedReader` e o `FileReader`.

Agora podemos configurar o *locale* e também o delimitador. Este último é o que separa cada *token* no arquivo. O padrão para o Scanner é um espaço em branco, mas como no nosso arquivo é um ponto-e-vírgula (;), vamos alterá-lo, usando o método `useDelimiter`.

```
scanner.useLocale(localeBrasil);  
scanner.useDelimiter(";");
```

A classe Scanner possui vários métodos para ler os diversos tipos de dados, como `String`, `int`, `double`, etc. O método `next()` retorna `true` se ainda existirem *tokens* para serem lidos.

Como no nosso arquivo temos um produto por linha, significa que no fim do arquivo existe uma *quebra de linha*. Podemos utilizar o método `nextLine()` para ler essa quebra e passar para a próxima entrada.

Veja abaixo o código completo do nosso programa:

```
Locale localeBrasil = new Locale("pt", "BR");  
  
try (Scanner scanner = new Scanner(  
    new BufferedReader(new FileReader("itens_pedido.txt")))) {  
  
    scanner.useLocale(localeBrasil);  
    scanner.useDelimiter(";");  
  
    NumberFormat formatador = NumberFormat.getCurrencyInstance(  
        localeBrasil);  
  
    while (scanner.hasNext()) {  
        String produto = scanner.next();  
        int quantidade = scanner.nextInt();  
        double valor = scanner.nextDouble();  
        scanner.nextLine();
```

```

        System.out.printf("Produto: %s. Quantidade: %d. Por: %s\n",
            produto, quantidade, formatador.format(valor));
    }

} catch (IOException e) {
    System.err.println("Não conseguiu ler o arquivo. Erro: "
        + e.getMessage());
    System.exit(1);
}

```

Usamos um `NumberFormat` para formatar o valor com a moeda do Brasil e o `printf` para deixar mais elegante nossa saída. Veja o resultado abaixo.

```

Produto: Arroz. Quantidade: 3. Por: R$ 9,80
Produto: Feijão. Quantidade: 2. Por: R$ 6,30
Produto: Extrato de tomate. Quantidade: 1. Por: R$ 3,25
Produto: Óleo de Soja. Quantidade: 2. Por: R$ 4,00

```

# Arquivos JAR e documentação com javadoc

## 17.1. O que são os arquivos JAR?

Os arquivos JAR (*Java ARchive*) são uma forma de empacotar vários arquivos em Java, assim como um *ZIP*. Normalmente são armazenadas as classes e outros arquivos de configuração.

Como eles agrupam várias classes em um único arquivo, são ótimos para distribuir bibliotecas, como drivers de banco de dados, frameworks, módulos de sistemas, etc.

Podemos também criar um *JAR* executável, que contém uma classe com o método `main`, e usando a JVM, conseguimos executá-lo.

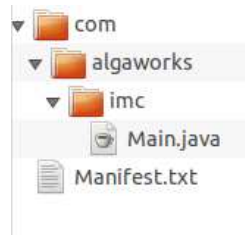
Para gerar um arquivo JAR, visualizar ou extrair o conteúdo do arquivo, usamos o programa de mesmo nome, o `jar`, que é distribuído com o Java Development Kit (JDK).

## 17.2. Gerando arquivos JAR executáveis

Vamos criar um programa que recebe do usuário seu peso e altura, calcula o IMC e imprime na tela o resultado. Assim conseguiremos ver o processo de criar e executar um JAR executável.

Não vamos utilizar nenhuma IDE para auxiliar esse procedimento, para demonstrar como utilizar a ferramenta.

Criaremos uma estrutura como a imagem abaixo. Uma classe `Main` no pacote `com.algaworks.imc` e um arquivo `Manifest.txt` na raiz.



A classe deve ter o código abaixo no método `main`.

```
try (Scanner entrada = new Scanner(System.in)) {
    entrada.useLocale(new Locale("pt", "BR"));

    System.out.print("Digite seu peso: ");
    double peso = entrada.nextDouble();

    System.out.print("Digite sua altura: ");
    double altura = entrada.nextDouble();

    double imc = peso / (altura * altura);

    System.out.printf("IMC: %.2f\n", imc);
}
```

Vamos compilar a classe usando o `javac`.

```
$ javac com/algaworks/imc/Main.java
```

O arquivo `Manifest.txt` deve conter o nome da classe que possui o método `main`. Veja o conteúdo abaixo:

```
Main-Class: com.algaworks.imc.Main
```

Agora é hora de gerar o arquivo JAR. Vamos executar o comando abaixo.

```
$ jar cfm imc.jar Manifest.txt com/algaworks/imc/Main.class
```

O comando está dizendo para criar um arquivo `imc.jar`, com a classe `Main` e com um arquivo `MANIFEST.MF`, no diretório `META-INF`.

Este arquivo `MANIFEST.MF` é lido no momento da execução para descobrir qual a classe possui o método `main`, além de conter outras informações, como a versão do JDK usado para sua criação. Veja o conteúdo dele abaixo:

```
Manifest-Version: 1.0
Created-By: 1.7.0_25 (Oracle Corporation)
Main-Class: com.algaworks.imc.Main
```

Para executar o JAR, basta usarmos o comando `java` com a opção `-jar`, veja a execução completa abaixo:

```
$ java -jar imc.jar
Digite seu peso: 78,4
Digite sua altura: 1,81
IMC: 23.93
```

### 17.3. Gerando arquivos JAR como bibliotecas

Agora que já vimos como criar um JAR executável, fica mais fácil criar um para ser apenas uma biblioteca. A principal diferença é não ter a propriedade "Main-Class" no *MANIFEST.MF*.

Criamos um JAR para ser uma biblioteca quando queremos distribuir um código para ser usado em programas de terceiros. No dia a dia, usamos várias bibliotecas em nosso código, por exemplo para acessar o banco de dados.

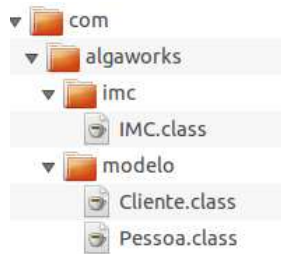
O comando para gerar o arquivo JAR é muito simples, veja abaixo sua sintaxe:

```
jar cf nome-do-arquivo.jar arquivo(s)
```

O *nome-do-arquivo.jar* é o nome do arquivo JAR que será gerado, e logo em seguida, os arquivos que queremos empacotar.

Podemos usar `*` para todos os arquivos, inclusive diretórios, que serão incluídos recursivamente.

Vamos a um exemplo simples. Suponha que temos um projeto com várias classes, em vários pacotes diferentes, que podem ser distribuídas para outros desenvolvedores utilizarem. Algo parecido com a estrutura abaixo:



Para gerar o arquivo chamado *utilitario.jar*, basta usar o comando abaixo:

```
$ jar cf utilitarios.jar *
```

Um comando útil que pode ser usado, principalmente quando estamos em servidores que não tem interface gráfica, é o comando `jar` com as opções `tf`, que lista o conteúdo do arquivo. Veja abaixo:

```
$ jar tf utilitarios.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/algaworks/
com/algaworks/modelo/
com/algaworks/modelo/Pessoa.class
com/algaworks/modelo/Cliente.class
com/algaworks/imc/
com/algaworks/imc/IMC.class
```

## 17.4. Importando arquivos JAR no projeto

Já que aprendemos a gerar arquivos JAR e dissemos que é muito comum utilizarmos várias bibliotecas de terceiros, vamos aprender a importá-las em nosso projeto.

As bibliotecas importadas devem ser inseridas dentro do *classpath* da aplicação. As IDEs facilitam muito esse procedimento, mas vamos entender como adicionar um JAR pela linha de comando.

Para isso, vamos utilizar o *Apache HttpComponents*, uma biblioteca muito útil para se trabalhar com HTTP. Nosso exemplo simplesmente irá fazer uma requisição à página do Google.



Primeiro, vamos baixar o zip em <http://hc.apache.org/downloads.cgi>. Para esse exemplo, usaremos a versão 4.2.5.

Extraindo o arquivo no diretório *lib*, visualizamos vários arquivos JAR. Utilizaremos apenas 3 deles:

- httpcore-4.2.4.jar
- httpclient-4.2.5.jar
- commons-logging-1.1.1.jar

Vamos criar uma classe de teste que possui o seguinte código:

```
import java.io.IOException;

import org.apache.http.HttpEntity;
import org.apache.http.HttpHost;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.ResponseHandler;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.conn.params.ConnRoutePNames;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.util.EntityUtils;

public class Teste {

    public static void main(String[] args) throws Exception {
        DefaultHttpClient client = new DefaultHttpClient();

        HttpGet get = new HttpGet("http://www.google.com.br");

        ResponseHandler<byte[]> handler = new ResponseHandler<byte[]>() {
            public byte[] handleResponse(HttpResponse response)
                throws ClientProtocolException, IOException {
                HttpEntity entity = response.getEntity();
                if (entity != null) {
                    return EntityUtils.toByteArray(entity);
                } else {
                    return null;
                }
            }
        };

        String response = new String(client.execute(get, handler));
        System.out.println(response);
    }
}
```

Repare nos *imports*, que existem várias classes do Apache. Não é escopo dessa capítulo explicar sobre a API *HttpComponents*, mas o código acima simplesmente faz uma requisição HTTP ao Google e imprime o resultado na tela, o código-fonte HTML.

Para compilar e executar esse código, precisamos informar as bibliotecas dependentes na linha de comando. Supondo que os arquivos estejam em um diretório *lib*, execute o comando abaixo no Linux ou Mac:

```
$ javac -cp .:lib/httpcore-4.2.4.jar:lib/httpclient-4.2.5.jar\
:lib/commons-logging-1.1.1.jar Teste.java
```

E para executar:

```
$ java -cp .:lib/httpcore-4.2.4.jar:lib/httpclient-4.2.5.jar\
:lib/commons-logging-1.1.1.jar Teste
```

Você verá o código-fonte HTML da página do Google na tela.

No Windows, substitua o `:` por `;` e a `/` por `\`.

## 17.5. Gerando documentação com javadoc

É sempre bom criar documentação das suas classes, e com Java, isso se torna um pouco mais fácil de se fazer e distribuir.

Vamos criar duas classes simples para ilustrar como devemos escrever a documentação e depois como gerar usando o comando `javadoc`.

A classe `Calculadora` e a classe `Matematico` são apenas simples exemplos de como começamos a escrever a documentação. Veja abaixo o código das classes e repare a documentação escrita através de `/**` e `*/`. Isso não é apenas um comentário, é documentação *javadoc*!

Classe `Calculadora`:

```
/**
 * Realiza uma série de operações matemáticas.
 *
 * @author normandes
 */
public class Calculadora {

    /**
```

```

* Faz a soma entre os dois valores.
*
* @param a Primeiro termo da operação
* @param b Segundo termo da operação
* @return resultado Resultado da soma
*
* @throws IllegalArgumentException se "a" ou "b" for null
*/
public Double somar(Double a, Double b) {
    validar(a, b);
    return a + b;
}

/**
* Faz a subtração entre o minuendo e o subtraendo.
* a - b = diferença
*
* @param a Minuendo
* @param b Subtraendo
* @return diferença
*
* @throws IllegalArgumentException se "a" ou "b" for null
*/
public Double subtrair(Double a, Double b) {
    validar(a, b);
    return a - b;
}

/**
* Multiplica "a" por "b".
*
* @param a Operando
* @param b Operando
* @return produto
*
* @throws IllegalArgumentException se "a" ou "b" for null
*/
public Double multiplicar(Double a, Double b) {
    validar(a, b);
    return a * b;
}

/**
* Divide-se o divisor "a" pelo dividendo "b".
*
* @param a Divisor
* @param b Dividendo
* @return Quociente
*
* @throws IllegalArgumentException se "a" ou "b" for null
*/
public Double dividir(Double a, Double b) {
    validar(a, b);
    return a / b;
}

```

```

    }

    private void validar(Double a, Double b) {
        if (a == null) {
            throw new IllegalArgumentException("A variável \"a\" "
                + " não pode ser nulo");
        }

        if (b == null) {
            throw new IllegalArgumentException("A variável \"b\" "
                + " não pode ser nulo");
        }
    }
}

```

Classe Matematico:

```

import java.util.Locale;
import java.util.Scanner;

/**
 * Sabe como realizar soma, subtração, multiplicação e divisão.
 *
 * Recebe entrada do usuário e executa a operação matemática.
 *
 * @author normandes
 *
 */
public class Matematico {

    private Scanner entrada;
    private Calculadora calculadora;
    private Double a;
    private Double b;

    public Matematico(Scanner entrada, Calculadora calculadora) {
        this.entrada = entrada;
        this.calculadora = calculadora;
    }

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        entrada.useLocale(new Locale("pt", "BR"));
        Matematico matematico = new Matematico(entrada,
            new Calculadora());

        while (true) {
            System.out.print("Informe a operação (somar, subtrair, "
                + "multiplicar, dividir ou sair): ");
            String operacao = entrada.next();

            boolean sair = false;

```

```

        switch (operacao) {
            case "somar":
                matematico.somar();
                break;

            case "subtrair":
                matematico.subtrair();
                break;

            case "multiplicar":
                matematico.multiplicar();
                break;

            case "dividir":
                matematico.dividir();
                break;

            case "sair":
                System.out.println("Obrigado.");
                sair = true;
                break;
        }

        if (sair) {
            entrada.close();
            break;
        }
    }
}

/**
 * Solicita ao usuário digitar dois números.
 * Faz a divisão e mostra na tela o resultado.
 *
 * @see Calculadora
 */
public void dividir() {
    receberAeB();

    try {
        Double resultado = this.calculadora.dividir(a, b);
        System.out.printf("A divisão de %.2f e %.2f = %.2f\n",
            a, b, resultado);
    } catch (IllegalArgumentException e) {
        System.err.println("Erro ao realizar a divisão. "
            + e.getMessage());
    }
}

/**
 * Solicita ao usuário digitar dois números.
 * Faz a multiplicação e mostra na tela o resultado.
 *
 * @see Calculadora

```

```

*/
public void multiplicar() {
    receberAeB();

    try {
        Double resultado = this.calculadora.multiplicar(a, b);
        System.out.printf("A multiplicação de %.2f e %.2f = %.2f\n",
            a, b, resultado);
    } catch (IllegalArgumentException e) {
        System.err.println("Erro ao realizar a multiplicação. "
            + e.getMessage());
    }
}

/**
 * Solicita ao usuário digitar dois números.
 * Faz a subtração e mostra na tela o resultado.
 *
 * @see Calculadora
 */
public void subtrair() {
    receberAeB();

    try {
        Double resultado = this.calculadora.subtrair(a, b);
        System.out.printf("A subtração de %.2f e %.2f = %.2f\n",
            a, b, resultado);
    } catch (IllegalArgumentException e) {
        System.err.println("Erro ao realizar a subtração. "
            + e.getMessage());
    }
}

/**
 * Solicita ao usuário digitar dois números.
 * Faz a soma e mostra na tela o resultado.
 *
 * @see Calculadora
 */
public void somar() {
    receberAeB();

    try {
        Double resultado = this.calculadora.somar(a, b);
        System.out.printf("A soma de %.2f e %.2f = %.2f\n",
            a, b, resultado);
    } catch (IllegalArgumentException e) {
        System.err.println("Erro ao realizar a soma. "
            + e.getMessage());
    }
}

/**
 * Através do Scanner, recebe do usuário dois números "a" e "b"

```

```

    */
    private void receberAeB() {
        System.out.print("Digite o valor de \"a\": ");
        this.a = entrada.nextDouble();
        System.out.print("Digite o valor de \"b\": ");
        this.b = entrada.nextDouble();
    }
}

```

Veja que temos documentação escrita em classes e métodos, mas também poderíamos escrever em atributos e construtores.

Existem uma série de tags que fazem marcações importantes para a geração do HTML final.

A tag `@author` indica quem são os autores da classe, já `@see`, criará um link para a outra classe.

Nos métodos e construtores, podemos usar `@param` para descrever o parâmetro de entrada e `@return` para documentar sobre o retorno do método.

Por fim, `@throws` descreve as exceções que um método pode lançar.

Para gerar a documentação, usaremos o comando `javadoc`, que vem com o JDK. Coloque as duas classes em um diretório qualquer e execute o comando abaixo.

```
$ javadoc -author -d docs *.java
```

A opção `-author` irá incluir a tag `@author` na documentação, e o argumento `-d docs` define o diretório onde a documentação será gerada. Por fim, informamos os arquivos que queremos gerar, colocando `/*.java`, incluímos todos os arquivos Java.

Ao executar o comando, você verá uma saída como abaixo:

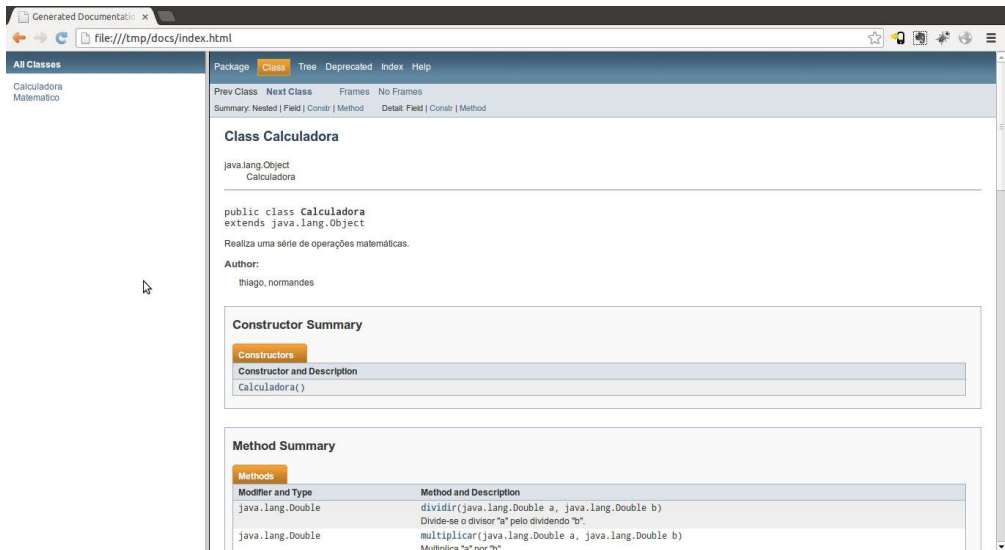
```

Creating destination directory: "docs/"
Loading source file Calculadora.java...
Loading source file Matematico.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_25
Building tree for all the packages and classes...
Generating docs/Calculadora.html...
Generating docs/Matematico.html...
Generating docs/package-frame.html...
Generating docs/package-summary.html...
Generating docs/package-tree.html...
Generating docs/constant-values.html...
Building index for all the packages and classes...

```

Generating docs/overview-tree.html...  
 Generating docs/index-all.html...  
 Generating docs/deprecated-list.html...  
 Building index for all classes...  
 Generating docs/allclasses-frame.html...  
 Generating docs/allclasses-noframe.html...  
 Generating docs/index.html...  
 Generating docs/help-doc.html...

Acesse o diretório *docs* e abra em seu navegador preferido o arquivo *index.html*. Você verá a documentação gerada como a da imagem abaixo.





# CURSOS ONLINE

## COM VÍDEO AULAS E SUPORTE

---



Java e Orientação a Objetos



Desenvolvimento Web  
com JSF 2



Persistência de Dados com  
JPA 2 e Hibernate



Sistemas Comerciais Java EE  
com CDI, JPA e PrimeFaces

[www.algaworks.com](http://www.algaworks.com)

Cursos presenciais in-company? Entre em contato.



# **JAVA E ORIENTAÇÃO A OBJETOS**

THIAGO FARIA E NORMANDES JR