



# Caelum

Ensino e Soluções em Java

# FJ-11

Java e Orientação a Objetos



[www.caelum.com.br](http://www.caelum.com.br)



# Caelum

*"Mata o tempo e matará a tua carreira"*

Bryan Forbes

## Sobre a empresa

A Caelum atua no mercado desde 2002, desenvolvendo sistemas e prestando consultoria em diversas áreas, sempre à luz da plataforma Java. Foi fundada por profissionais que se encontraram no Brasil após experiências na Alemanha e Itália, desenvolvendo sistemas de grande porte, com integração aos mais variados ERPs. Seus profissionais já publicaram diversos artigos nas revistas brasileiras sobre Java, artigos acadêmicos e são presença constante nos eventos de tecnologia.

Em 2004, a Caelum criou uma gama de cursos que rapidamente ganharam grande reconhecimento no mercado. Os cursos foram elaborados por ex-instrutores da Sun que queriam trazer mais dinamismo e aplicar as ferramentas e bibliotecas utilizadas no mercado, tais como Eclipse, Hibernate, Struts, e outras tecnologias *open source* que não são abordadas pela Sun. O material utilizado foi elaborado durante os cursos de verão de Java na Universidade de São Paulo, em janeiro de 2004, pelos instrutores da Caelum.

Em 2008, a empresa foca seus esforços em duas grandes áreas:

- Consultoria, mentoring e coaching de desenvolvimento
- Treinamento com intuito de formação

## Sobre a apostila

Esta apostila da Caelum visa ensinar Java de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material e que ele possa ser de grande valia para auto-didatas e estudantes. Todos os comentários, críticas e sugestões serão muito bem-vindos.

O material aqui contido pode ser publicamente distribuído, desde que seu conteúdo não seja alterado e que seus créditos sejam mantidos. Ele não pode ser usado para ministrar qualquer outro curso, porém pode ser utilizado como referência e material de apoio. Caso você esteja interessado em usá-lo para fins comerciais, entre em contato com a empresa.

**Atenção:** Você pode verificar a versão da apostila no fim do índice. Não recomendamos imprimir a apostila que você receber de um amigo ou pegar por e-mail, pois atualizamos constantemente esse material, quase que mensalmente. Vá até nosso site e veja a última versão.

---

# Índice

<b>1</b>	<b>Como Aprender Java</b>	<b>1</b>
1.1	O que é realmente importante? . . . . .	1
1.2	Sobre os exercícios . . . . .	2
1.3	Tirando dúvidas . . . . .	2
1.4	Bibliografia . . . . .	2
<b>2</b>	<b>O que é Java</b>	<b>3</b>
2.1	Java . . . . .	3
2.2	Máquina Virtual . . . . .	4
2.3	Java lento? Hotspot e JIT . . . . .	6
2.4	Versões do Java... e a confusão do Java2 . . . . .	6
2.5	JVM? JRE? JDK? . . . . .	7
2.6	Onde usar e os objetivos do Java . . . . .	7
2.7	Especificação versus implementação . . . . .	8
2.8	Como o FJ-11 está organizado . . . . .	8
2.9	Compilando o primeiro programa . . . . .	9
2.10	Executando seu primeiro programa . . . . .	10
2.11	O que aconteceu? . . . . .	10
2.12	E o bytecode? . . . . .	10
2.13	Exercícios: Modificando o Hello World . . . . .	11
2.14	O que pode dar errado? . . . . .	11
2.15	Um pouco mais... . . . . .	12
2.16	Exercícios adicionais . . . . .	13

<b>3</b>	<b>Variáveis primitivas e Controle de fluxo</b>	<b>14</b>
3.1	Declarando e usando variáveis . . . . .	14
3.2	Tipos primitivos e valores . . . . .	16
3.3	Exercícios: Variáveis e tipos primitivos . . . . .	17
3.4	Discussão em aula: convenções de código e código legível . . . . .	18
3.5	Casting e promoção . . . . .	18
3.6	O If-Else . . . . .	20
3.7	O While . . . . .	22
3.8	O For . . . . .	22
3.9	Controlando loops . . . . .	23
3.10	Escopo das variáveis . . . . .	24
3.11	Um bloco dentro do outro . . . . .	25
3.12	Um pouco mais... . . . .	25
3.13	Exercícios: Fixação de sintaxe . . . . .	26
3.14	Desafios: Fibonacci . . . . .	27
<b>4</b>	<b>Orientação a objetos básica</b>	<b>28</b>
4.1	Motivação: problemas do paradigma procedural . . . . .	28
4.2	Criando um tipo . . . . .	29
4.3	Uma classe em Java . . . . .	31
4.4	Criando e usando um objeto . . . . .	31
4.5	Métodos . . . . .	32
4.6	Métodos com retorno . . . . .	33
4.7	Objetos são acessados por referências . . . . .	35
4.8	O método transfere() . . . . .	37
4.9	Continuando com atributos . . . . .	39
4.10	Para saber mais: Uma Fábrica de Carros . . . . .	41
4.11	Um pouco mais... . . . .	43
4.12	Exercícios: Orientação a Objetos . . . . .	43
4.13	Desafios . . . . .	47
4.14	Fixando o conhecimento . . . . .	47

<b>5</b>	<b>Um pouco de arrays</b>	<b>49</b>
5.1	O problema . . . . .	49
5.2	Arrays de referências . . . . .	50
5.3	Percorrendo uma array . . . . .	51
5.4	Percorrendo uma array no Java 5.0 . . . . .	52
5.5	Exercícios: Arrays . . . . .	52
5.6	Um pouco mais... . . . .	54
5.7	Desafios . . . . .	55
5.8	Testando o conhecimento . . . . .	55
<b>6</b>	<b>Modificadores de acesso e atributos de classe</b>	<b>56</b>
6.1	Controlando o acesso . . . . .	56
6.2	Encapsulamento . . . . .	59
6.3	Getters e Setters . . . . .	61
6.4	Construtores . . . . .	62
6.5	A necessidade de um construtor . . . . .	64
6.6	Atributos de classe . . . . .	65
6.7	Um pouco mais... . . . .	67
6.8	Exercícios . . . . .	67
6.9	Desafios . . . . .	68
<b>7</b>	<b>Orientação a Objetos – herança, reescrita e polimorfismo</b>	<b>70</b>
7.1	Repetindo código? . . . . .	70
7.2	Reescrita de método . . . . .	73
7.3	Invocando o método reescrito . . . . .	74
7.4	Polimorfismo . . . . .	74
7.5	Um outro exemplo . . . . .	76
7.6	Um pouco mais... . . . .	77
7.7	Exercícios: Herança e Polimorfismo . . . . .	78
7.8	Dicussões em aula: Alternativas ao atributo protected . . . . .	82

<b>8 Eclipse IDE</b>	<b>83</b>
8.1 O Eclipse . . . . .	83
8.2 Apresentando o Eclipse . . . . .	83
8.3 Views e Perspective . . . . .	84
8.4 Criando um projeto novo . . . . .	86
8.5 Criando o main . . . . .	90
8.6 Rodando o main . . . . .	92
8.7 Pequenos truques . . . . .	93
8.8 Exercícios: Eclipse . . . . .	93
8.9 Discussão em aula: Refactorings . . . . .	95
<b>9 Orientação a Objetos – Classes Abstratas</b>	<b>96</b>
9.1 Repetindo mais código? . . . . .	96
9.2 Classe abstrata . . . . .	97
9.3 Métodos abstratos . . . . .	99
9.4 Aumentando o exemplo . . . . .	100
9.5 Para saber mais... . . . .	101
9.6 Exercícios: Classes Abstratas . . . . .	101
9.7 Desafios . . . . .	102
<b>10 Orientação a Objetos - Interfaces</b>	<b>103</b>
10.1 Aumentando nosso exemplo . . . . .	103
10.2 Interfaces . . . . .	107
10.3 Dificuldade no aprendizado de interfaces . . . . .	110
10.4 Exemplo interessante: conexões com o banco de dados . . . . .	110
10.5 Um pouco mais... . . . .	111
10.6 Exercícios: Interfaces . . . . .	111
10.7 Exercícios avançados . . . . .	114
10.8 Discussão em aula: Favorecer composição em relação à herança . . . . .	115

<b>11 Controlando os erros com Exceções</b>	<b>117</b>
11.1 Motivação . . . . .	117
11.2 Exercício para começar com os conceitos . . . . .	118
11.3 Exceções de Runtime mais comuns . . . . .	123
11.4 Outro tipo de exceção: Checked Exceptions . . . . .	124
11.5 Um pouco da grande família Throwable . . . . .	127
11.6 Mais de um erro . . . . .	127
11.7 Lançando exceções . . . . .	128
11.8 O que colocar dentro do try? . . . . .	129
11.9 Criando seu próprio tipo de exceção . . . . .	130
11.10 Para saber mais: finally . . . . .	132
11.11 Um pouco mais... . . . .	132
11.12 Exercícios: Exceções . . . . .	132
11.13 Desafios . . . . .	134
<b>12 Pacotes - Organizando suas classes e bibliotecas</b>	<b>135</b>
12.1 Organização . . . . .	135
12.2 Diretórios . . . . .	136
12.3 Import . . . . .	136
12.4 Acesso aos atributos, construtores e métodos . . . . .	138
12.5 Usando o Eclipse com pacotes . . . . .	139
12.6 Exercícios: Pacotes . . . . .	140
<b>13 Ferramentas: jar e javadoc</b>	<b>141</b>
13.1 Arquivos, bibliotecas e versões . . . . .	141
13.2 Gerando o jar pelo Eclipse . . . . .	142
13.3 Javadoc . . . . .	146
13.4 Gerando o Javadoc . . . . .	147
13.5 Exercícios: Jar e Javadoc . . . . .	151

<b>14 O pacote java.lang</b>	<b>152</b>
14.1 Pacote java.lang . . . . .	152
14.2 Um pouco sobre a classe System . . . . .	152
14.3 java.lang.Object . . . . .	153
14.4 Casting de referências . . . . .	153
14.5 Métodos do java.lang.Object: equals e toString . . . . .	155
14.6 Integer e classes wrappers (box) . . . . .	157
14.7 Autoboxing no Java 5.0 . . . . .	157
14.8 java.lang.String . . . . .	158
14.9 java.lang.Math . . . . .	160
14.10 Exercícios: java.lang . . . . .	161
14.11 Desafio . . . . .	162
<b>15 Pacote java.io</b>	<b>164</b>
15.1 Conhecendo uma API . . . . .	164
15.2 Orientação a objeto . . . . .	164
15.3 InputStream, InputStreamReader e BufferedReader . . . . .	165
15.4 Lendo Strings do teclado . . . . .	167
15.5 A analogia na saída: OutputStream, OutputStreamWriter e BufferedWriter . . . . .	168
15.6 Uma maneira mais fácil: Scanner e PrintStream . . . . .	168
15.7 Um pouco mais... . . . . .	169
15.8 Exercícios: Java I/O . . . . .	169
15.9 Discussão em aula: Design Patterns e o Template Method . . . . .	172
<b>16 Collections framework</b>	<b>174</b>
16.1 Arrays são trabalhosos, utilizar estrutura de dados . . . . .	174
16.2 Listas: java.util.List . . . . .	175
16.3 Listas no Java 5.0 com Generics . . . . .	178
16.4 Ordenação: Collections.sort . . . . .	179
16.5 Exercícios: Ordenação . . . . .	181
16.6 Conjunto: java.util.Set . . . . .	183
16.7 Principais interfaces: java.util.Collection . . . . .	185

16.8	Percorrendo coleções no Java 5	186
16.9	Para saber mais: Iterando sobre coleções com java.util.Iterator	187
16.10	Mapas - java.util.Map	189
16.11	Mapas no Java 5.0	191
16.12	Para saber mais: Properties	191
16.13	Para saber mais: Equals e hashCode	191
16.14	Para saber mais: Boas práticas	192
16.15	Exercícios: Collections	192
16.16	Desafios	194
<b>17</b>	<b>Programação Concorrente e Threads</b>	<b>195</b>
17.1	Threads	195
17.2	Escalonador e trocas de contexto	197
17.3	Garbage Collector	199
17.4	Para saber mais: problemas com concorrência	200
17.5	Para saber mais: Vector e Hashtable	202
17.6	Um pouco mais...	203
17.7	Exercícios: Threads	203
17.8	Desafios	204
<b>18</b>	<b>E agora?</b>	<b>205</b>
18.1	Exercício prático	205
18.2	Certificação	205
18.3	Web	205
18.4	Revistas	205
18.5	Grupos de Usuários	206
18.6	Falando em Java - Próximos módulos	206
<b>19</b>	<b>Apêndice A - Utilizando bibliotecas e frameworks: O JFreeChart</b>	<b>207</b>
19.1	Bibliotecas e o Java	207
19.2	Como usar uma biblioteca?	207
19.3	Classpath	208
19.4	JFreeChart	209

19.5	Preparando um projeto no Eclipse para utilizar o JFreeChart . . . . .	210
19.6	Utilizando o JFreeChart . . . . .	214
19.7	Aplicando as boas práticas de OO . . . . .	216
19.8	Para saber mais: Design Patterns Factory Method e Builder . . . . .	218
19.9	Exercícios: JFreeChart . . . . .	218
19.10	Exercícios adicionais . . . . .	220
19.11	Desafio: Fluent Interface . . . . .	220
<b>20</b>	<b>Apêndice B - Sockets</b>	<b>222</b>
20.1	Motivação: uma API que usa os conceitos aprendidos . . . . .	222
20.2	Protocolo . . . . .	222
20.3	Porta . . . . .	223
20.4	Socket . . . . .	223
20.5	Servidor . . . . .	224
20.6	Cliente . . . . .	225
20.7	Imagem geral . . . . .	227
20.8	Exercícios: Sockets . . . . .	227
20.9	Desafio: Múltiplos Clientes . . . . .	229
20.10	Desafio: broadcast das mensagens . . . . .	229
20.11	Solução do sistema de chat . . . . .	230
<b>21</b>	<b>Apêndice C - Swing básico</b>	<b>234</b>
21.1	Interfaces gráficas em Java . . . . .	234
21.2	Portabilidade . . . . .	234
21.3	Começando com Swing . . . . .	234
21.4	Nosso primeiro formulário . . . . .	235
21.5	Adicionando eventos . . . . .	237
21.6	Gerenciadores de Layout . . . . .	239
21.7	Look And Feel . . . . .	240
21.8	Para saber mais . . . . .	241

<b>22 Apêndice D - Outros conceitos e APIs importantes</b>	<b>242</b>
22.1 Import Estático . . . . .	242
22.2 final . . . . .	243
22.3 Calendar . . . . .	243
22.4 Date . . . . .	245
22.5 DateFormat . . . . .	245
22.6 Outras classes muito úteis . . . . .	245
22.7 Anotações . . . . .	246
<b>23 Apêndice E - Instalação do Java</b>	<b>247</b>
23.1 Escolhendo a VM . . . . .	247
23.2 Instalando no Ubuntu e outros Linux . . . . .	247
23.3 No Mac OS X e Solaris . . . . .	247
23.4 Instalação do JDK em ambiente Windows . . . . .	248
<b>Índice Remissivo</b>	<b>253</b>

**Versão: 9.8.12**

# Como Aprender Java

*“Homens sábios fazem provérbios, tolos os repetem”*  
– Samuel Palmer

## 1.1 - O que é realmente importante?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes da linguagem juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial pois o estudante não consegue distinguir exatamente o que é importante aprender e reter naquele momento daquilo que será necessário mais tempo e principalmente experiência para dominar.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o if só a aceitar argumentos booleanos e todos os detalhes de classes internas realmente não devem ser preocupações para aquele que possui como objetivo primário aprender Java. Esse tipo de informação será adquirida com o tempo, e não é necessário até um segundo momento.

Neste curso, separamos essas informações em quadros especiais, já que são informações extras. Ou então, apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Algumas informações não são mostradas e podem ser adquiridas em tutoriais ou guias de referência, são detalhes que para um programador experiente em Java pode ser importante, mas não para quem está começando.

Por fim falta mencionar sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos quando o curso acabar. De qualquer maneira recomendamos aos alunos estudar em casa, principalmente aqueles que fazem os cursos intensivos.

### O curso

Para aqueles que estão fazendo o curso Java e Orientação a Objetos, é recomendado estudar em casa aquilo que foi visto durante a aula, tentando resolver os exercícios que não foram feitos e os desafios que estão lá para envolver mais o leitor no mundo de Java.

### Convenções de Código

Para mais informações sobre as convenções de código-fonte Java, acesse: <http://java.sun.com/docs/codeconv/>

## 1.2 - Sobre os exercícios

Os exercícios do curso variam de práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

Existem também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e, principalmente, familiarizar o aluno com a biblioteca padrão Java, além de proporcionar um ganho na velocidade de desenvolvimento.

## 1.3 - Tirando dúvidas

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (<http://www.guj.com.br/>), onde sua dúvida será respondida prontamente.

Se você já participa de um grupo de usuários java ou alguma lista de discussão, pode tirar suas dúvidas nos dois lugares.

Fora isso, sinta-se a vontade de entrar em contato conosco para tirar todas as suas dúvidas que tiver durante o curso.

## 1.4 - Bibliografia

Bibliografia recomendada para iniciantes na plataforma Java

- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra
- (Avançado) Effective Java Programming Language Guide, de Josh Bloch

# O que é Java

*“Computadores são inúteis, eles apenas dão respostas”*

– Picasso

Chegou a hora de responder as perguntas mais básicas sobre Java. Ao término desse capítulo você será capaz de:

- responder o que é Java;
- mostrar as vantagens e desvantagens do Java;
- compilar e executar um programa simples.

## 2.1 - Java

Muitos associam Java com uma maneira de deixar suas páginas da web mais bonitas, com efeitos especiais, ou para fazer pequenos formulários na web.

O que associa as **empresas** ao **Java**?

Já chegaremos neste ponto, mas antes vamos ver o motivo pelo qual as empresas fogem das outras linguagens.

Quais são os seus maiores problemas quando está programando?

- ponteiros?
- gerenciamento de memória?
- organização?
- falta de bibliotecas?
- ter de reescrever parte do código ao mudar de sistema operacional?
- custo de usar a tecnologia?

Java tenta amenizar esses problemas. Alguns desses objetivos foram atingidos muito tempo atrás, porque, antes do Java 1.0, a idéia é que essa linguagem fosse usada em pequenos dispositivos, como tvs, video-cassetes, aspiradores, liquidificadores e outros. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (**applets**). Hoje em dia esse não é mais o foco da linguagem: apesar de ter sido idealizada com um propósito e lançada com outro, o Java ganhou destaque no lado do servidor.

O Java é desenvolvido e mantido pela Sun (<http://www.sun.com>) e seu site principal é o <http://java.sun.com>. ([java.com](http://java.com) é um site mais institucional, voltado ao consumidor de produtos e usuários leigos, não desenvolvedores).

#### Breve história do Java

A Sun criou um time (conhecido como Green Team) para desenvolver inovações tecnológicas em 1992. Esse time foi liderado por James Gosling, considerado o pai do Java. O time voltou com a idéia de criar um interpretador (já era uma máquina virtual, veremos o que é isso mais a frente) para pequenos dispositivos, facilitando a reescrita de software para aparelhos eletrônicos, como vídeo cassete, televisão e aparelhos de tv a cabo.

A idéia não deu certo. Tentaram fechar diversos contratos com grandes fabricantes de eletrônicos, como a Panasonic, mas não houve êxito devido ao conflito de interesses. Hoje sabemos que o Java domina o mercado de aplicações para celulares, porém parece que em 1994 ainda era muito cedo. Com o advento da web, a Sun percebeu que poderia utilizar a idéia criada em 1992 para possibilitar rodar pequenas aplicações dentro do browser. A semelhança era que na internet havia uma grande quantidade de sistemas operacionais e browsers, e com isso seria grande vantagem poder programar numa única linguagem, independente da plataforma. Foi aí que o Java 1.0 foi lançado: focado em transformar o browser de apenas um cliente burro para uma aplicação que possa também realizar operações, não apenas renderizar html.

Sabemos que, hoje, os applets realmente não são o foco da Sun. É engraçado que a tecnologia Java nasceu com um objetivo em mente, foi lançado com outro, mas, no final, decolou mesmo no desenvolvimento de aplicações do lado do servidor. Sorte?

Você pode ler a história da linguagem Java em: <http://java.sun.com/java2/whatis/1996/storyofjava.html>

No Brasil, diversos grupos de usuários se juntaram para tentar disseminar o conhecimento da linguagem. Um deles é o GUJ (<http://www.guj.com.br>), uma comunidade virtual com artigos, tutoriais e fórum para tirar dúvidas, o maior em língua portuguesa.

Encorajamos todos os alunos a usar muito os fóruns do mesmo, pois é uma das melhores maneiras para achar soluções para pequenos problemas que acontecem com grande frequência.

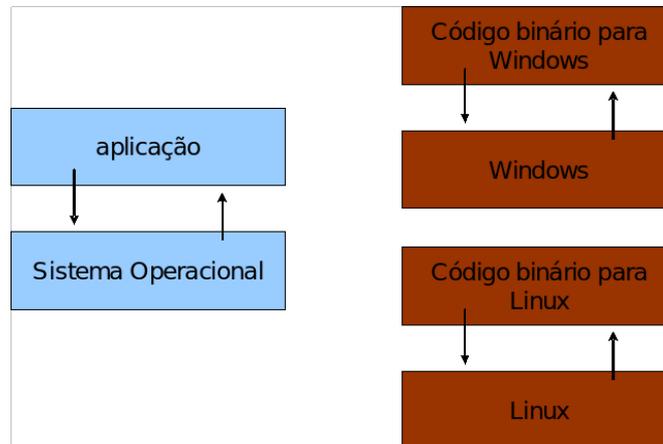
## 2.2 - Máquina Virtual

Em uma linguagem de programação como C e Pascal, temos o seguinte quadro quando vamos compilar um programa.



O código fonte é compilado para uma plataforma e sistema operacional específicos. Muitas vezes, o próprio código fonte é desenvolvido visando uma única plataforma!

Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber conversar com o sistema operacional em questão.

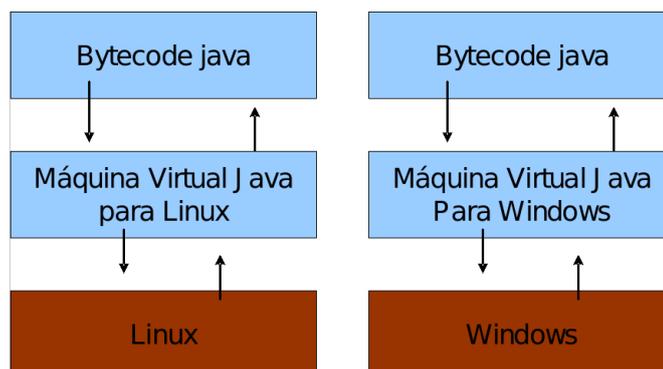


Isto é, temos um código executável para cada sistema operacional. É necessário compilar uma vez para Windows, outra para o Linux, etc...

Como foi dito anteriormente, na maioria das vezes, a sua aplicação se utiliza das bibliotecas do sistema operacional, como, por exemplo, a de interface gráfica para desenhar as ‘telinhas’. A biblioteca de interface gráfica do Windows é bem diferente das do Linux; resultado?

Você precisa reescrever o mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis.

Já o Java, se utiliza do conceito de **máquina virtual**, onde existe, entre o sistema operacional e a aplicação, uma camada extra responsável por “traduzir” - mas não apenas isso - o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento:



Dessa forma, a maneira com a qual você abre uma janela no Linux ou no Windows é a mesma: você ganha independência de sistema operacional. Ou, melhor ainda, independência de plataforma em geral: não é preciso se preocupar em qual sistema operacional sua aplicação está rodando, nem em que tipo de máquina, configurações etc.

Repare que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Como o próprio nome diz, uma máquina virtual é como um computador de mentira: tem tudo que um computador tem. Em outras palavras, ela é responsável por gerenciar memória e threads, a pilha de execução, etc.

Sua aplicação roda sem nenhum envolvimento com o sistema operacional! Sempre conversando apenas com a **Java Virtual Machine (JVM)**.

Essa característica é interessante: como tudo passa pela JVM, ela pode tirar métricas, decidir onde é melhor alocar a memória, entre outros. Uma JVM isola totalmente a aplicação do sistema operacional. Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar: isso não afetará outras JVMs que estejam rodando no mesmo computador, nem afetará o sistema operacional.

Essa camada de isolamento também é interessante quando pensamos em um servidor que não pode se sujeitar a rodar código que possa interferir na boa execução de outras aplicações.

Para tudo isso precisamos de um “**bytecode**”. Bytecode é o termo dado ao código binário gerado pelo compilador Java (pois existem menos de 256 códigos de operação dessa linguagem, e cada “opcode” gasta um byte, dando origem ao nome bytecode). O compilador Java gera esse bytecode que, diferente das linguagens sem máquina virtual, vai servir para diferentes sistemas operacionais, já que ele vai ser “traduzido” pela máquina virtual.

Write once, run anywhere

Esse é um slogan que a Sun usa para o Java, já que você não precisa reescrever parte da sua aplicação toda vez que quiser mudar de sistema operacional.

Muitas pessoas criticam ou criam piadas em cima desse slogan, por acharem que nem sempre uma aplicação Java pode rodar em duas plataformas diferentes sem problemas.

## 2.3 - Java lento? Hotspot e JIT

*Hotspot* é a tecnologia que a JVM utiliza para detectar *pontos quentes* da sua aplicação: código que é executado muito, provavelmente dentro de um ou mais loops. Quando a JVM julgar necessário, ela vai compilar aquele código para instruções nativas da plataforma, tendo em vista que isso vai provavelmente melhorar a performance da sua aplicação. Esse compilador é o *JIT: Just inTime Compiler*, o compilador que aparece “bem na hora” que você precisa.

Você pode pensar então: porque a JVM não compila tudo antes de executar a aplicação? É que teoricamente compilar dinamicamente, a medida do necessário, pode gerar uma performance melhor. O motivo é simples: imagine um .exe gerado pelo VisualBasic, pelo gcc ou pelo Delphi; ele é estático. Ele já foi otimizado baseado em heurísticas, o compilador pode ter tomado uma decisão não tão boa.

Já a JVM, por estar compilando dinamicamente durante a execução, pode perceber que um determinado código não está com performance adequada e otimizar mais um pouco aquele trecho, ou ainda mudar a estratégia de otimização. É por esse motivo que as JVMs mais recentes (como a do Mustang, Java 6), **em alguns casos**, chega a ganhar, em muitos casos, de códigos C compilados com o GCC 3.x, se rodados durante um certo tempo.

## 2.4 - Versões do Java... e a confusão do Java2

Java 1.0 e 1.1 são as versões muito antigas do Java.

Com o Java 1.2 houve um aumento grande no tamanho da API, e foi nesse momento em que trocaram a nomenclatura de Java para Java2, com o objetivo de diminuir a confusão entre Java e Javascript. Mas lembre-se, não há versão do Java 2.0, o 2 foi incorporado ao nome: Java2 1.2.

Depois vieram o Java2 1.3 e 1.4, e o Java 1.5 passou a se chamar Java 5, tanto por uma questão de marketing e porque mudanças significativas na linguagem foram incluídas. É nesse momento que o “2” do nome Java desaparece. Repare que para fins de desenvolvimento, o Java 5 ainda é referido como Java 1.5.

Até a versão 1.4, existia a terceira numeração (1.3.1, 1.4.1, 1.4.2, etc), indicando bug fixes e melhorias. A partir do Java 5 existem apenas updates: Java 5 update 7, por exemplo.

Hoje a última versão disponível do Java é a 6.0, lançada em 2006. Da versão 1.4 para a 5.0, a linguagem sofreu muitas modificações, o que de certa forma fomentou a mudança no versionamento do Java. Já o Java 6.0 não trouxe nenhuma mudança na linguagem, mas trouxe mais recursos na API e *muitas* melhorias de performance na VM.

Existe compatibilidade para trás em todas as versões do Java. Um class gerado pelo javac da versão 1.2 precisa necessariamente rodar na JVM 6.0. Por isso, recomendamos sempre usar a última versão do Java para usufruir das tecnologias mais modernas mas sem correr o risco de quebrar aplicações antigas.

## 2.5 - JVM? JRE? JDK?

O que você vai baixar no site do java?

- JVM = apenas a virtual machine, esse download não existe
- JRE = **Java Runtime Environment**, ambiente de execução Java, formado pela JVM e bibliotecas, tudo que você precisa para executar uma aplicação Java.
- JDK = **Java Development Kit**: Nós, desenvolvedores, faremos o download do JDK do Java SE (Standard Edition).

## 2.6 - Onde usar e os objetivos do Java

No decorrer do curso, você pode achar que o Java tem baixa produtividade, que a linguagem com a qual você está acostumado é mais simples para criar os pequenos sistemas que estamos vendo aqui.

Queremos deixar **claro** que a premissa do Java não é a de criar sistemas pequenos, onde temos um ou dois desenvolvedores, mais rapidamente que linguagens como php, perl, entre outras.

O foco da plataforma é outro: aplicações de *médio a grande porte*, onde o time de desenvolvedores tem *várias pessoas* e sempre pode vir a *mudar e crescer*. Não tenha dúvidas que criar a primeira versão uma aplicação usando Java, mesmo utilizando IDEs e ferramentas poderosas, será mais trabalhoso que usar uma linguagem script ou de alta produtividade. Porém, com uma linguagem orientada a objetos e madura como o Java, será extremamente mais fácil e rápido fazer alterações no sistema, desde que você siga as boas práticas, recomendações e *design patterns*.

Além disso, a quantidade enorme de bibliotecas gratuitas para realizar os mais diversos trabalhos (tais como relatórios, gráficos, sistemas de busca, geração de código de barra, manipulação de XML, tocadores de vídeo, manipuladores de texto, persistência transparente, impressão, etc) é um ponto fortíssimo para adoção do Java: você pode criar uma aplicação sofisticada, usando diversos recursos, sem precisar comprar um componente específico, que costuma ser caro de acordo com sua especialização.

Cada linguagem tem seu espaço e seu melhor uso. O uso do Java é interessante em aplicações que virão a crescer, em que a legibilidade do código é importante, onde temos muita conectividade e se temos plataformas (ambientes e sistemas operacionais) heterogêneos (Linux, Unix, OSX e Windows misturados).

Você pode ver isso pela quantidade enorme de ofertas de emprego procurando desenvolvedores Java para trabalhar com sistemas web e aplicações de integração no servidor.

Apesar disto, a Sun empenha-se em tentar popularizar o uso do Java em aplicações desktop, mesmo com o fraco marketshare do Swing/AWT/SWT em relação às tecnologias concorrentes (em especial Microsoft .NET).

## 2.7 - Especificação versus implementação

Outro ponto importante: quando falamos de Java Virtual Machine, estamos falando de uma especificação. Ela diz como o bytecode deve ser interpretado pela JVM. Quando fazemos o download no site da Sun, o que vem junto é a Sun JVM. Em outras palavras, existem outras JVMs disponíveis, como a Jrockit da BEA, entre outras.

Isso é outro ponto interessante para as empresas. Caso não estejam gostando de algum detalhe da JVM da Sun ou prefiram trabalhar com outra empresa, pagando por suporte, elas podem trocar de JVM com a garantia absoluta de que todo o sistema continuará funcionando, tendo em vista que a JVM é certificada pela Sun, precisando aceitar o mesmo bytecode. Você nem precisa recompilar nenhuma de suas classes.

Além de independência de hardware e sistema operacional, você tem a independência de **vendor**: graças a idéia da JVM ser uma especificação e não um software.

## 2.8 - Como o FJ-11 está organizado

Java é uma linguagem simples no sentido de que as regras não são muitas. Quebrar o paradigma procedural para mergulhar na orientação a objetos não é simples. Esse é o objetivo do FJ-11.

O começo pode ser um pouco frustrante: exemplos banais, controle de fluxo simples com o if, for, while e criação de pequenos programas que nem ao menos captam informação do teclado. Apesar de isto tudo ser necessário, é só nos 20% finais do curso que estaremos utilizando bibliotecas para, no final, criarmos um chat entre duas máquinas que transferem Strings por TCP/IP. Neste ponto, teremos tudo que é necessário para entender completamente como a API funciona, quem estende quem, e o porquê.

Depois desse capítulo onde o Java, a JVM e primeiros conceitos são passados, veremos os comandos básicos do java para controle de fluxo e utilização de variáveis do tipo primitivo. Criaremos classes para testar esse pequeno aprendizado, sem mesmo saber o que realmente é uma classe. Isso dificulta ainda mais a curva de aprendizado, porém cada conceito será introduzido no momento considerado mais apropriado pelos autores.

Passamos para o capítulo de orientação a objetos básico, mostrando os problemas do paradigma procedural e a necessidade de algo para resolvê-los. Atributos, métodos, variáveis do tipo referência e outros. Depois, um rápido pulo pelos arrays.

Os capítulos de modificadores de acesso, herança, classes abstratas e interfaces demonstram o conceito fundamental que o curso quer passar: encapsule, exponha o mínimo de suas classes, foque no que elas fazem, no relacionamento entre elas. Com uma modelagem boa a codificação fica fácil e a modificação e expansão do sistema também.

Enquanto isso, o Eclipse é introduzido de forma natural, evitando-se ao máximo wizards e menus, e sim o code assist e seus quick fixes. Isso faz com que o Eclipse trabalhe de forma simbiótica com o desenvolvedor, sem se intrometer e fazer mágica.

Pacotes, javadoc, jars e java.lang apresentam os últimos conceitos fundamentais do Java, dando toda a fundação para, então, passarmos a estudar as principais e mais utilizadas APIs do Java SE.

Java.util, java.io e java.net são essas APIs. Todas elas usam e abusam dos conceitos vistos no decorrer do curso, ajudando a sedimentá-los. Juntamente, temos os conceitos básicos do uso de Threads, e os problemas

e perigos da programação concorrente quando dados são compartilhados.

Resumindo: o objetivo do curso é apresentar o Java ao mesmo tempo que os fundamentos da orientação a objetos são introduzidos. Bateremos muito no ponto de dizer que o importante é como as classes se relacionam e qual é o papel de cada uma, e não em como elas realizam as suas obrigações. *Programa voltado à interface, e não à implementação.*

No final do curso, a utilização dos pacotes `java.io`, `java.util` e `java.net`, e de uma biblioteca de desenho de gráficos (JFreeChart) reforçarão todo nosso aprendizado, tendo em vista que essas bibliotecas foram desenvolvidas com o objetivo de reutilização e extensão, portanto abusam dos conceitos previamente vistos.

## 2.9 - Compilando o primeiro programa

Vamos para o nosso primeiro código! O programa que imprime uma linha simples!

```
1 class MeuPrograma {
2     public static void main(String[] args) {
3         System.out.println("Minha primeira aplicação Java!!");
4     }
5 }
```

### Notação

Todos os códigos apresentados na apostila estão formatados com recursos visuais para auxiliar a leitura e compreensão dos mesmos. Quando for digitar os códigos no computador, trate os códigos como texto simples.

A numeração das linhas **não** faz parte do código e não deve ser digitada; é apenas um recurso didático. O Java é case sensitive: tome cuidado com maiúsculas e minúsculas.

Após digitar o código acima, grave-o como `MeuPrograma.java` em algum diretório. Para compilar, você deve pedir para que o compilador de Java da Sun, chamado `javac`, gere o bytecode correspondente ao seu código Java.

```
teste@andrade:/home/moreira/java$ javac MeuPrograma.java
teste@andrade:/home/moreira/java$ ls -l
total 8
-rw-r--r-- 1 teste teste 442 2006-09-06 16:41 MeuPrograma.class
-rw-r--r-- 1 teste teste 119 2006-09-06 16:36 MeuPrograma.java
teste@andrade:/home/moreira/java$ █
```

Depois de compilar, o **bytecode** foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo **.class** foi gerado, com o mesmo nome da sua classe Java.

### Assustado com o código?

Para quem já tem uma experiência com Java, esse primeiro código é muito simples. Mas, se é seu primeiro código em Java, pode ser um pouco traumatizante. Não deixe de ler o prefácio do curso, que deixará você mais tranquilo.

### Preciso sempre programar usando o Notepad ou similar?

Não é necessário digitar sempre seu programa em um simples aplicativo como o Notepad. Você pode usar um editor que tenha **syntax highlighting** e outros benefícios.

Mas, no começo, é interessante você usar algo que não possua ferramentas, para que você possa se acostumar com os erros de compilação, sintaxe e outros. Depois do capítulo de polimorfismo e herança sugerimos a utilização do Eclipse (<http://www.eclipse.org>), a IDE líder no mercado, e gratuita. Existe um capítulo a parte para o uso do Eclipse nesta apostila.

No Linux, recomendamos o uso do gedit ou do kate. No Windows, você pode usar o Notepad++ ou o TextPad. No Mac, TextWrangler ou TextMate.

## 2.10 - Executando seu primeiro programa

Os procedimentos para executar seu programa são muito simples. O javac é o compilador Java, e o java é o responsável por invocar a máquina virtual para interpretar o seu programa.

```
teste@andrade:/home/moreira/java$ java MeuPrograma
Meu primeiro programa java
teste@andrade:/home/moreira/java$ █
```

Ao executar, pode ser que a acentuação resultante saia errada devido a algumas configurações que deixamos de fazer. Sem problemas.

## 2.11 - O que aconteceu?

```
1 class MeuPrograma {
2     public static void main(String[] args) {
3         // miolo do programa começa aqui!
4         System.out.println("Minha primeira aplicação Java!!");
5         // fim do miolo do programa
6     }
7 }
```

O miolo do programa é o que será executado quando chamamos a máquina virtual. Por enquanto, todas as linhas anteriores, onde há a declaração de uma classe e a de um método, não importam para nós. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e este ponto de entrada é um método `main`.

Ainda não sabemos o que é método, mas veremos no capítulo 4. Até lá, não se preocupe com essas declarações. Sempre que um exercício for feito, o código sempre estará nesse miolo.

No caso do nosso código, a linha do `System.out.println` faz com que o conteúdo entre aspas seja colocado na tela.

## 2.12 - E o bytecode?

O `MeuPrograma.class` gerado não é legível por seres humanos (não que seja impossível). Ele está escrito no formato que a virtual machine sabe entender e que foi especificado que ela entendesse.

É como um assembly, escrito para esta máquina em específico. Podemos ler os mnemônicos utilizando a ferramenta javap que acompanha o JDK:

```
javap -c MeuPrograma
```

E a saída:

```
MeuPrograma();  
Code:  
0:  aload_0  
1:  invokespecial  #1; //Method java/lang/Object."<init>":()V  
4:  return  
  
public static void main(java.lang.String[]);  
Code:  
0:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;  
3:  ldc           #3; //String Minha primeira aplicação Java!!  
5:  invokevirtual  #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V  
8:  return  
  
}
```

É o código acima, que a JVM sabe ler. É o “código de máquina”, da máquina virtual.

Um bytecode pode ser revertido para o .java original (com perda de comentários e nomes de variáveis locais). Caso seu software vá virar um produto de prateleira, é fundamental passar um ofuscador no seu código, que vai embaralhar classes, métodos e um monte de outros recursos (indicamos o <http://proguard.sf.net>).

## 2.13 - Exercícios: Modificando o Hello World

- 1) Altere seu programa para imprimir uma mensagem diferente.
- 2) Altere seu programa para imprimir duas linhas de texto usando duas linhas de código `System.out`.
- 3) Sabendo que os caracteres `\n` representam uma quebra de linhas, imprima duas linhas de texto usando uma única linha de código `System.out`.

## 2.14 - O que pode dar errado?

Muitos erros podem ocorrer no momento que você rodar seu primeiro código. Vamos ver alguns deles:

Código:

```
1 public class X {  
2     public static void main (String[] args) {  
3         System.out.println("Falta ponto e vírgula")  
4     }  
5 }
```

Erro:

```
X.java:4: ';' expected
    }
    ^
1 error
```

Esse é o erro de compilação mais comum: aquele onde um ponto e vírgula fora esquecido. Outros erros de compilação podem ocorrer se você escreveu palavras chaves (as que colocamos em negrito) em maiúsculas, esqueceu de abrir e fechar as {}, etc.

Durante a execução, outros erros podem aparecer:

- Se você declarar a classe como X, compilá-la e depois tentar usá-la como x minúsculo (java x), o Java te avisa:

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

- Se tentar acessar uma classe no diretório ou classpath errado, ou se o nome estiver errado, ocorrerá o seguinte erro:

```
Exception in thread "main" java.lang.NoClassDefFoundError: X
```

- Se esquecer de colocar `static` ou o argumento `String[] args` no método `main`:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Por exemplo:

```
1 public class X {
2     public void main (String[] args) {
3         System.out.println("Faltou o static");
4     }
5 }
```

- Se não colocar o método `main` como `public`:

```
Main method not public.
```

Por exemplo:

```
1 public class X {
2     static void main (String[] args) {
3         System.out.println("Faltou o public");
4     }
5 }
```

## 2.15 - Um pouco mais...

1) Procure um colega, ou algum conhecido, que esteja em um projeto Java. Descubra porque Java foi escolhido como tecnologia. O que é importante para esse projeto e o que acabou fazendo do Java a melhor escolha?

---

## 2.16 - Exercícios adicionais

- 1) Um arquivo fonte Java deve sempre ter a extensão `.java`, ou o compilador o rejeitará. Além disso, existem algumas outras regras na hora de dar o nome de um arquivo Java. Experimente gravar o código deste capítulo com `OutroNome.java` ou algo similar. Compile e verifique o nome do arquivo gerado. Como executar a sua aplicação agora?

### Curiosidade

Tente compilar um arquivo sem nada dentro, nem uma letra, nem uma quebra de linha. O que acontece?

---

# Variáveis primitivas e Controle de fluxo

*“Péssima idéia, a de que não se pode mudar”*  
– Montaigne

Aprenderemos a trabalhar com os seguintes recursos da linguagem Java:

- declaração, atribuição de valores, casting e comparação de variáveis;
- controle de fluxo através de `if` e `else`;
- instruções de laço `for` e `while`, controle de fluxo com `break` e `continue`.

## 3.1 - Declarando e usando variáveis

**Dentro de um bloco**, podemos declarar variáveis e usá-las.

Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, é possível ter uma `idade` que vale um número inteiro:

```
int idade;
```

Com isso, você declara a variável `idade`, que passa a existir a partir deste momento. Ela é do tipo `int`, que guarda um número inteiro. A partir de agora você pode usá-la, primeiro atribuindo valores.

A linha a seguir é a tradução de **“idade deve valer agora quinze”**.

```
idade = 15;
```

### Comentários em Java

Para fazer um comentário em java, você pode usar o `//` para comentar até o final da linha, ou então usar o `/* */` para comentar o que estiver entre eles.

```
/* comentário daqui,  
ate aqui */
```

```
//uma linha de comentário sobre a idade  
int idade;
```

Além de atribuir, você pode utilizar esse valor. O código a seguir declara novamente a variável `idade` com valor 15 e imprime seu valor na saída padrão através da chamada a `System.out.println`.

```
//declara a idade
int idade;
idade = 15;

// imprime a idade
System.out.println(idade);
```

Por fim, podemos utilizar o valor de uma variável para algum outro propósito, como alterar ou definir uma segunda variável. O código a seguir cria uma variável chamada `idadeNoAnoQueVem` com valor de **idade mais um**.

```
//gera a idade no ano seguinte
int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

No momento que você declara uma variável, também é possível inicializá-la por praticidade:

```
int idade = 15;
```

Você pode usar os operadores `+`, `-`, `/` e `*` para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador `%` (módulo) que nada mais é que o **resto de uma divisão inteira**. Veja alguns exemplos:

```
int quatro = 2 + 2;
int tres = 5 - 2;

int oito = 4 * 2;
int dezesseis = 64 / 4;

int um = 5 % 2; // 5 dividido por 2 dá 2 e tem resto 1;
               // o operador % pega o resto da divisão inteira
```

### Onde testar esses códigos?

Você deve colocar esses trechos de código dentro do método main, que vimos no capítulo anterior. Isto é, isso deve ficar no miolo do programa. Use bastante `System.out.println`, dessa forma você pode ver algum resultado, caso contrário, ao executar a aplicação, nada aparecerá. Por exemplo, para imprimir a idade e a idadeNoAnoQueVem podemos escrever o seguinte programa de exemplo:

```
class TestaIdade {  
  
    public static void main(String[] args) {  
  
        // declara a idade  
        int idade;  
        idade = 15;  
  
        // imprime a idade  
        System.out.println(idade);  
  
        // gera uma idade no ano seguinte  
        int idadeNoAnoQueVem;  
        idadeNoAnoQueVem = idade + 1;  
  
        // imprime a idade  
        System.out.println(idadeNoAnoQueVem);  
  
    }  
}
```

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o `double`, que armazena um número com ponto flutuante.

```
double pi = 3.14;  
double x = 5 * 10;
```

O tipo `boolean` armazena um valor verdadeiro ou falso, e só.

```
boolean verdade = true;
```

O tipo `char` guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo `char`! Por exemplo, ela não pode guardar um código como "" pois o vazio não é um caractere!

```
char letra = 'a';  
System.out.println(letra);
```

## 3.2 - Tipos primitivos e valores

Esses tipos de variáveis são tipos primitivos do Java: o valor que elas guardam são o real conteúdo da variável. Quando você utilizar o **operador de atribuição** = o valor será **copiado**.

```
int i = 5; // i recebe uma cópia do valor 5
int j = i; // j recebe uma cópia do valor de i
i = i + 1; // i vira 6, j continua 5
```

Aqui, *i* fica com o valor de 6. Mas e *j*? Na segunda linha, *j* está valendo 5. Quando *i* passa a valer 6, será que *j* também muda de valor? Não, pois o valor de um tipo primitivo sempre é copiado.

Apesar da linha 2 fazer *j = i*, a partir desse momento essas variáveis não tem relação nenhuma: o que acontece com uma, não reflete em nada com a outra.

#### Outros tipos primitivos

Vimos aqui os tipos primitivos que mais aparecem. O Java tem outros, que são o `byte`, `short`, `long` e `float`.

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença.

### 3.3 - Exercícios: Variáveis e tipos primitivos

1) Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:

- Crie uma classe chamada `BalancoTrimestral` com um bloco `main`, como nos exemplos anteriores;
- Dentro do `main` (o miolo do programa), declare uma variável inteira chamada `gastosJaneiro` e inicialize-a com 15000;
- Crie também as variáveis `gastosFevereiro` e `gastosMarco`, inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;
- Crie uma variável chamada `gastosTrimestre` e inicialize-a com a soma das outras 3 variáveis:

```
int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;
```

e) Imprima a variável `gastosTrimestre`.

2) (opcional) Adicione código (sem alterar as linhas que já existem) no programa a seguir para imprimir o resultado:

Resultado: 15, 15.1, y, false

```
class ExercicioSimples {

    public static void main(String[] args) {

        int i = 10;
        double d = 5;
        char c = 't';
        boolean b = true;

        // imprime concatenando diversas variáveis
    }
}
```

```
        System.out.println("Resultado: " + i + ", " + d + ", " + c + ", " + b);  
    }  
}
```

### 3.4 - Discussão em aula: convenções de código e código legível

Discuta com o instrutor e seus colegas sobre convenções de código Java. Por que existem? Por que são importantes?

Discuta também as vantagens de se escrever código fácil de ler e se evitar comentários em excesso.

(Dica: procure por *code conventions* no campo de busca do site <http://java.sun.com>.)

### 3.5 - Casting e promoção

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo `double`, tentar atribuir ele a uma variável `int` não funciona porque é um código que diz: “**i deve valer d**”, mas não se sabe se `d` realmente é um número inteiro ou não.

```
double d = 3.1415;  
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro  
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um `int`, o compilador não tem como saber que valor estará dentro desse `double` no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

Já no caso a seguir, é o contrário:

```
int i = 5;  
double d2 = i;
```

O código acima compila sem problemas, já que um `double` pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo `int` podem ser guardados em uma variável `double`, então não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;  
int i = (int) d3;
```

O casting foi feito para moldar a variável `d3` como um `int`. O valor de `i` agora é 3.

O mesmo ocorre entre valores `int` e `long`.

```
long x = 10000;  
int i = x; // nao compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;  
int i = (int) x;
```

#### Casos não tão comuns de casting e atribuição

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados `double` pelo Java. E `float` não pode receber um `double` sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra `f`, que pode ser maiúscula ou minúscula, indica que aquele literal deve ser tratado como `float`. Outro caso, que é mais comum:

```
double d = 5;  
float f = 3;
```

```
float x = f + (float) d;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o `double`.

E uma observação: no mínimo, o Java armazena o resultado em um `int`, na hora de fazer as contas.

Até casting com variáveis do tipo `char` podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o `boolean`.

### Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo boolean não pode ser convertido para nenhum outro tipo).

<b>PARA:</b>	<b>byte</b>	<b>short</b>	<b>char</b>	<b>int</b>	<b>long</b>	<b>float</b>	<b>double</b>
<b>DE:</b>							
<b>byte</b>	----	<i>Impl.</i>	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
<b>short</b>	(byte)	----	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
<b>char</b>	(byte)	(short)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
<b>int</b>	(byte)	(short)	(char)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
<b>long</b>	(byte)	(short)	(char)	(int)	----	<i>Impl.</i>	<i>Impl.</i>
<b>float</b>	(byte)	(short)	(char)	(int)	(long)	----	<i>Impl.</i>
<b>double</b>	(byte)	(short)	(char)	(int)	(long)	(float)	----

### Tamanho dos tipos

Na tabela abaixo, estão os tamanhos de cada tipo primitivo do Java.

<b>TIPO</b>	<b>TAMANHO</b>
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

## 3.6 - O If-Else

A sintaxe do if no Java é a seguinte

```
if (condicaoBooleana) {
    codigo;
}
```

Uma **condição booleana** é qualquer expressão que retorne true ou false. Para isso, você pode usar os operadores <, >, <=, >= e outros. Um exemplo:

```
int idade = 15;
if (idade < 18) {
```

```
    System.out.println("Não pode entrar");  
}
```

Além disso, você pode usar a cláusula `else` para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
int idade = 15;  
if (idade < 18) {  
    System.out.println("Não pode entrar");  
}  
else {  
    System.out.println("Pode entrar");  
}
```

Você pode concatenar expressões booleanas através dos operadores lógicos “E” e “OU”. O “E” é representado pelo `&` e o “OU” é representado pelo `|`.

```
int idade = 15;  
boolean amigoDoDono = true;  
if (idade < 18 & amigoDoDono == false) {  
    System.out.println("Não pode entrar");  
}  
else {  
    System.out.println("Pode entrar");  
}
```

Esse código poderia ficar ainda mais legível, utilizando-se o operador de negação, o `!`. Esse operador transforma uma expressão booleana de `false` para `true` e vice versa.

```
int idade = 15;  
boolean amigoDoDono = true;  
if (idade < 18 & !amigoDoDono) {  
    System.out.println("Não pode entrar");  
}  
else {  
    System.out.println("Pode entrar");  
}
```

Repare na linha 3 que o trecho `amigoDoDono == false` virou `!amigoDoDono`. **Eles têm o mesmo valor.**

Para comparar se uma variável tem o mesmo valor que outra variável ou valor, utilizamos o operador `==`. Repare que utilizar o operador `=` vai retornar um erro de compilação, já que o operador `=` é o de atribuição.

```
int mes = 1;  
if (mes == 1) {  
    System.out.println("Você deveria estar de férias");  
}
```

#### && ou &?

Em alguns livros, logo será apresentado a você dois tipos de operadores de OU e de E. Você realmente não precisa saber distinguir a diferença entre eles por enquanto.

O que acontece é que os operadores `&&` e `||` funcionam como seus operadores irmãos, porém eles funcionam da maneira mais rápida possível: quando percebem que a resposta não mudará mais, eles param de verificar as outras condições booleanas. Por isso, eles são chamados de operadores de curto circuito (short circuit operators).

```
if (true | algumaCoisa) {  
    // ...  
}
```

O valor de `algumaCoisa` será analisado, nesse caso. Repare que não precisaria, pois já temos um `true`. `true` ou qualquer outra coisa, dá sempre `true`.

```
if (true || algumaCoisa) {  
    // ...  
}
```

Neste caso o `algumaCoisa` não será analisado. Pode não fazer sentido ter as duas opções, mas em alguns casos é interessante e útil usar um ou outro, além de dar diferença no resultado. Veremos mais adiante em outros capítulos.

## 3.7 - O While

O `while` é um comando usado para fazer um **laço (loop)**, isto é, repetir um trecho de código algumas vezes. A idéia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;  
while(idade < 18) {  
    System.out.println(idade);  
    idade = idade + 1;  
}
```

O trecho dentro do bloco do `while` será executado até o momento em que a condição `idade < 18` passe a ser falsa. E isso ocorrerá exatamente no momento em que `idade == 18`, o que não o fará imprimir 18.

```
int i = 0;  
while(i < 10) {  
    System.out.println(i);  
    i = i + 1;  
}
```

Já o `while` acima imprime de 0 a 9.

## 3.8 - O For

Outro comando de **loop** extremamente utilizado é o `for`. A idéia é a mesma do `while`: fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o `for` isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fiquem mais legíveis, as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {  
    codigo;  
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {  
    System.out.println("olá!");  
}
```

Repare que esse `for` poderia ser trocado por:

```
int i = 0;  
while (i < 10) {  
    System.out.println("olá!");  
    i = i + 1;  
}
```

Porém, o código do `for` indica claramente que a variável `i` serve, em especial, para controlar a quantidade de laços executados. Quando usar o `for`? Quando usar o `while`? Depende do gosto e da ocasião.

#### Pós incremento ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

```
int i = 5;  
int x = i++;
```

Qual é o valor de `x`? O de `i`, após essa linha, é 6.

O operador `++`, quando vem após a variável, retorna o valor antigo, e incrementa (pós incremento), fazendo `x` valer 5.

Se você tivesse usado o `++` antes da variável (pré incremento), o resultado seria 6:

```
int i = 5;  
int x = ++i; // aqui x valera 6
```

## 3.9 - Controlando loops

Apesar de termos condições booleanas nos nossos laços, em algum momento podemos decidir parar o loop por algum motivo especial, sem que o resto do laço seja executado.

```
for (int i = x; i < y; i++) {  
    if (i % 19 == 0) {  
        System.out.println("Achei um número divisível por 19 entre x e y");  
        break;  
    }  
}
```

O código acima vai percorrer os números de `x` a `y` e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave `break`.

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave `continue`.

```
for (int i = 0; i < 100; i++) {
    if(i > 50 && i < 60) {
        continue;
    }
    System.out.println(i);
}
```

O código acima não vai imprimir alguns números. (Quais exatamente?)

### 3.10 - Escopo das variáveis

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
//aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e que é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
//aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
while (condicao) {
    // o i ainda vale aqui
    int j = 7;
    // o j passa a existir
}
// aqui o j não existe mais, mas o i continua a valer
```

No bloco acima, a variável `j` pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

```
EscopoDeVariavel.java:8: cannot find symbol
symbol   : variable j
location: class EscopoDeVariavel
        System.out.println(j);
                        ^
1 error
```

O mesmo vale para um `if`:

```
if (algumBooleano) {
    int i = 5;
}
```

```
else {  
    int i = 10;  
}  
System.out.println(i); // cuidado!
```

Aqui a variável `i` não existe fora do `if` e do `else`! Se você declarar a variável antes do `if`, vai haver outro erro de compilação: dentro do `if` e do `else` a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;  
if (algumBooleano) {  
    i = 5;  
}  
else {  
    i = 10;  
}  
System.out.println(i);
```

Uma situação parecida pode ocorrer com o `for`:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("olá!");  
}  
System.out.println(i); // cuidado!
```

Neste `for` a variável `i` morre ao seu término, não podendo ser acessada de fora do `for`, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

```
int i;  
for (i = 0; i < 10; i++) {  
    System.out.println("olá!");  
}  
System.out.println(i);
```

### 3.11 - Um bloco dentro do outro

Um bloco também pode ser declarado dentro de outro. Isto é, um `if` dentro de um `for`, ou um `for` dentro de um `for`, algo como:

```
while (condicao) {  
    for (int i = 0; i < 10; i++) {  
        // código  
    }  
}
```

### 3.12 - Um pouco mais...

1) Vimos apenas os comandos mais usados para controle de fluxo. O Java ainda possui o `do..while` e o `switch`. Pesquise sobre eles e diga quando é interessante usar cada um deles.

- 2) Algumas vezes temos vários laços encadeados. Podemos utilizar o break para quebrar o laço mais interno, mas se quisermos quebrar um laço mais externo, teremos de encadear diversos ifs e seu código ficará uma bagunça. O Java possui um artifício chamado **labeled loops**, pesquise sobre eles.
- 3) O que acontece se você tentar dividir um número inteiro por 0? E por 0.0?
- 4) Existe um caminho entre os tipos primitivos que indicam se há a necessidade ou não de casting entre os tipos. Por exemplo, `int -> long -> double` (um `int` pode ser tratado como um `double`, mas não o contrário). Pesquise (ou teste), e posicione os outros tipos primitivos nesse fluxo.
- 5) Existem outros operadores, como o `%`, `<<`, `>>`. Descubra para que servem.
- 6) Além dos operadores de incremento, existem os de decremento, como `--i` e `i--`. Além desses, você pode usar instruções do tipo `i += x` e `i -= x`, o que essas instruções fazem? Teste.

### 3.13 - Exercícios: Fixação de sintaxe

Mais exercícios de fixação de sintaxe. Para quem já conhece um pouco de java pode ser muito simples, mas recomendamos fortemente que você faça os exercícios para se acostumar com erros de compilação, mensagens do javac, convenção de código, etc...

Apesar de extremamente simples, precisamos praticar a sintaxe que estamos aprendendo. Para cada exercício, crie um novo arquivo com extensão java, e declare aquele estranho cabeçalho, dando nome a uma classe e com um método main dentro dele:

```
class ExercicioX {  
    public static void main(String[] args) {  
        // seu exercicio vai aqui  
    }  
}
```

Não copie e cole de um exercício já existente! Aproveite para praticar.

- 1) Imprima todos os números de 150 a 300.
- 2) Imprima a soma de 1 até 1000.
- 3) Imprima todos os múltiplos de 3, entre 1 e 100.
- 4) Imprima os fatoriais de 1 a 10.  
O fatorial de um número  $n$  é  $n * n-1 * n-2 \dots$  até  $n = 1$ . Lembre-se de utilizar os parênteses. O fatorial de 0 é 1 O fatorial de 1 é  $(0!) * 1 = 1$  O fatorial de 2 é  $(1!) * 2 = 2$  O fatorial de 3 é  $(2!) * 3 = 6$  O fatorial de 4 é  $(3!) * 4 = 24$   
Faça um for que inicie uma variável  $n$  (número) como 1 e fatorial (resultado) como 1 e varia  $n$  de 1 até 10:  

```
for (int n=1, fatorial=1; n <= 10; n++) {
```
- 5) Aumente a quantidade de números que terão os fatoriais impressos, até 20, 30, 40. Em um determinado momento, além desse cálculo demorar, vai começar a mostrar respostas completamente erradas. Porque? Mude de `int` para `long`, e você poderá ver alguma mudança.

- 6) (opcional) Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc... Para calculá-la, o primeiro e segundo elementos valem 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somando ao (n-2)-ésimo elemento (ex:  $8 = 5 + 3$ ).
- 7) (opcional) Escreva um programa que, dada uma variável x (com valor 180, por exemplo), temos y de acordo com a seguinte regra:
- se x é par,  $y = x / 2$
  - se x é ímpar,  $y = 3 * x + 1$
  - imprime y
  - O programa deve então jogar o valor de y em x e continuar até que y tenha o valor final de 1. Por exemplo, para  $x = 13$ , a saída será:  
40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

#### Imprimindo sem pular linha

Um detalhe importante do método que estamos usando até agora é que uma quebra de linha é impressa toda vez que chamado. Para não pular uma linha usamos o método a seguir:

```
System.out.print(variavel);
```

- 8) (opcional) Imprima a seguinte tabela, usando `for`s encadeados:

```
1
2 4
3 6 9
4 8 12 16
n n*2 n*3 . . . . n*n
```

### 3.14 - Desafios: Fibonacci

- 1) Faça o exercício da série de Fibonacci usando apenas duas variáveis.

# Orientação a objetos básica

*“Programação orientada à objetos é uma péssima idéia, que só poderia ter nascido na Califórnia.”*  
– Edsger Dijkstra

Ao término deste capítulo, você será capaz de:

- dizer o que é e para que serve orientação a objetos;
- conceituar classes, atributos e comportamentos;
- entender o significado de variáveis e objetos na memória.

## 4.1 - Motivação: problemas do paradigma procedural

Orientação a objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural.

Consideremos o clássico problema da validação de um CPF. Normalmente, temos um formulário, no qual recebemos essa informação, e depois temos que enviar esses caracteres para uma função que irá validá-lo, como no pseudo código abaixo:

```
cpf = formulario->campo_cpf  
valida(cpf)
```

Alguém te obriga a sempre validar esse CPF? Você pode, inúmeras vezes, esquecer de chamar esse validador. Mais: considere que você tem 50 formulários e precise validar em todos eles o CPF. Se sua equipe tem 3 programadores trabalhando nesses formulários, quem fica responsável por essa validação? Todos!

A situação pode piorar: na entrada de um novo desenvolvedor, precisaríamos avisá-lo que sempre devemos validar o cpf de um formulário. É nesse momento que nascem aqueles guias de programação para o desenvolvedor que for entrar nesse projeto - às vezes é um documento enorme. Em outras palavras, **todo** desenvolvedor precisa ficar sabendo de uma quantidade enorme de informações, que, na maioria das vezes, não está realmente relacionado à sua parte no sistema, mas ele **precisa** ler tudo isso, resultando um entrave muito grande!

Outra situação onde ficam claros os problemas da programação procedural, é quando nos encontramos na necessidade de ler o código que foi escrito por outro desenvolvedor e descobrir como ele funciona internamente. Um sistema bem encapsulado não deveria gerar essa necessidade. Em um sistema grande, simplesmente não temos tempo de ler uma parte grande do código.

Considerando que você não erre aí e que sua equipe tenha uma comunicação muito boa (perceba que comunicação excessiva pode ser prejudicial e atrapalhar o andamento), ainda temos outro problema: imagine que, agora, em todo formulário, você também quer que a idade do cliente seja validada - precisa ser maior de

18 anos. Vamos ter de colocar um `if...` mas onde? Espalhado por todo seu código... Mesmo que se crie outra função para validar, precisaremos incluir isso nos nossos 50 formulários já existentes. Qual é a chance de esquecermos em um deles? É muito grande.

A responsabilidade de verificar se o cliente tem ou não tem 18 anos, ficou espalhada por todo o seu código. Seria interessante poder concentrar essa responsabilidade em um lugar só, para não ter chances de esquecer isso.

Melhor ainda seria se conseguíssemos mudar essa validação e os outros programadores nem precisassem ficar sabendo disso. Em outras palavras, eles criariam formulários e um único programador seria responsável pela validação: os outros nem sabem da existência desse trecho de código. Um mundo ideal? Não, o paradigma da orientação a objetos facilita tudo isso.

O problema é que não existe uma conexão entre seus dados! Não existe uma conexão entre seus dados e suas funcionalidades! A idéia é ter essa amarra através da linguagem.

#### Quais as vantagens?

Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação, **encapsulando** a lógica de negócios.

Outra enorme vantagem, onde você realmente vai economizar montanhas de código, é o **polimorfismo das referências**, que veremos em um posterior capítulo.

## 4.2 - Criando um tipo

Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a conta. Nossa idéia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

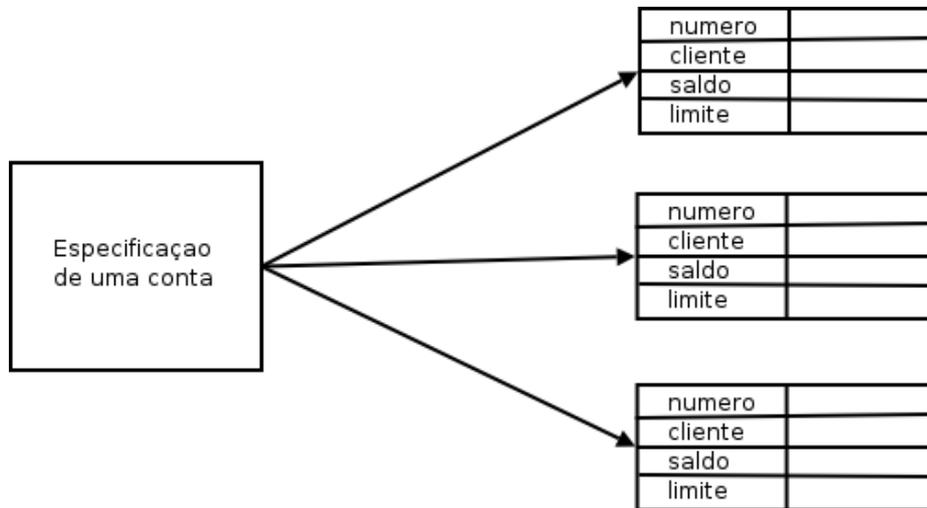
O que toda conta tem e é importante para nós?

- número da conta
- nome do cliente
- saldo
- limite

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de “pedir à conta”.

- saca uma quantidade `x`
- deposita uma quantidade `x`
- imprime o nome do dono da conta
- devolve o saldo atual
- transfere uma quantidade `x` para uma outra conta `y`
- devolve o tipo de conta

Com isso, temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o **projeto**. Antes, precisamos **construir** uma conta, para poder acessar o que ela tem, e pedir a ela que faça alguma coisa.



Repare na figura: apesar do papel do lado esquerdo especificar uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, onde podemos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como à esquerda na figura), são nas instâncias desse projeto que realmente há espaço para armazenar esses valores.

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. Ao que podemos construir a partir desse projeto, as contas de verdade, damos o nome de **objetos**.

A palavra **classe** vem da taxonomia da biologia. Todos os seres vivos de uma mesma **classe** biológica têm uma série de **atributos** e **comportamentos** em comum, mas não são iguais, podem variar nos valores desses **atributos** e como realizam esses **comportamentos**.

**Homo Sapiens** define um grupo de seres que possuem características em comum, porém a definição (a idéia, o conceito) de um **Homo Sapiens** é um ser humano? Não. Tudo está especificado na **classe** Homo Sapiens, mas se quisermos mandar alguém correr, comer, pular, precisaremos de uma instância de **Homo Sapiens**, ou então de um **objeto** do tipo **Homo Sapiens**.

Um outro exemplo: uma receita de bolo. A pergunta é certa: você come uma receita de bolo? Não. Precisamos **instaciá-la**, criar um **objeto** bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são **objetos** diferentes.

Podemos fazer milhares de analogias semelhantes. A planta de uma casa é uma casa? Definitivamente não. Não podemos morar dentro da planta de uma casa, nem podemos abrir sua porta ou pintar suas paredes. Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.

Podem parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justo saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

### 4.3 - Uma classe em Java

Vamos começar apenas com o que uma *Conta* tem, e não com o que ela faz (veremos logo em seguida).

Um tipo desses, como o especificado de *Conta* acima, pode ser facilmente traduzido para Java:

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

    // ..
}
```

#### String

String é uma classe em Java. Ela guarda uma palavra, isso é um punhado de caracteres. Como estamos aprendendo o que é uma classe, entenderemos com detalhes a classe String apenas em capítulos posteriores.

Por enquanto, declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que fazíamos quando tinha aquele main. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

### 4.4 - Criando e usando um objeto

Agora, temos uma classe em Java que especifica o que todo objeto dessa classe deve ter. Mas como usá-la? Além dessa classe, ainda teremos o **Programa.java** e a partir dele é que iremos utilizar a classe *Conta*.

Para criar (construir, instanciar) uma *Conta*, basta usar a palavra chave `new`, utilizamos também os parênteses, que descobriremos o que são, exatamente, em um capítulo posterior:

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

Bem, o código acima cria um objeto do tipo *Conta*, mas como acessar esse objeto que foi criado? Precisamos ter alguma forma de nos referenciar a esse objeto. Precisamos de uma variável:

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

Pode parecer estranho escrevermos duas vezes `Conta`: uma vez na declaração da variável e outra vez no uso do `new`. Mas há um motivo, que entenderemos também posteriormente.

Através da variável `minhaConta`, agora, podemos acessar o objeto recém criado para alterar seu `nome`, seu `saldo` etc:

```
1 class Programa {
2     public static void main(String[] args) {
3         Conta minhaConta;
4         minhaConta = new Conta();
5
6         minhaConta.nome = "Duke";
7         minhaConta.saldo = 1000.0;
8
9         System.out.println("Saldo atual: " + minhaConta.saldo);
10    }
11 }
```

É importante fixar que o *ponto* foi utilizado para acessar algo em `minhaConta`. Agora, `minhaConta` pertence ao `Duke`, e tem saldo de mil reais.

## 4.5 - Métodos

Dentro da classe, também declararemos o que cada conta faz e como isto é feito - os comportamentos que cada classe tem, isto é, o que ela faz. Por exemplo, de que maneira que uma `Conta` saca dinheiro? Especificaremos isso dentro da própria classe `Conta`, e não em um local desatrelado das informações da própria `Conta`. É por isso que essas “funções” são chamadas de **métodos**. Pois é a maneira de fazer uma operação com um objeto.

Queremos criar um método que **saca** uma determinada **quantidade** e não devolve **nenhuma informação** para quem acionar esse método:

```
1 class Conta {
2     double salario;
3     // ... outros atributos ...
4
5     void saca(double quantidade) {
6         double novoSaldo = this.saldo - quantidade;
7         this.saldo = novoSaldo;
8     }
9 }
```

A palavra chave `void` diz que, quando você pedir para a conta sacar uma quantia, nenhuma informação será enviada de volta a quem pediu.

Quando alguém pedir para sacar, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses - o que vai aí dentro é chamado de **argumento** do método (ou **parâmetro**). Essa variável é uma variável comum, chamada também de temporária ou local, pois, ao final da execução desse método, ela deixa de existir.

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu escopo. No momento que vamos acessar nosso atributo, usamos a

palavra chave `this` para mostrar que esse é um atributo, e não uma simples variável. (veremos depois que é opcional)

Repare que, nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente, evitaremos essa situação, e de uma maneira muito elegante.

Da mesma forma, temos o método para depositar alguma quantia:

```
1 class {
2     // ... outros atributos e métodos ...
3
4     void deposita(double quantidade) {
5         this.saldo += quantidade;
6     }
7 }
```

Observe que, agora, não usamos uma variável auxiliar e, além disso, usamos a abreviação `+=` para deixar o método bem simples. O `+=` soma quantidade ao valor antigo do saldo e guarda no próprio saldo, o valor resultante.

Para mandar uma mensagem ao objeto e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é **invocação de método**.

O código a seguir saca dinheiro e depois deposita outra quantia na nossa conta:

```
1 class SacaeDeposita {
2     public static void main(String[] args) {
3         // criando a conta
4         Conta minhaConta;
5         minhaConta = new Conta();
6
7         // alterando os valores de minhaConta
8         minhaConta.nome = "Duke";
9         minhaConta.saldo = 1000;
10
11        // saca 200 reais
12        minhaConta.saca(200);
13
14        // deposita 500 reais
15        minhaConta.deposita(500);
16        System.out.println(minhaConta.saldo);
17    }
18 }
```

Uma vez que seu saldo inicial é 1000 reais, se sacarmos 200 reais, depositarmos 500 reais e imprimirmos o valor do saldo, o que será impresso?

## 4.6 - Métodos com retorno

Um método sempre tem que retornar alguma coisa, nem que essa coisa seja nada, como nos exemplos anteriores onde estávamos usando o `void`.

Um método pode retornar um valor para o código que o chamou. No caso do nosso método `saca` podemos devolver um valor booleano indicando se a operação foi bem sucedida.

```
1 class Conta {
2     // ... outros metodos e atributos ...
3
4     boolean saca(double valor) {
5         if (this.saldo < valor) {
6             return false;
7         }
8         else {
9             this.saldo = this.saldo - valor;
10            return true;
11        }
12    }
13 }
```

Agora a declaração do método mudou! O método `saca` não tem `void` na frente, isto quer dizer que, quando é acessado, ele devolve algum tipo de informação. No caso, um `boolean`. A palavra chave `return` indica que o método vai terminar ali, retornando tal informação.

Conta
+numero: int
+saldo: double
+limite: double
+nome: String
+saca(valor:double): boolean
+deposita(valor:double)

Exemplo de uso:

```
minhaConta.saldo = 1000;
boolean consegui = minhaConta.saca(2000);
if(conseguir){
    System.out.println("Conseguir sacar");
}else{
    System.out.println("Não consegui sacar");
}
```

Ou então, posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
System.out.println(minhaConta.saca(2000));
```

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (exception) nesses casos.

Meu programa pode manter na memória não apenas uma conta, como mais de uma:

```
1 class TestaDuasContas {
2     public static void main(String[] args) {
3
4         Conta minhaConta;
5         minhaConta = new Conta();
```

```
6     minhaConta.saldo = 1000;
7
8
9     Conta meuSonho;
10    meuSonho = new Conta();
11    meuSonho.saldo = 1500000;
12
13 }
14 }
```

## 4.7 - Objetos são acessados por referências

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.

É por esse motivo que, diferente dos *tipos primitivos* como `int` e `long`, precisamos dar `new` depois de declarada a variável:

```
1 public static void main(String args[]) {
2     Conta c1;
3     c1 = new Conta();
4
5     Conta c2;
6     c2 = new Conta();
7 }
```

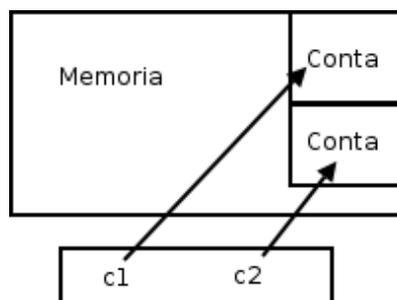
O correto aqui, é dizer que `c1` se refere a um objeto. **Não é correto** dizer que `c1` é um objeto, pois `c1` é uma variável referência, apesar de, depois de um tempo, os programadores java falem “Tenho um **objeto c** do tipo **Conta**”, mas apenas para encurtar a frase “Tenho uma **referência c** a um **objeto** do tipo **Conta**”.

Basta lembrar que, em java, **uma variável nunca é um objeto**. Não há, no java, uma maneira de criarmos o que é conhecido como “*objeto pilha*” ou “*objeto local*”, pois todo objeto em java, sem exceção, é acessado por uma variável referência.

Esse código nos deixa na seguinte situação:

```
Conta c1;
c1 = new Conta();

Conta c2;
c2 = new Conta();
```



Internamente, `c1` e `c2` vão guardar um número que identifica em que posição da memória aquela `Conta` se encontra. Dessa maneira, ao utilizarmos o “.” para navegar, o java vai acessar a `Conta` que se encontra naquela posição de memória, e não uma outra.

Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo e utilizá-lo para guardar outras coisas.

Agora vamos a um outro exemplo:

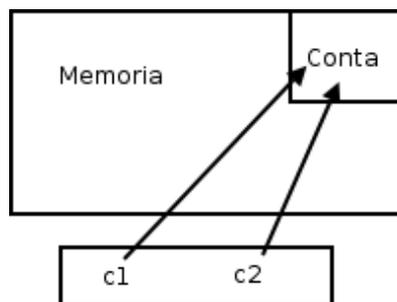
```
1 class TestaReferencias {
2     public static void main(String args[]) {
3         Conta c1 = new Conta();
4         c1.deposita(100);
5
6         Conta c2 = c1; // linha importante!
7         c2.deposita(200);
8
9         System.out.println(c1.saldo);
10        System.out.println(c2.saldo);
11    }
12 }
```

Qual é o resultado do código acima? O que aparece ao rodar?

O que acontece aqui? O operador `=` copia o valor de uma variável. Mas qual é o valor da variável `c1`? É o objeto? Não. Na verdade, o valor guardado é a referência (**endereço**) de onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

```
Conta c1 = new Conta();
Conta c2 = c1;
```



Quando fizemos `c2 = c1`, `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante.

Então, nesse código em específico, quando utilizamos `c1` ou `c2` estamos nos referindo exatamente ao **mesmo** objeto! Elas são duas referências distintas, porém apontam para o **mesmo** objeto! Compará-las com “`==`” irá nos retornar `true`, pois o valor que elas carregam é o mesmo!

Outra forma de perceber, é que demos apenas um `new`, então só pode haver um objeto `Conta` na memória.

**Atenção:** não estamos discutindo aqui a utilidade de fazer uma referência apontar pro mesmo objeto que outra. Essa utilidade ficará mais clara quando passarmos variáveis do tipo referência como argumento para métodos.

## new

O que exatamente faz o `new`?

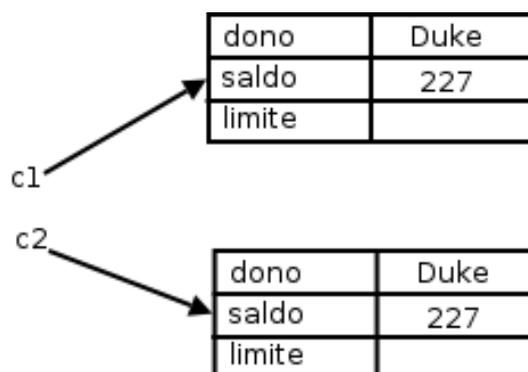
O `new` executa uma série de tarefas, que veremos mais adiante.

Mas, para melhor entender as referências no Java, saiba que o `new`, depois de alocar a memória para esse objeto, devolve uma “flecha”, isto é, um valor de referência. Quando você atribui isso a uma variável, essa variável passa a se referir para esse mesmo objeto.

Podemos então ver outra situação:

```
1 public static void main(String args[]) {
2     Conta c1 = new Conta();
3     c1.nome = "Duke";
4     c1.saldo = 227;
5
6     Conta c2 = new Conta();
7     c2.dono = "Duke";
8     c2.saldo = 227;
9
10    if (c1 == c2) {
11        System.out.println("Contas iguais");
12    }
13 }
```

O operador `==` compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, eles estão em espaços diferentes da memória, o que faz o teste no `if` valer `false`. As contas podem ser equivalentes no nosso critério de igualdade, porém elas não são o mesmo objeto. Quando se trata de objetos, pode ficar mais fácil pensar que o `==` compara se os objetos (referências, na verdade) são o mesmo, e não se são iguais.



Para saber se dois objetos têm o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

## 4.8 - O método transfere()

E se quisermos ter um método que transfere dinheiro entre duas contas? Podemos ficar tentados a criar um método que recebe dois parâmetros: `conta1` e `conta2` do tipo `Conta`. Mas cuidado: assim estamos pensando de maneira procedural.

A idéia é que, quando chamarmos o método `transfere`, já teremos um objeto do tipo `Conta` (o `this`), portanto o método recebe apenas **um** parâmetro do tipo `Conta`, a `Conta destino` (além do `valor`):

```
class Conta {  
  
    // atributos e metodos...  
  
    void transfere(Conta destino, double valor) {  
        this.saldo = this.saldo - valor;  
        destino.saldo = destino.saldo + valor;  
    }  
}
```

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor:double): boolean +deposita(valor:double) +transfere(destino:Conta, valor:double)

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos `deposita` e `saca` já existentes para fazer essa tarefa:

```
class Conta {  
  
    // atributos e metodos...  
  
    boolean transfere(Conta destino, double valor) {  
        boolean retirou = this.saca(valor);  
        if (retirou == false) {  
            // não deu pra sacar!  
            return false;  
        }  
        else {  
            destino.deposita(valor);  
            return true;  
        }  
    }  
}
```

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor:double): boolean +deposita(valor:double) +transfere(destino:Conta, valor:double): boolean

Quando passamos uma `Conta` como argumento, o que será que acontece na memória? Será que o objeto é *clonado*?

No java, a passagem de parâmetro funciona como uma simples atribuição como no uso do “=”. Então, esse parâmetro vai copiar o valor da variável do tipo `Conta` que for passado como argumento. E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

Esse último código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

#### Transfere Para

Perceba que o nome deste método poderia ser `transferePara` ao invés de só `transfere`. A chamada do método fica muito mais natural, é possível ler a frase em português que ela tem um sentido:

```
conta1.transferePara(conta2, 50);
```

A leitura deste código seria “Conta1 transfere para conta2 50 reais”.

## 4.9 - Continuando com atributos

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem 0, no caso de `boolean`, valem `false`.

Você também pode dar **valores default**, como segue:

```
1 class Conta {
2     int numero = 1234;
3     String dono = "Duke";
4     String cpf = "123.456.789-10";
5     double saldo = 1000;
6     double limite = 1000;
7 }
```

Nesse caso, quando você criar um carro, seus atributos já estão “populados” com esses valores colocados.

Imagine que, agora, começamos a aumentar nossa classe `Conta` e adicionar `nome`, `sobrenome` e `cpf` do cliente dono da conta. Começaríamos a ter muitos atributos... e, se você pensar direito, uma `Conta` não tem `nome`, nem `sobrenome` nem `cpf`, quem tem esses atributos é um `Cliente`. Então podemos criar uma nova classe e fazer uma composição

Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe `Cliente`:

```
1 class Cliente {
2     String nome;
3     String sobrenome;
4     String cpf;
5 }
```

```
1 class Conta {
2     int numero;
3     double saldo;
4     double limite;
```

```
5     Cliente titular;  
6     // ..  
7 }
```

E dentro do main da classe de teste:

```
1 class Teste {  
2     public static void main(String[] args) {  
3         Conta minhaConta = new Conta();  
4         Cliente c = new Cliente();  
5         minhaConta.titular = c;  
6         // ...  
7     }  
8 }
```

Aqui, simplesmente houve uma atribuição. O valor da variável `c` é copiado para o atributo titular do objeto ao qual `minhaConta` se refere. Em outras palavras, `minhaConta` agora tem uma referência ao mesmo `Cliente` que `c` se refere, e pode ser acessado através de `minhaConta.titular`.

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.titular;  
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda, pode fazer isso de uma forma mais direta e até mais elegante:

```
minhaConta.titular.nome = "Duke";
```

Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe `Banco` usa a classe `Conta` que usa a classe `Cliente`, que usa a classe `Endereco`. Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

Mas, e se dentro do meu código eu não desse `new` em `Cliente` e tentasse acessá-lo diretamente?

```
class Teste {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
  
        minhaConta.titular.nome = "paulo";  
        // ...  
    }  
}
```

Quando damos `new` em um objeto, ele o inicializa com seus valores default, 0 para números, `false` para boolean e `null` para referências. `null` é uma palavra chave em java, que indica uma referência para nenhum objeto.

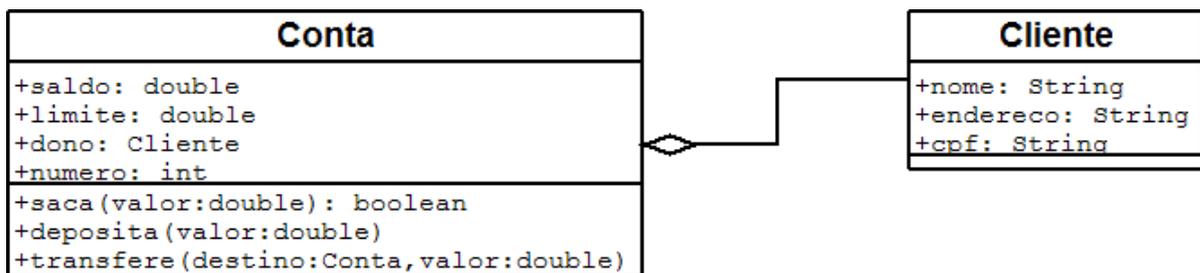


Se, em algum caso, você tentar acessar um atributo ou método de alguém que está se referenciando para null, você receberá um erro durante a execução (NullPointerException, que veremos mais à frente). Da para perceber, então, que o new não traz um efeito cascata, a menos que você dê um valor default (ou use construtores, que também veremos mais a frente):

```

1 class Conta {
2     int numero;
3     double saldo;
4     double limite;
5     Cliente titular = new Cliente();    // quando chamarem new Conta,
6                                         //havera um new Cliente para ele.
7 }

```



Com esse código, toda nova Conta criada criado já terá um novo Cliente associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma Conta. Qual alternativa você deve usar? Depende do caso: para toda nova Conta você precisa de um novo Cliente? É essa pergunta que deve ser respondida. Nesse nosso caso a resposta é não, mas depende do nosso problema.

**Atenção:** para quem não está acostumado com referências, pode ser bastante confuso pensar sempre em como os objetos estão na memória para poder tirar as conclusões de o que ocorrerá ao executar determinado código, por mais simples que ele seja. Com tempo, você adquire a habilidade de rapidamente saber o efeito de atrelar as referências, sem ter de gastar muito tempo para isso. É importante, nesse começo, você estar sempre pensando no estado da memória. E realmente lembrar que, no Java *“uma variável nunca carrega um objeto, e sim uma referência para ele”* facilita muito.

## 4.10 - Para saber mais: Uma Fábrica de Carros

Além do Banco que estamos criando, vamos ver como ficariam certas classes relacionadas a uma fábrica de carros. Vamos criar uma classe Carro, com certos atributos, que descrevem suas características, e com certos métodos, que descrevem seu comportamento.

```

1 class Carro {
2     String cor;
3     String modelo;
4     double velocidadeAtual;

```

```
5     double velocidadeMaxima;
6
7     //liga o carro
8     void liga() {
9         System.out.println("O carro está ligado");
10    }
11
12    //acelera uma certa quantidade
13    void acelera(double quantidade) {
14        double velocidadeNova = this.velocidadeAtual + quantidade;
15        this.velocidadeAtual = velocidadeNova;
16    }
17
18    //devolve a marcha do carro
19    int pegaMarcha() {
20        if (this.velocidadeAtual < 0) {
21            return -1;
22        }
23        if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
24            return 1;
25        }
26        if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80) {
27            return 2;
28        }
29        return 3;
30    }
31 }
```

Agora, vamos testar nosso Carro em um programa de testes:

```
1 class TestaCarro {
2     public static void main(String[] args) {
3         Carro meuCarro;
4         meuCarro = new Carro();
5         meuCarro.cor = "Verde";
6         meuCarro.modelo = "Fusca";
7         meuCarro.velocidadeAtual = 0;
8         meuCarro.velocidadeMaxima = 80;
9
10        // liga o carro
11        meuCarro.liga();
12
13        // acelera o carro
14        meuCarro.acelera(20);
15        System.out.println(meuCarro.velocidadeAtual);
16    }
17 }
```

Nosso carro pode conter também um Motor:

```
1 class Motor {
2     int potencia;
3     String tipo;
4 }
```

```
1 class Carro {
2     String cor;
3     String modelo;
4     double velocidadeAtual;
5     double velocidadeMaxima;
6     Motor motor;
7
8     // ..
9 }
```

Podemos, agora, criar diversos Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do Banco.

### 4.11 - Um pouco mais...

- 1) Quando declaramos uma classe, um método ou um atributo, podemos dar o nome que quisermos, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.
- 2) Como você pode ter reparado, sempre damos nomes às variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Sun, para facilitar a legibilidade do código entre programadores. Essa convenção é *muito seguida*. Pesquise sobre ela no <http://java.sun.com>, procure por “code conventions”.
- 3) É necessário usar a palavra chave `this` quando for acessar um atributo? Para que, então, utilizá-la?
- 4) O exercício a seguir pedirá para modelar um “funcionário”. Existe um padrão para representar suas classes em diagramas, que é amplamente utilizado, chamado **UML**. Pesquise sobre ele.

### 4.12 - Exercícios: Orientação a Objetos

O modelo de funcionários a seguir será utilizado para os exercícios de alguns dos posteriores capítulos.

O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco. Os exercícios desse capítulo são extremamente importantes.

- 1) Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (`double`), a data de entrada no banco (`String`), seu RG (`String`) e um valor booleano que indique se o funcionário está na empresa no momento ou se já foi embora.

Você deve criar alguns métodos de acordo com sua necessidade. Além deles, crie um método `bonifica` que aumenta o `salario` do funcionário de acordo com o parâmetro passado como argumento. Crie, também, um método `demite`, que não recebe parâmetro algum, só modifica o valor booleano indicando que o funcionário não trabalha mais aqui.

A idéia aqui é apenas modelar, isto é, só identifique que informações são importantes e o que um funcionário faz. Desenhe no papel tudo o que um `Funcionario` tem e tudo que ele faz.

- 2) Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main`. Você deve criar a classe do funcionário chamada `Funcionario`, e a classe de teste você pode nomear como quiser. A de teste deve possuir o método `main`.

Um esboço da classe:

```
class Funcionario {  
  
    double salario;  
    // seus outros atributos e métodos  
  
    void bonifica(double aumento) {  
        // o que fazer aqui dentro?  
    }  
  
    void demite() {  
        // o que fazer aqui dentro?  
    }  
}
```

Você pode (e deve) compilar seu arquivo java sem que você ainda tenha terminado sua classe `Funcionario`. Isso evitará que você receba dezenas de erros de compilação de uma vez só. Crie a classe `Funcionario`, coloque seus atributos e, antes de colocar qualquer método, compile o arquivo java. `Funcionario.class` será gerado, não podemos “executá-la” pois não há um `main`, mas assim verificamos que nossa classe `Funcionario` já está tomando forma.

Funcionario
+nome: String
+departamento: String
+salario: double
+dataEntrada: String
+trg: String
+testaNaEmpresa: boolean
+bonifica(aumento:double)
+demite()

Esse é um processo incremental. Procure desenvolver assim seus exercícios, para não descobrir só no fim do caminho que algo estava muito errado.

Um esboço da classe que possui o `main`:

```
1 class TestaFuncionario {  
2  
3     public static void main(String[] args) {  
4         Funcionario f1 = new Funcionario();  
5  
6         f1.nome = "Fiodor";  
7         f1.salario = 100;  
8         f1.bonifica(50);  
9  
10        System.out.println("salario atual:" + f1.salario);  
11  
12    }  
13 }
```

Incremente essa classe. Faça outros testes, imprima outros atributos e invoque os métodos que você criou a mais.

Lembre-se de seguir a convenção java, isso é importantíssimo. Isto é, `nomeDeAtributo`, `nomeDeMetodo`, `nomeDeVariavel`, `NomeDeClasse`, etc...

**Todas as classes no mesmo arquivo?**

Por enquanto, você pode colocar todas as classes no mesmo arquivo e apenas compilar esse arquivo. Ele vai gerar os dois `.class`.

Porém, é boa prática criar um arquivo `.java` para cada classe e, em determinados casos, você será obrigado a declarar uma classe em um arquivo separado, como veremos no capítulo 10. Isto não é importante para o aprendizado no momento.

- 3) Crie um método `mostra()`, que não recebe nem devolve parâmetro algum e simplesmente imprime todos os atributos do nosso funcionário. Dessa maneira, você não precisa ficar copiando e colando um monte de `System.out.println()` para cada mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

```
Funcionario f1 = new Funcionario();
//brincadeiras com f1...
f1.mostra();
```

Veremos mais a frente o método `toString`, que é uma solução muito mais elegante para mostrar a representação de um objeto como `String`, além de não jogar tudo pro `System.out` (só se você desejar).

O esqueleto do método ficaria assim:

```
class Funcionario {

    // seus outros atributos e métodos

    void mostra() {
        System.out.println("Nome: " + this.nome);
        // imprimir aqui os outros atributos...
    }
}
```

- 4) Construa dois funcionários com o `new` e compare-os com o `==`. E se eles tiverem os mesmos atributos? Para isso você vai precisar criar outra referência:

```
Funcionario f1 = new Funcionario();
f1.nome = "Fiodor";
f1.salario = 100;

Funcionario f2 = new Funcionario();
f2.nome = "Fiodor";
f2.salario = 100;

if (f1 == f2) {
    System.out.println("iguais");
} else {
    System.out.println("diferentes");
}
```

- 5) Crie duas referências para o **mesmo** funcionário, compare-os com o `==`. Tire suas conclusões. Para criar duas referências pro mesmo funcionário:

```
Funcionario f1 = new Funcionario();
```

```
f1.nome = "Fiodor";  
f1.salario = 100;
```

```
Funcionario f2 = f1;
```

O que acontece com o `if` do exercício anterior?

- 6) (opcional) Em vez de utilizar uma `String` para representar a data, crie uma outra classe, chamada `Data`. Ela possui 3 campos `int`, para dia, mês e ano. Faça com que seu funcionário passe a usá-la. (é parecido com o último exemplo, em que a `Conta` passou a ter referência para um `Cliente`).

```
class Funcionario {  
  
    Data dataDeEntrada; // qual é o valor default aqui?  
    // seus outros atributos e métodos  
  
}  
  
class Data {  
  
    int dia;  
    int mes;  
    int ano;  
  
}
```

Modifique sua classe `TestaFuncionario` para que você crie uma `Data` e atribua ela ao `Funcionario`:

```
Funcionario f1 = new Funcionario();  
//...  
Data data = new Data(); // ligação!  
f1.dataDeEntrada = data;
```

Faça o desenho do estado da memória quando criarmos um `Funcionario`.

- 7) (opcional) Modifique seu método `mostra` para que ele imprima o valor da `dataDeEntrada` daquele `Funcionario`:

```
class Funcionario {  
  
    // seus outros atributos e métodos  
    Data dataDeEntrada;  
  
    void mostra() {  
        System.out.println("Nome: " + this.nome);  
        // imprimir aqui os outros atributos...  
  
        System.out.println("Dia: " + this.dataDeEntrada.dia);  
        System.out.println("Mês: " + this.dataDeEntrada.mes);  
        System.out.println("Ano: " + this.dataDeEntrada.ano);  
    }  
  
}
```

Teste-o.

Agora, o que acontece se chamarmos o método `mostra` antes de atribuirmos uma data para este `Funcionario`?

- 8) (opcional) O que acontece se você tentar acessar um atributo diretamente na classe? Como, por exemplo:

```
Conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
Conta.saca(50);
```

Faz sentido pedir para o esquema do conta sacar uma quantia?

## 4.13 - Desafios

- 1) Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

```
Fibonacci fibo = new Fibonacci();  
int i = fibo.calculaFibonacci(5);  
System.out.println(i);
```

Aqui imprimirá 8, já que este é o sexto número da série.

Este método `calculaFibonacci` não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

- 2) Por que o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?
- 3) Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

## 4.14 - Fixando o conhecimento

O objetivo dos exercícios a seguir é fixar o conceito de classes e objetos, métodos e atributos. Dada a estrutura de uma classe, basta traduzí-la para a linguagem Java e fazer uso de um objeto da mesma em um programa simples.

Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora nos pequenos programas abaixo:

- 1) Programa 1

Classe: Pessoa Atributos: nome, idade. Método: void fazAniversario()

Crie uma pessoa, coloque seu nome e idade iniciais, faça alguns aniversários (aumentando a idade) e imprima seu nome e sua idade.

- 2) Programa 2

Classe: Porta Atributos: aberta, cor, dimensaoX, dimensaoY, dimensaoZ Métodos: void abre(), void fecha(), void pinta(String s), boolean estaAberta()

Crie uma porta, abra e feche a mesma, pinte-a de diversas cores, altere suas dimensões e use o método estaAberta para verificar se ela está aberta.

### 3) Programa 3

Classe: Casa Atributos: cor, porta1, porta2, porta3 Método: void pinta(String s), int quantasPortasEstaoAbertas()

Crie uma casa e pinte-a. Crie três portas e coloque-as na casa; abra e feche as mesmas como desejar. Utilize o método quantasPortasEstaoAbertas para imprimir o número de portas abertas.

# Um pouco de arrays

*“O homem esquecerá antes a morte do pai que a perda da propriedade”*  
– Maquiavel

Ao término desse capítulo, você será capaz de:

- declarar e instanciar arrays;
- popular e percorrer arrays.

## 5.1 - O problema

Dentro de um bloco, podemos declarar variáveis e usá-las.

```
int idade1;  
int idade2;  
int idade3;  
int idade4;
```

Mas também podemos declarar um **vetor (array)** de inteiros:

```
int[] idades;
```

O `int[]` é um tipo. Uma array é sempre um objeto, portanto, a variável `idades` é uma referência. Vamos precisar criar um objeto para poder usar a array. Como criamos o objeto-array?

```
idades = new int[10];
```

Aqui, o que fizemos foi criar uma array de `int` de 10 posições, e atribuir o endereço no qual ela foi criada. Agora, podemos acessar as posições do array.

```
idades[5] = 10;
```



O código acima altera a sexta posição do array. No Java, os índices do array vão de 0 a n-1, onde n é o tamanho dado no momento em que você criou o array. Se você tentar acessar uma posição fora desse alcance, um erro ocorrerá durante a execução.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at ArrayIndexOutOfBoundsExceptionTeste.main(ArrayIndexOutOfBoundsExceptionTeste.java:5)
```

#### Arrays – um problema no aprendizado de muitas linguagens

Aprender a usar arrays pode ser um problema em qualquer linguagem. Isso porque envolve uma série de conceitos, sintaxe e outros. No Java, muitas vezes utilizamos outros recursos em vez de arrays, em especial os pacotes de coleções do Java, que veremos no capítulo 11. Portanto, fique tranqüilo caso não consiga digerir toda sintaxe das arrays num primeiro momento.

## 5.2 - Arrays de referências

É comum ouvirmos “array de objetos”. Porém quando criamos uma array de alguma classe, ela possui referências. O objeto, como sempre, está na memória principal e, na sua array, só ficam guardadas as **referências** (endereços).

```
Conta[] minhasContas;  
minhasContas = new Conta[10];
```

Quantas contas foram criadas aqui? Na verdade, **nenhuma**. Foram criados 10 espaços que você pode utilizar para guardar uma referência a uma Conta. Por enquanto, eles se referenciam para lugar nenhum (`null`). Se você tentar:

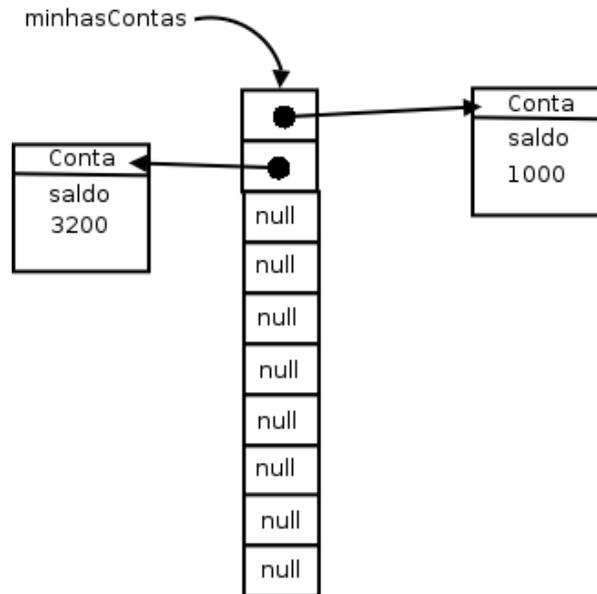
```
System.out.println(minhasContas[0].saldo);
```

Um erro durante a execução ocorrerá! Pois, na primeira posição da array, não há uma referência para uma conta, nem para lugar nenhum. Você deve **popular** sua array antes.

```
Conta contaNova = new Conta();  
contaNova.saldo = 1000.0;  
minhasContas[0] = contaNova;
```

Ou você pode fazer isso diretamente:

```
minhasContas[1] = new Conta();  
minhasContas[1].saldo = 3200.0;
```



Uma array de tipos primitivos guarda valores, uma array de objetos guarda referências.

### 5.3 - Percorrendo uma array

Percorrer uma array é muito simples quando fomos nós que a criamos:

```
public static void main(String args[]) {
    int[] idades = new int[10];
    for (int i = 0; i < 10; i++) {
        idades[i] = i * 10;
    }
    for (int i = 0; i < 10; i++) {
        System.out.println(idades[i]);
    }
}
```

Porém, em muitos casos, recebemos uma array como argumento em um método:

```
void imprimeArray(int[] array) {
    // não compila!!
    for (int i = 0; i < ???; i++) {
        System.out.println(array[i]);
    }
}
```

Até onde o for deve ir? Toda array em Java tem um atributo que se chama `length`, e você pode acessá-lo para saber o tamanho do array ao qual você está se referenciando naquele momento:

```
void imprimeArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```

#### Arrays não podem mudar de tamanho

A partir do momento que uma array foi criada, ela **não pode** mudar de tamanho.

Se você precisar de mais espaço, será necessário criar uma nova array e, antes de se referir ela, copie os elementos da array velha.

## 5.4 - Percorrendo uma array no Java 5.0

O Java 5.0 traz uma nova sintaxe para percorrermos arrays (e coleções, que veremos mais a frente).

No caso de você não ter necessidade de manter uma variável com o índice que indica a posição do elemento no vetor, podemos usar o **enhanced-for**.

```
public static void main(String args[]) {
    int[] idades = new int[10];
    for (int i = 0; i < 10; i++) {
        idades[i] = i * 10;
    }
    for (int x : idades) {
        System.out.println(x);
    }
}
```

E agora nem precisamos mais do `length` para percorrer matrizes cujo tamanho não conhecemos:

```
void imprimeArray(int[] array) {
    for (int x : array) {
        System.out.println(x);
    }
}
```

## 5.5 - Exercícios: Arrays

- 1) Volte ao nosso sistema de Funcionario e crie uma classe `Empresa`. A `Empresa` tem um nome, `cnj` e uma array de `Funcionario`, além de outros atributos que você julgar necessário.

```
class Empresa {
    // outros atributos
    Funcionario[] funcionarios;
    String cnj;
}
```

- 2) A empresa deve ter um método `adiciona`, que recebe uma referência a `Funcionario` como argumento, e guarda esse funcionário. Algo como:

```
...
    void adiciona(Funcionario f) {
        // algo tipo: this.funcionarios[ ??? ] = f
        // mas que posição colocar?
    }
...

```

Você deve inserir o `Funcionario` em uma posição da array que esteja livre. Existem várias maneiras para você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma posição vazia toda vez. O que seria mais interessante?

É importante reparar que o método `adiciona` não recebe nome, `rg`, salário, etc. Essa seria uma maneira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria um `Funcionario` e já passa a referência dele, que dentro do objeto possui `rg`, salário, etc.

- 3) Crie uma outra classe que possuirá o seu método `main`. Dentro dele crie algumas instâncias de `Funcionario` e passe para a empresa pelo método `adiciona`. Repare que antes você vai precisar criar a array, pois inicialmente o atributo `funcionarios` da classe `Empresa` não referencia lugar nenhum (`null`):

```
Empresa empresa = new Empresa();
empresa.funcionarios = new Funcionario[10];
//      ....
```

Ou você pode construir a array dentro da própria declaração da classe `Empresa`.

Crie alguns funcionários e passe como argumento para o `adiciona` da empresa:

```
Funcionario f1 = new Funcionario();
f1.salario = 1000;
empresa.adiciona(f1);
```

Você pode criar esses funcionários dentro de um loop, se preferir.

Opcional: o método `adiciona` pode gerar uma mensagem de erro indicando que o array está cheio.

- 4) Percorra o atributo `funcionarios` da sua instância da `Empresa` e imprima o salários de todos seus funcionários.

Ou você pode chamar o método `mostra()` de cada `Funcionario` da sua array.

Use também o `for` novo do java 5.0.

Cuidado: alguns índices do seu array podem não conter referência para um `Funcionario` construído, isto é, ainda se referirem para `null`.

- 5) (Opcional) Crie um método para verificar se um determinado `Funcionario` se encontra ou não como funcionário desta empresa:

```
boolean contem(Funcionario f) {
    // ...
}
```

Você vai precisar fazer um `for` na sua array e verificar se a referência passada como argumento se encontra dentro da array. Evite ao máximo usar números hard-coded, isto é, use o `.length`.

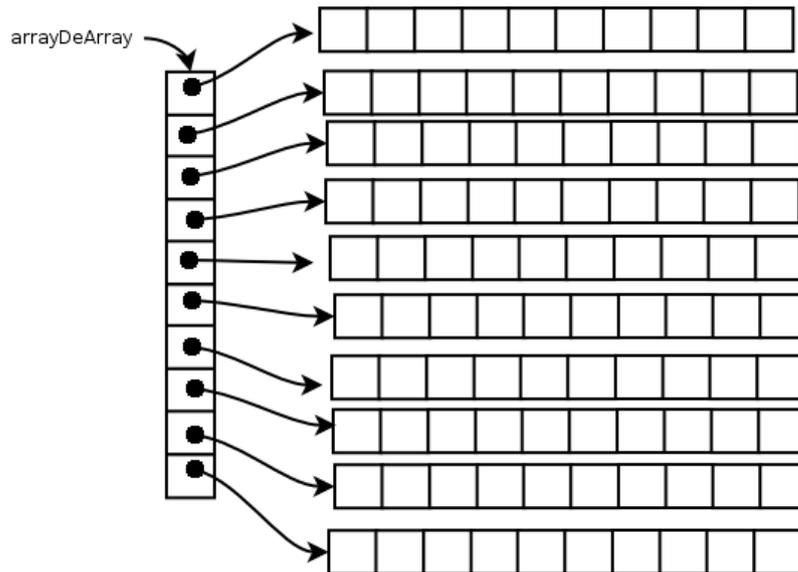
- 6) (Opcional) Caso a array já esteja cheia no momento de adicionar um outro funcionário, criar uma nova maior e copiar os valores. Isto é, fazer a realocação já que java não tem isso: uma array nasce e morre com o mesmo `length`.

#### Usando o `this` para passar argumento

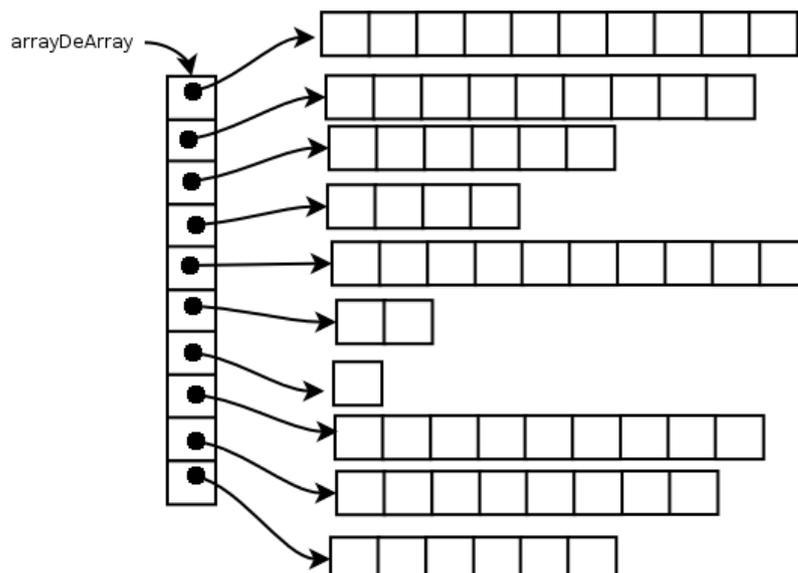
Dentro de um método, você pode usar a palavra `this` para referenciar a si mesmo e pode passar essa referência como argumento.

## 5.6 - Um pouco mais...

- Arrays podem ter mais de uma dimensão. Isto é, em vez de termos uma array de 10 contas, podemos ter uma array de 10 por 10 contas e você pode acessar a conta na posição da coluna x e linha y. Na verdade, uma array bidimensional em Java é uma array de arrays. Pesquise sobre isso.



- Uma array bidimensional não precisa ser retangular, isto é, cada linha pode ter um número diferente de colunas. Como? Porque?



- O que acontece se criarmos uma array de 0 elementos? e -1?
- O método `main` recebe uma **array de Strings** como argumento. Essa array é passada pelo usuário quando ele invoca o programa:

```
$ java Teste argumento1 outro maisoutro
```

E nossa classe:

```
class Teste {
    public static void main (String[] args) {
        for(String argumento: args) {
            System.out.println(argumento);
        }
    }
}
```

Isso imprimirá:

```
argumento1
outro
maisoutro
```

## 5.7 - Desafios

- 1) No capítulo anterior, você deve ter reparado que a versão recursiva para o problema de Fibonacci é lenta porque toda hora estamos recalculando valores. Faça com que a versão recursiva seja tão boa quanto a versão iterativa. (Dica: use arrays para isso)

## 5.8 - Testando o conhecimento

- 1) O objetivo dos exercícios a seguir é fixar os conceitos vistos até agora. Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora no pequeno programa abaixo:

- Programa:

Classe: Casa Atributos: cor, totalDePortas, portas[] Métodos: void pinta(String s), int quantasPortasEstaoAbertas(), void adicionaPorta(Porta p), int totalDePortas()

Crie uma casa, pinte-a. Crie três portas e coloque-as na casa através do método `adicionaPorta`, abra e feche-as como desejar. Utilize o método `quantasPortasEstaoAbertas` para imprimir o número de portas abertas e o método `totalDePortas` para imprimir o total de portas em sua casa.

# Modificadores de acesso e atributos de classe

*“A marca do homem imaturo é que ele quer morrer nobremente por uma causa, enquanto a marca do homem maduro é querer viver modestamente por uma.”*  
– J. D. Salinger

Ao término desse capítulo, você será capaz de:

- controlar o acesso aos seus métodos, atributos e construtores através dos modificadores `private` e `public`;
- escrever métodos de acesso a atributos do tipo getters e setters;
- escrever construtores para suas classes;
- utilizar variáveis e métodos estáticos.

## 6.1 - Controlando o acesso

Um dos problemas mais simples que temos no nosso sistema de contas é que a função `saca` permite sacar mesmo que o limite tenha sido atingido. A seguir você pode lembrar como está a classe `Conta`:

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;

    // ..

    void saca(double quantidade) {
        this.saldo = this.saldo - quantidade;
    }
}
```

A classe a seguir mostra como é possível ultrapassar o limite usando o método `saca`:

```
class TestaContaEstouro1 {
    public static void main(String args[]) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 1000.0;
        minhaConta.limite = 1000.0;
        minhaConta.saca(50000); // saldo + limite é só 2000!!
    }
}
```

```
}  
}
```

Podemos incluir um `if` dentro do nosso método `saca()` para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo abaixo do limite. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir ultrapassa o limite diretamente:

```
class TestaContaEstouro2 {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        minhaConta.limite = 100;  
        minhaConta.saldo = -200; //saldo está abaixo dos 100 de limite  
    }  
}
```

Como evitar isso? Uma idéia simples seria testar se não estamos ultrapassando o limite toda vez que formos alterar o saldo:

```
class TestaContaEstouro3 {  
  
    public static void main(String args[]) {  
        // a Conta  
        Conta minhaConta = new Conta();  
        minhaConta.limite = 100;  
        minhaConta.saldo = 100;  
  
        // quero mudar o saldo para -200  
        double novoSaldo = -200;  
  
        // testa se o novoSaldo ultrapassa o limite da conta  
        if (novoSaldo < -minhaConta.limite) { //  
            System.out.println("Não posso mudar para esse saldo");  
        } else {  
            minhaConta.saldo = novoSaldo;  
        }  
    }  
}
```

Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe `Conta` a chamar o método `saca` e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

Para fazer isso no Java, basta declarar que os atributos não podem ser acessados de fora da classe usando a palavra chave `private`:

```
class Conta {  
    private double saldo;  
    private double limite;
```

```
} // ...  
}
```

`private` é um **modificador de acesso** (também chamado de **modificador de visibilidade**).

Marcando um atributo como privado, fechamos o acesso ao mesmo de todas as outras classes, fazendo com que o seguinte código não compile:

```
class TestaAcessoDireto {  
    public static void main(String args[]) {  
        Conta minhaConta = new Conta();  
        //não compila! você não pode acessar o atributo privado de outra classe  
        minhaConta.saldo = 1000;  
    }  
}
```

```
TestaAcessoDireto.java:5: saldo has private access in Conta  
        minhaConta.saldo = 1000;  
                        ^  
1 error
```

Enquanto programando orientado a objetos, é prática quase que obrigatória proteger seus atributos com `private`. (discutiremos outros modificadores de acesso em outros capítulos).

Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não! Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema.

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

A palavra chave `private` também pode ser usada para modificar o acesso a um método. Tal funcionalidade é normalmente usada quando existe um método apenas auxiliar a própria classe e não queremos que outras pessoas o usem (ou apenas para seguir a boa prática de expôr-se ao mínimo). Sempre devemos expôr o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

Da mesma maneira que temos o `private`, temos o modificador `public`, que permite a todos acessarem um determinado atributo ou método :

```
class Conta {  
    //...  
    public void saca(double quantidade) {  
        if (quantidade > this.saldo + this.limite){ //posso sacar até saldo+limite  
            System.out.println("Não posso sacar fora do limite!");  
        } else {  
            this.saldo = this.saldo - quantidade;  
        }  
    }  
}
```

**E quando não há modificador de acesso?**

Até agora, tínhamos declarado variáveis e métodos sem nenhum modificador como `private` e `public`. Quando isto acontece, o seu método ou atributo fica num estado de visibilidade intermediário entre o `private` e o `public`, que veremos mais pra frente, no capítulo de pacotes.

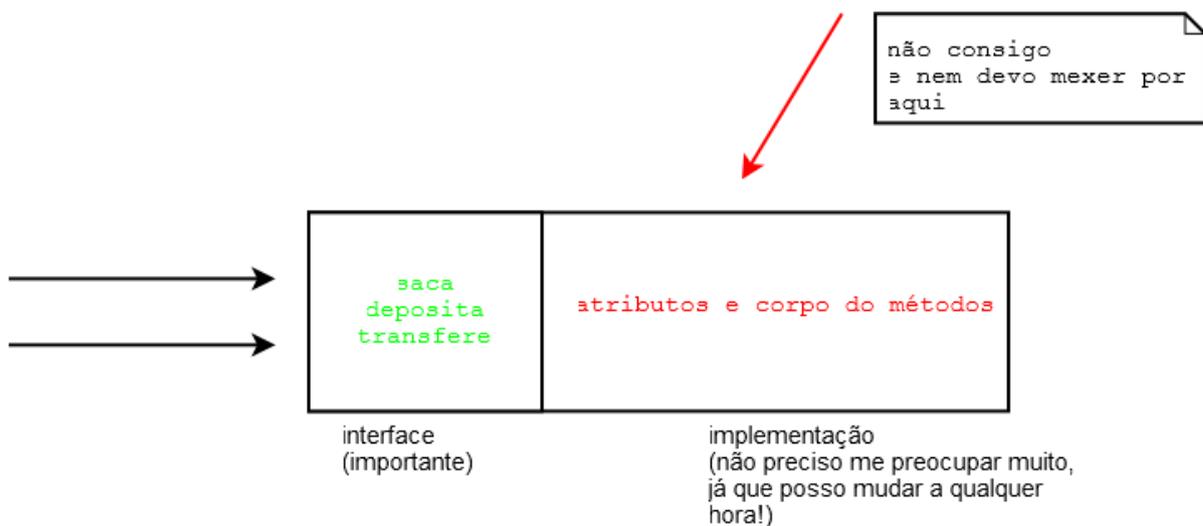
É muito comum, e faz todo sentido, que seus atributos sejam `private` e quase todos seus métodos sejam `public` (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!

Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca`, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque: basta você modificar ali, e nenhum outro código, fora a classe `Conta`, precisará ser recompilado. Mais: as classes que usam esse método nem precisam ficar sabendo de tal modificação! Você precisa apenas recompilar aquela classe e substituir aquele arquivo `.class`.

## 6.2 - Encapsulamento

O que começamos a ver nesse capítulo é a idéia de **encapsular**, isto é, esconder todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**. (veja o caso do método `saca`)



O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

### Programando voltado para a interface e não para a implementação

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela irá funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe, uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz, pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma et al. Um livro cultuado no meio da orientação a objetos.

Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

- Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas o **que ele faz** é a mesma coisa que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).
- Todos os celulares fazem a mesma coisa (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem (implementação), mas repare que para o usuário comum pouco importa se o celular é GSM ou CDMA, isso fica encapsulado na implementação (que aqui são os circuitos).

Repare que, agora, temos conhecimentos suficientes para resolver aquele problema da validação de CPF:

```
class Cliente {
    private String nome;
    private String endereco;
    private String cpf;
    private int idade;

    public void mudaCPF(String cpf) {
        validaCPF(cpf);
        this.cpf = cpf;
    }

    private void validaCPF(String cpf) {
        // série de regras aqui, falha caso nao seja válido
    }

    // ..
}
```

Agora, se alguém tentar criar um `Cliente` e não usar o `mudaCPF`, vai receber um erro de compilação, já que o atributo `CPF` é **privado**. E o dia que você não precisar validar quem tem mais de 60 anos? Seu método fica o seguinte:

```
public void mudaCPF(String cpf) {
    if (this.idade <= 60) {
        validaCPF(cpf);
    }
}
```

```
    }  
    this.cpf = cpf;  
}
```

O controle sobre o CPF está centralizado: ninguém consegue acessá-lo sem passar por aí, a classe `Cliente` é a única responsável pelos seus próprios atributos!

## 6.3 - Getters e Setters

Para permitir o acesso aos atributos (já que eles são `private`) de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor.

O padrão para esses métodos é de colocar a palavra `get` ou `set` antes do nome do atributo. Por exemplo, a nossa conta com `saldo`, `limite` e `titular` fica assim:

```
public class Conta {  
  
    private double saldo;  
    private double limite;  
    private Cliente titular;  
  
    public double getSaldo() {  
        return this.saldo;  
    }  
  
    public void setSaldo(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public double getLimite() {  
        return this.limite;  
    }  
  
    public void setLimite(double limite) {  
        this.limite = limite;  
    }  
  
    public Cliente getTitular() {  
        return this.titular;  
    }  
  
    public void setTitular(Cliente titular) {  
        this.titular = titular;  
    }  
}
```

É uma má prática criar uma classe e, logo em seguida, criar getters e setters pros seus atributos. Você só deve criar um getter ou setter se tiver a real necessidade. Repare que nesse exemplo `setSaldo` não deveria ter sido criado, já que queremos que todos usem `deposita()` e `saca()`.

Outro detalhe importante, um método `getX` não necessariamente retorna o valor de um atributo que chama `x` do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação: queremos que o banco sempre mostre como `saldo` o valor do limite somado ao saldo (uma prática comum dos bancos que costuma

iludir seus clientes). Poderíamos sempre chamar `c.getLimite() + c.getSaldo()`, mas isso poderia gerar uma situação de “replace all” quando precisássemos mudar como o saldo é mostrado. Podemos encapsular isso em um método e, porque não, dentro do próprio `getSaldo`? Repare:

```
public class Conta {  
  
    private double saldo;  
    private double limite;  
    private Cliente titular;  
  
    private double getSaldo() {  
        return this.saldo + this.limite;  
    }  
  
    // deposita() e saca()  
  
    public Cliente getTitular() {  
        return this.titular;  
    }  
  
    public void setTitular(Cliente titular) {  
        this.titular = titular;  
    }  
}
```

O código acima nem possibilita a chamada do método `getLimite()`, ele não existe. E nem deve existir enquanto não houver essa necessidade. O método `getSaldo` não devolve simplesmente o `saldo...` e sim o que queremos que seja mostrado como se fosse o `saldo`. Utilizar getters e setters não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

Nossa classe está agora totalmente pronta? Isto é, existe a chance dela ficar com menos dinheiro do que o limite? Pode parecer que não, mas, e se depositarmos um valor negativo na conta? Ficaríamos com menos dinheiro que o permitido, já que não esperávamos por isso. Para nos proteger disso basta mudarmos o método `deposita()` para que ele verifique se o valor é necessariamente positivo.

Depois disso precisaríamos mudar mais algum outro código? A resposta é não, graças ao encapsulamento dos nossos dados.

#### Cuidado com os getters e setters!

Como já dito, não devemos criar getters e setters sem um motivo explícito. No blog da Caelum há um artigo que ilustra bem esses casos:

<http://blog.caelum.com.br/2006/09/14/nao-aprender-oo-getters-e-setters/>

## 6.4 - Construtores

Quando usamos a palavra chave `new`, estamos construindo um objeto. Sempre quando o `new` é chamado, ele executa o **construtor da classe**. O construtor da classe é um bloco declarado com o **mesmo nome** que a classe:

```
class Conta {  
    int numero;
```

```
    Cliente titular;
    double saldo;
    double limite;

    // construtor
    Conta() {
        System.out.println("Construindo uma conta.");
    }

    // ..
}
```

Então, quando fizermos:

```
Conta c = new Conta();
```

A mensagem “construindo uma conta” aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Um construtor pode parecer, mas **não é** um método.

#### O construtor default

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar `new`, se todo `new` chama um construtor **obrigatoriamente**?

Quando você não declara nenhum construtor na sua classe, o Java cria um para você. Esse construtor é o **construtor default**, ele não recebe nenhum argumento e o corpo dele é vazio.

A partir do momento que você declara um construtor, o construtor default não é mais fornecido.

O interessante é que um construtor pode receber um argumento, podendo assim inicializar algum tipo de informação:

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;

    // construtor
    Conta(Cliente titular) {
        this.titular = titular;
    }

    // ..
}
```

Esse construtor recebe o titular da conta. Assim, quando criarmos uma conta, ela já terá um determinado titular.

```
Cliente carlos = new Cliente();
carlos.nome = "Carlos";
```

```
Conta c = new Conta(carlos);
System.out.println(c.titular.nome);
```

## 6.5 - A necessidade de um construtor

Tudo estava funcionando até agora. Para que utilizamos um construtor?

A idéia é bem simples. Se toda conta precisa de um titular, como obrigar todos os objetos que forem criados a ter um valor desse tipo? Basta criar um único construtor que recebe essa String!

O construtor se resume a isso! Dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação do mesmo.

Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto, nada mais natural que passar uma `String` representando o nome de um arquivo na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.

Você pode ter mais de um construtor na sua classe e, no momento do `new`, o construtor apropriado será escolhido.

### Construtor: um método especial?

Um construtor não é um método. Algumas pessoas o chamam de um método especial, mas definitivamente não é, já que não possui retorno e só é chamado durante a construção do objeto.

### Chamando outro construtor

Um construtor só pode rodar durante a construção do objeto, isto é, você nunca conseguirá chamar o construtor em um objeto já construído. Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro, para não ter de ficar copiando e colando:

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;

    // construtor
    Conta(Cliente titular) {
        // faz mais uma série de inicializações e configurações
        this.titular = titular;
    }

    public Conta (int numero, Cliente titular) {
        this(titular); // chama o construtor que foi declarado acima
        this.numero = numero;
    }

    //..
}
```

Existe um outro motivo, o outro lado dos construtores: facilidade. Às vezes, criamos um construtor que recebe diversos argumentos para não obrigar o usuário de uma classe a chamar diversos métodos do tipo 'set'.

No nosso exemplo do CPF, podemos forçar que a classe `Cliente` receba no mínimo o CPF, dessa maneira um `Cliente` já será construído e com um CPF válido.

#### Java Bean

Quando criamos uma classe com todos os atributos privados, seus getters e setters e um construtor vazio (padrão), na verdade estamos criando um Java Bean (mas não confunda com EJB, que é Enterprise Java Beans).

Para saber mais acesse: <http://java.sun.com/products/javabeans/>

## 6.6 - Atributos de classe

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A idéia mais simples:

```
Conta c = new Conta();  
totalDeContas = totalDeContas + 1;
```

Aqui, voltamos em um problema parecido com o da validação de CPF. Estamos espalhando um código por toda aplicação, e quem garante que vamos conseguir lembrar de incrementar a variável `totalDeContas` toda vez?

Tentamos então, passar para a seguinte proposta:

```
class Conta {  
    private int totalDeContas;  
    //...  
  
    Conta() {  
        this.totalDeContas = this.totalDeContas + 1;  
    }  
}
```

Quando criarmos duas contas, qual será o valor do `totalDeContas` de cada uma delas? Vai ser 1. Pois cada uma tem essa variável. **O atributo é de cada objeto.**

Seria interessante então, que essa variável fosse **única**, compartilhada por todos os objetos dessa classe. Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso em java, declaramos a variável como `static`.

```
private static int totalDeContas;
```

Quando declaramos um atributo como `static`, ele passa a não ser mais um atributo de cada objeto, e sim um **atributo da classe**, a informação fica guardada pela classe, não é mais individual para cada objeto.

Para acessarmos um atributo estático, não usamos a palavra chave `this`, mas sim o nome da classe:

```
class Conta {  
    private static int totalDeContas;  
    //...  
  
    Conta() {
```

```
        Conta.totalDeContas = Conta.totalDeContas + 1;
    }
}
```

Já que o atributo é privado, como podemos acessar essa informação a partir de outra classe? Precisamos de um getter para ele!

```
class Conta {
    private static int totalDeContas;
    //...

    Conta() {
        Conta.totalDeContas = Conta.totalDeContas + 1;
    }

    public int getTotalDeContas() {
        return Conta.totalDeContas;
    }
}
```

Como fazemos então para saber quantas contas foram criadas?

```
Conta c = new Conta();
int total = c.getTotalDeContas();
```

Precisamos criar uma conta antes de chamar o método! Isso não é legal, pois gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta. A idéia aqui é a mesma, transformar esse método que todo objeto conta tem em um método de toda a classe. Usamos a palavra static de novo, mudando o método anterior.

```
public static int getTotalDeContas() {
    return Conta.totalDeContas;
}
```

Para acessar esse novo método:

```
int total = Conta.getTotalDeContas();
```

Repare que estamos chamando um método não com uma referência para uma `Conta`, e sim usando o nome da classe.

#### Métodos e atributos estáticos

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz todo sentido já que dentro de um método estático não temos acesso a referência "this", pois um método estático é chamado através da classe, e não de um objeto.

O static realmente traz um "cheiro" procedural, porém em muitas vezes é necessário.

## 6.7 - Um pouco mais...

- 1) Em algumas empresas, o UML é amplamente utilizado. Às vezes, o programador recebe o UML já pronto, completo, e só deve preencher a implementação, devendo seguir à risca o UML. O que você acha dessa prática? Vantagens e desvantagens.
- 2) Se uma classe só tem atributos e métodos estáticos, que conclusões podemos tirar? O que lhe parece um método estático?
- 3) O padrão dos métodos `get` e `set` não vale para as variáveis de tipo `boolean`. Esses atributos são acessados via `is` e `set`. Por exemplo, para verificar se um carro está ligado seriam criados os métodos `isLigado` e `setLigado`.

## 6.8 - Exercícios

- 1) Adicione o modificador de visibilidade (`private`, se necessário) para cada atributo e método da classe `Funcionario`. Tente criar um `Funcionario` no `main` e modificar ou ler um de seus atributos privados. O que acontece?
- 2) Crie os getters e setters necessários da sua classe `Funcionario`. Por exemplo:

```
class Funcionario {
    private double salario;

    double getSalario() {
        return this.salario;
    }

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

- 3) Modifique suas classes que acessam e modificam atributos de um `Funcionario` para utilizar os getters e setters.

Por exemplo:

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

passa para:

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

- 4) Faça com que sua classe `Funcionario` possa receber, opcionalmente, o nome do `Funcionario` durante a criação do objeto. Utilize construtores para obter esse resultado.

Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o nome do Funcionario.

Seria algo mais ou menos assim:

```
class Funcionario {
    Funcionario() {
        // construtor sem argumentos
    }

    Funcionario(String nome) {
        // construtor que recebe o nome
    }
}
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

- 5) (opcional) Adicione um atributo na classe `Funcionario` de tipo `int` que se chama `identificador`. Esse identificador deve ter um valor único para cada instância do tipo `Funcionario`. O primeiro `Funcionario` instanciado tem `identificador` 1, o segundo 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

Crie um getter para o `identificador`. Devemos ter um setter?

- 6) (opcional) Crie os getters e setters da sua classe `Empresa` e coloque seus atributos como `private`. Lembre-se de que não necessariamente todos os atributos devem ter getters e setters.

Por exemplo, na classe `Empresa`, seria interessante ter um setter e getter para a sua array de funcionários? Não seria mais interessante ter um método como este?

```
class Empresa {
    // ...

    Funcionario getFuncionario(int posicao) {
        return this.funcionarios[posicao];
    }
}
```

- 7) (opcional) Na classe `Empresa`, em vez de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de `Funcionario`.

Agora, com esse construtor, o que acontece se tentarmos dar `new Empresa()` sem passar argumento algum? Por quê?

- 8) (opcional) Como garantir que datas como 31/2/2005 não sejam aceitas pela sua classe `Data`?
- 9) (opcional) Crie a classe `PessoaFisica`. Queremos ter a garantia de que pessoa física alguma tenha CPF inválido, nem seja criada `PessoaFisica` sem cpf inicial. (você não precisa escrever o algoritmo de validação de cpf, basta passar o cpf por um método `valida(String x)...`)

## 6.9 - Desafios

- 1) Porque esse código não compila?

```
class Teste {  
    int x = 37;  
    public static void main(String [] args) {  
        System.out.println(x);  
    }  
}
```

- 2) Imagine que tenha uma classe `FabricaDeCarro` e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)

# Orientação a Objetos – herança, reescrita e polimorfismo

*“O homem absurdo é aquele que nunca muda.”*  
– Georges Clemenceau

Ao término desse capítulo, você será capaz de:

- dizer o que é herança e quando utilizá-la;
- reutilizar código escrito anteriormente;
- criar classes filhas e reescrever métodos;
- usar todo o poder que o polimorfismo dá.

## 7.1 - Repetindo código?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe `Funcionario`:

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

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco:

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

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

```

        System.out.println("Acesso Negado!");
        return false;
    }
}

// outros métodos
}

```

#### Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe `Funcionario` mais genérica, mantendo nela senha de acesso. Caso o funcionário não fosse um gerente, deixaríamos este atributo vazio.

Essa é uma possibilidade. Mas e em relação aos métodos? A classe `Gerente` tem o método `autentica`, que não faz sentido existir em um funcionário que não é gerente.

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizar as informações principais do funcionário em um único lugar!

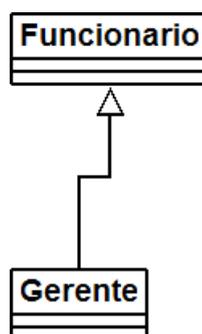
Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o `Gerente` tivesse tudo que um `Funcionario` tem, gostaríamos que ela fosse uma **extensão** de `Funcionario`. Fazemos isto através da palavra chave `extends`.

```

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

```

Em todo momento que criarmos um objeto do tipo `Gerente`, este objeto possuirá também os atributos definidos na classe `Funcionario`, pois agora um `Gerente` **é um** `Funcionario`:



```
class TestaGerente {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();
        gerente.setNome("João da Silva");
        gerente.setSenha(4231);
    }
}
```

Dizemos que a classe `Gerente` **herda** todos os atributos e métodos da classe mãe, no nosso caso, a `Funcionario`. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente.

#### Super e Sub classe

A nomenclatura mais encontrada é que `Funcionario` é a **superclasse** de `Gerente`, e `Gerente` é a **subclasse** de `Funcionario`. Dizemos também que todo `Gerente` é um `Funcionário`.

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de `Funcionario`, `public`, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o `protected`, que fica entre o `private` e o `public`. Um atributo `protected` só pode ser acessado (visível) pela própria classe ou suas subclasses.

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

#### Sempre usar protected?

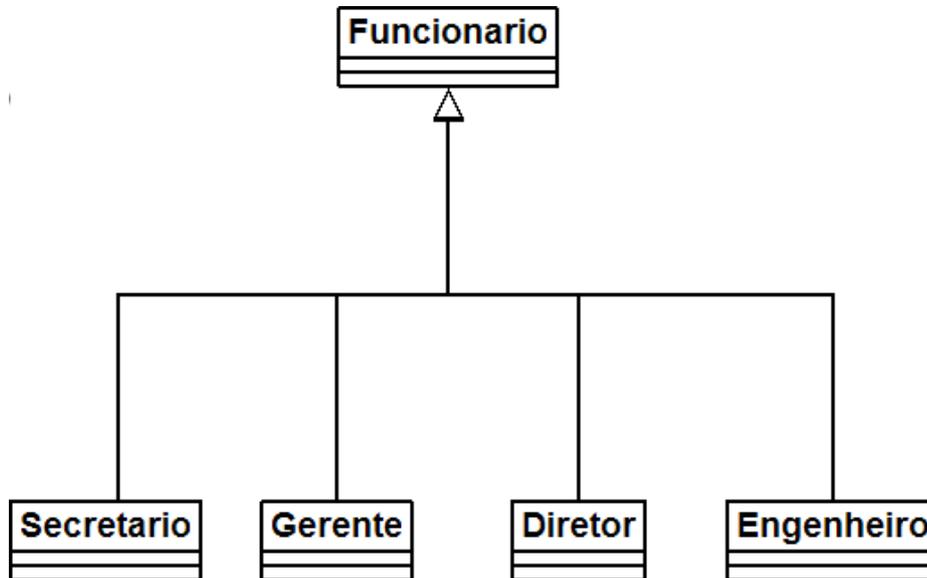
Então porque usar `private`? Depois de um tempo programando orientado a objetos, você vai começar a sentir que nem sempre é uma boa idéia deixar que a classe filha acesse os atributos da classe mãe, pois isso quebra um pouco a idéia de que só aquela classe deveria manipular seus atributos. Essa é uma discussão um pouco mais avançada.

Além disso, não só as subclasses, mas também as outras classes, podem acessar os atributos `protected`, que veremos mais a frente (mesmo pacote). Veja outras alternativas ao `protected` no exercício.

Da mesma maneira podemos ter uma classe `Diretor` que estenda `Gerente`, e a classe `Presidente` pode estender diretamente de `Funcionario`.

Fique claro que essa é uma decisão de negócio. Se você vai estender `Diretor` de `Gerente` ou não, vai depender se `Diretor` “é um” `Gerente`.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.



## 7.2 - Reescrita de método

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe `Funcionario`:

```
class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;
    public double getBonificacao() {
        return this.salario * 0.10;
    }
    // métodos
}
```

Se deixarmos a classe `Gerente` como ela está, ela vai herdar o método `getBonificacao`.

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta. Para consertar isso, uma das opções seria criar um novo método na classe `Gerente`, chamado, por exemplo, `getBonificacaoDoGerente`. O problema é que agora teríamos dois métodos em `Gerente`, confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (sobrescrever, *override*) este método:

```
class Gerente extends Funcionario {
    int senha;
    public double getBonificacao() {
        return this.salario * 0.15;
    }
}
```

```
}  
  // ...  
}
```

Agora sim, o método está correto para o `Gerente`. Refaça o teste e veja que agora o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```

### 7.3 - Invocando o método reescrito

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um `Gerente` devemos fazer igual ao cálculo de um `Funcionario` porem adicionando R\$ 1000. Poderíamos fazer assim:

```
class Gerente extends Funcionario {  
    int senha;  
  
    public double getBonificacao() {  
        return this.salario * 0.10 + 1000;  
    }  
    // ...  
}
```

Aqui teríamos um problema: o dia que o `getBonificacao` do `Funcionario` mudar, precisaremos mudar o método do `Gerente` para acompanhar a nova bonificação. Para evitar isso, o `getBonificacao` do `Gerente` pode chamar o do `Funcionario` utilizando-se da palavra chave `super`.

```
class Gerente extends Funcionario {  
    int senha;  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    // ...  
}
```

Essa invocação vai procurar o método com o nome `getBonificacao` de uma super classe de `Gerente`. No caso ele logo vai encontrar esse método em `Funcionario`.

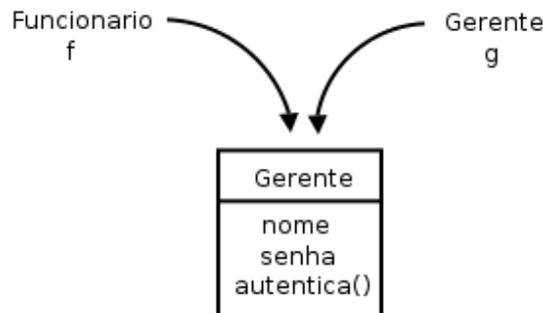
Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz “algo a mais” que o método da classe mãe. Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

### 7.4 - Polimorfismo

O que guarda uma variável do tipo `Funcionario`? Uma referência para um `Funcionario`.

Na herança, vimos que todo Gerente é um Funcionario, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Porque? Pois Gerente **é um** Funcionario. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();  
Funcionario funcionario = gerente;  
funcionario.setSalario(5000.0);
```



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser **decidida em tempo de execução**. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é o do Gerente. O retorno é 750.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que fariamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario:

```
class ControleDeBonificacoes {  
    private double totalDeBonificacoes = 0;  
  
    public void registra(Funcionario funcionario) {  
        this.totalDeBonificacoes += funcionario.getBonificacao();  
    }  
  
    public double getTotalDeBonificacoes() {  
        return this.totalDeBonificacoes;  
    }  
}
```

E, em algum lugar da minha aplicação (ou no main se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
```

```
Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um `Gerente` para um método que recebe um `Funcionario` como argumento. Pense como numa porta na agência bancária com o seguinte aviso: “Permitida a entrada apenas de Funcionários”. Um gerente pode passar nessa porta? Sim, pois `Gerente` **é um** `Funcionario`.

Qual será o valor resultante? Não importa que dentro do método `registra` do `ControleDeBonificacoes` receba `Funcionario`. Quando ele receber um objeto que realmente é um `Gerente`, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.**

No dia em que criarmos uma classe `Secretaria`, por exemplo, que é filha de `Funcionario`, precisaremos mudar a classe de `ControleDeBonificacoes`? Não. Basta a classe `Secretaria` reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a herança e reescrita de método: diminuir acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou `ControleDeBonificacoes` pode nunca ter imaginado a criação da classe `Secretaria` ou `Engenheiro`. Isso traz um reaproveitamento enorme de código.

## 7.5 - Um outro exemplo

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade {
    private String nome;
    private double salario;
    double getGastos() {
        return this.salario;
    }
    String getInfo() {
        return "nome: " + this.nome + " com salário " + this.salario;
    }
    // métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas/aula também.

```
class ProfessorDaFaculdade extends EmpregadoDaFaculdade {
    private int horasDeAula;
    double getGastos() {
```

```
        return this.getSalario() + this.horasDeAula * 10;
    }
    String getInfo() {
        String informacaoBasica = super.getInfo();
        String informacao = informacaoBasica + " horas de aula: " + this.horasDeAula;
        return informacao;
    }
    // métodos de get, set e outros
}
```

A novidade, aqui, é a palavra chave `super`. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```
class GeradorDeRelatorio {
    public void adiciona(EmpregadoDaFaculdade f) {
        System.out.println(f.getInfo());
        System.out.println(f.getGastos());
    }
}
```

Podemos passar para nossa classe qualquer `EmpregadoDaFaculdade`! Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o `Reitor`. Como ele também é um `EmpregadoDaFaculdade`, será que vamos precisar alterar alguma coisa na nossa classe de `Relatorio`? Não. Essa é a idéia! Quem programou a classe `GeradorDeRelatorio` nunca imaginou que existiria uma classe `Reitor` e, mesmo assim, o sistema funciona.

```
class Reitor extends ProfessorDaFaculdade {
    // informações extras
    String getInfo() {
        return super.getInfo() + " e ele é um reitor";
    }
    // não sobrescrevemos o getGastos!!!
}
```

## 7.6 - Um pouco mais...

- 1) Se não houvesse herança em Java, como você poderia reaproveitar o código de outra classe?
- 2) Uma discussão muito atual é sobre o abuso no uso da herança. Algumas pessoas usam herança apenas para reaproveitar o código, quando poderiam ter feito uma **composição**. Procure sobre herança versus composição.
- 3) Mesmo depois de reescrever um método da classe mãe, a classe filha ainda pode acessar o método antigo. Isto é feito através da palavra chave `super.método()`. Algo parecido ocorre entre os construtores das classes, o que?

#### Mais sobre o mau uso da herança

No blog da Caelum existe um artigo interessante abordando esse tópico:

<http://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/>

James Gosling, um dos criadores do Java, é um crítico do mau uso da herança. Nesta entrevista ele discute a possibilidade de se utilizar apenas interfaces e composição, eliminando a necessidade da herança:

<http://www.artima.com/intv/gosling3P.html>

## 7.7 - Exercícios: Herança e Polimorfismo

1) Vamos criar uma classe `Conta`, que possua um saldo, e os métodos para pegar saldo, depositar, e sacar.

a) Crie a classe `Conta`

```
class Conta {  
}
```

b) Adicione o atributo `saldo`

```
class Conta {  
    private double saldo;  
}
```

c) Crie os métodos `getSaldo()`, `deposita(double)` e `saca(double)`

```
class Conta {  
    private double saldo;  
    void deposita(double valor) {  
        this.saldo += valor;  
    }  
    void saca(double valor) {  
        this.saldo -= valor;  
    }  
    double getSaldo() {  
        return this.saldo;  
    }  
}
```

2) Adicione um método na classe `Conta`, que atualiza essa conta de acordo com uma taxa percentual fornecida.

```
class Conta {  
    private double saldo;  
  
    // outros métodos aqui também ...  
  
    void atualiza(double taxa) {  
        this.saldo += this.saldo * taxa;  
    }  
}
```

3) Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa.

Além disso, a ContaCorrente deve reescrever o método deposita, afim de retirar uma taxa bancária de dez centavos de cada depósito.

- Crie as classes ContaCorrente e ContaPoupanca. Ambas são filhas da classe Conta:

```
class ContaCorrente extends Conta {  
}
```

```
class ContaPoupanca extends Conta {  
}
```

- Reescreva o método atualiza na classe ContaCorrente, seguindo o enunciado:

```
class ContaCorrente extends Conta {  
    void atualiza(double taxa) {  
        this.saldo += this.saldo * taxa * 2;  
    }  
}
```

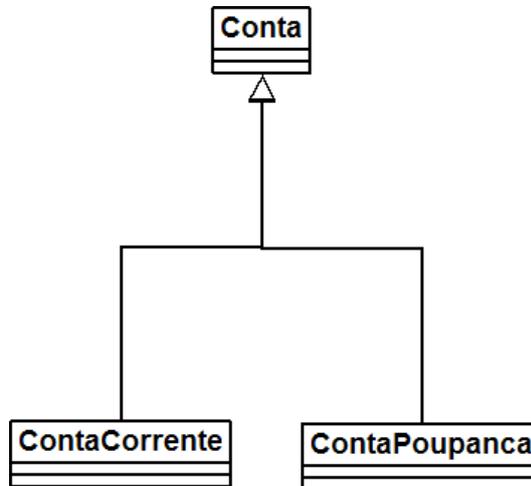
Repare que, para acessar o atributo saldo herdado da classe Conta, você vai precisar trocar o modificador de visibilidade de saldo para protected.

- Reescreva o método atualiza na classe ContaPoupanca, seguindo o enunciado:

```
class ContaPoupanca extends Conta {  
    void atualiza(double taxa) {  
        this.saldo += this.saldo * taxa * 3;  
    }  
}
```

- Na classe ContaCorrente, reescreva o método deposita para descontar a taxa bancária de dez centavos:

```
class ContaCorrente extends Conta {  
    void atualiza(double taxa) {  
        this.saldo += this.saldo * taxa * 2;  
    }  
  
    void deposita(double valor) {  
        this.saldo += valor - 0.10;  
    }  
}
```



Observação: existem outras soluções para modificar o saldo da sua classe mãe: você pode utilizar os métodos `retira` e `deposita`, se preferir continuar com o `private` (recomendado!), ou então criar um método `setSaldo`, mas `protected`, para não deixar outras pessoas alterarem o saldo sem passar por um método (nem mesmo sua filha conseguiria burlar isso). Hoje em dia, muitas pessoas dizem que o `protected` quebra encapsulamento, assim como alguns casos de herança onde a mãe e filha têm um acoplamento muito forte.

4) Crie uma classe com método `main` e instancie essas classes, atualize-as e veja o resultado. Algo como:

```
class TestaContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        ContaCorrente cc = new ContaCorrente();
        ContaPoupanca cp = new ContaPoupanca();

        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        c.atualiza(0.01);
        cc.atualiza(0.01);
        cp.atualiza(0.01);

        System.out.println(c.getSaldo());
        System.out.println(cc.getSaldo());
        System.out.println(cp.getSaldo());
    }
}
```

Após imprimir o saldo (`getSaldo()`) de cada uma das contas, o que acontece?

5) O que você acha de rodar o código anterior da seguinte maneira:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo - veremos o seu real poder no próximo exercício. Porém existe uma utilidade de declararmos

uma variável de um tipo menos específico do que o objeto realmente é.

É **extremamente importante** perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto e não de acordo com como nos referimos a ele.

- 6) (opcional) Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas.

```
class AtualizadorDeContas {
    private double saldoTotal = 0;
    private double selic;

    AtualizadorDeContas(double selic) {
        this.selic = selic;
    }

    void roda(Conta c) {
        // aqui voce imprime o saldo anterior, atualiza a conta,
        // e depois imprime o saldo final
        // lembrando de somar o saldo final ao atributo saldoTotal
    }

    // outros métodos, colocar o getter para saldoTotal!
}
```

- 7) (opcional) No método main, vamos criar algumas contas e rodá-las:

```
class TestaAtualizadorDeContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        Conta cc = new ContaCorrente();
        Conta cp = new ContaPoupanca();

        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        AtualizadorDeContas adc = new AtualizadorDeContas(0.01);

        adc.roda(c);
        adc.roda(cc);
        adc.roda(cp);

        System.out.println("Saldo Total: " + adc.getSaldoTotal());
    }
}
```

- 8) (Opcional) Use a palavra chave super nos métodos atualiza reescritos, para não ter de refazer o trabalho.
- 9) (Opcional) Se você precisasse criar uma classe ContaInvestimento, e seu método atualiza fosse complicadíssimo, você precisaria alterar as classes Banco e AtualizadorDeContas?
- 10) (Opcional, Trabalhoso) Crie uma classe Banco que possui um array de Conta. Repare que num array de

Conta você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca`. Crie um método `void adiciona(Conta c)`, um método `Conta pegaConta(int x)` e outro `int pegaTotalDeContas()`, muito similar a relação anterior de `Empresa-Funcionario`.

Faça com que seu método `main` crie diversas contas, insira-as no `Banco` e depois, com um `for`, percorra todas as contas do `Banco` para passá-las como argumento para o `AtualizadorDeContas`.

## 7.8 - Discussões em aula: Alternativas ao atributo `protected`

Discuta com o instrutor e seus colegas alternativas ao uso do atributo `protected` na herança. Preciso realmente afrouxar o encapsulamento o atributo por causa da herança? Como fazer para o atributo continuar `private` na mãe e as filhas conseguirem acessar?

# Eclipse IDE

*“Dá-se importância aos antepassados quando já não temos nenhum.”*  
– François Chateaubriand

Neste capítulo, você será apresentado ao Ambiente de Desenvolvimento Eclipse e suas principais funcionalidades.

## 8.1 - O Eclipse

O Eclipse (<http://www.eclipse.org>) é uma IDE (integrated development environment). Diferente de uma RAD, onde o objetivo é desenvolver o mais rápido possível através do arrastar-e-soltar do mouse, onde montanhas de código são gerados em background, uma IDE te auxilia no desenvolvimento, evitando se intrometer e fazer muita mágica.

O Eclipse é a IDE líder de mercado. Formada por um consórcio liderado pela IBM, possui seu código livre. A última versão é a 3.3. Precisamos do Eclipse 3.1 ou posterior, pois a partir dessa versão é que a plataforma dá suporte ao java 5.0. Você precisa ter apenas a Java RE instalada.

Veremos aqui os principais recursos do Eclipse. Você perceberá que ele evita ao máximo te atrapalhar e apenas gera trechos de códigos óbvios, sempre ao seu comando. Existem também centenas de plugins gratuitos para gerar diagramas UML, suporte a servidores de aplicação, visualizadores de banco de dados e muitos outros.

Baixe o Eclipse do site oficial <http://www.eclipse.org>. Apesar de ser escrito em Java, a biblioteca gráfica usada no Eclipse, chamada SWT, usa componentes nativos do sistema operacional. Por isso você deve baixar a versão correspondente ao seu sistema operacional.

Descompacte o arquivo e pronto: agora basta rodar o executável.

### Outras IDEs

Uma outra IDE open source famosa é o Netbeans, da Sun. (<http://www.netbeans.org>).

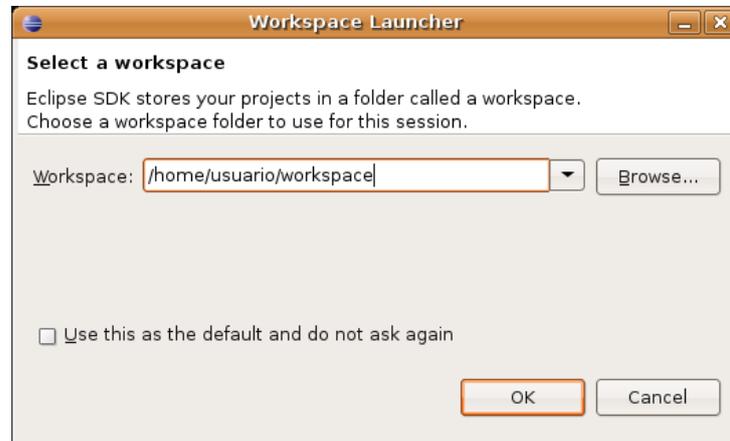
Além dessas, Oracle, Borland e a própria IBM possuem IDEs comerciais e algumas versões mais restritas de uso livre.

A empresa JetBrains desenvolve o IntelliJ IDEA, uma IDE paga que tem ganho muitos adeptos ultimamente.

## 8.2 - Apresentando o Eclipse

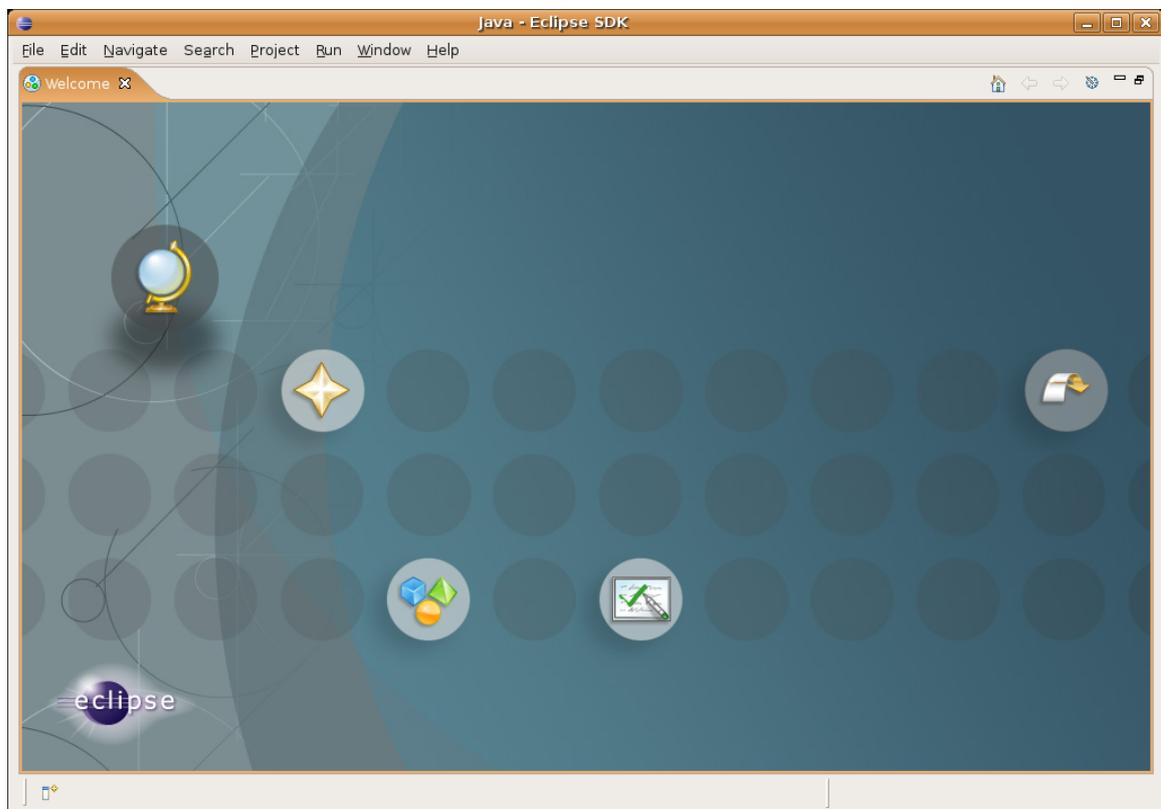
Clique no ícone do Eclipse no seu Desktop.

A primeira pergunta que ele te faz é que workspace você vai usar. Workspace define o diretório em que as suas configurações pessoais e seus projetos serão gravados.



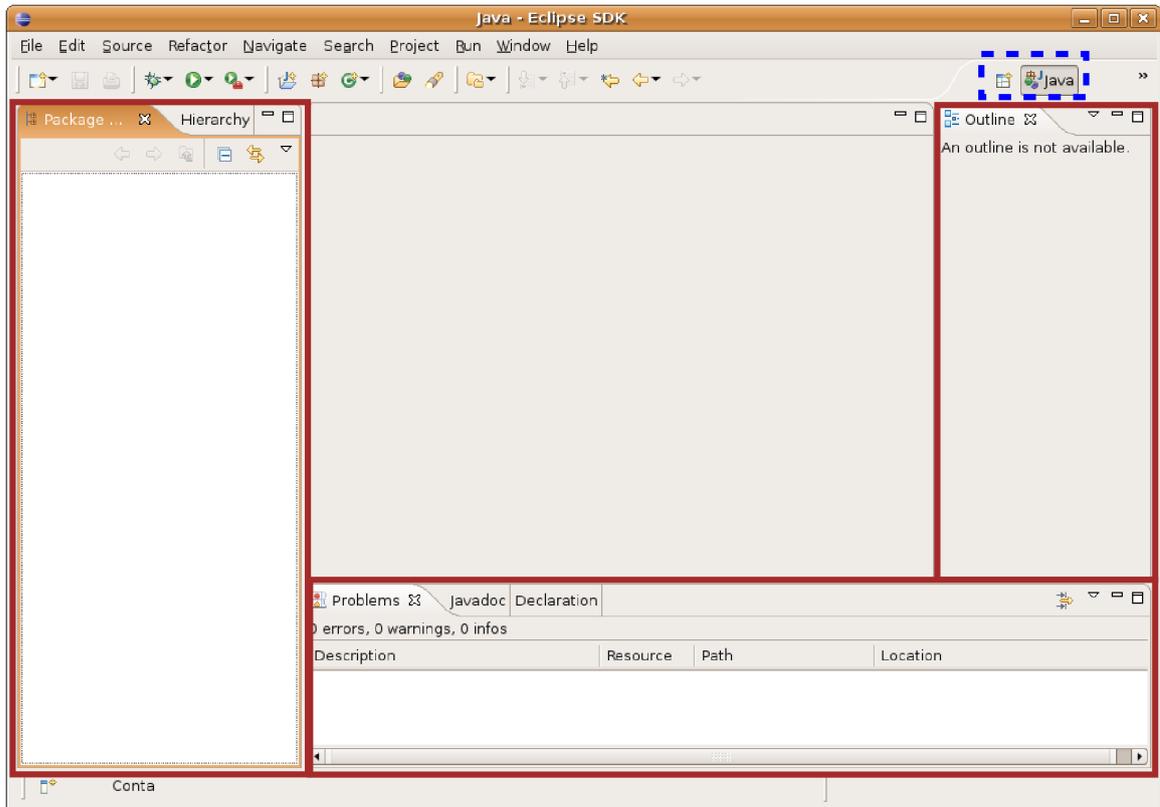
Você pode deixar o diretório pré-definido.

Logo em seguida, uma tela de Welcome será aberta, onde você tem diversos links para tutoriais e ajuda. Clique em Workbench. A tela de Welcome do Eclipse 3.3 e do 3.2 (que está na figura abaixo) são um pouco diferentes da do 3.1.

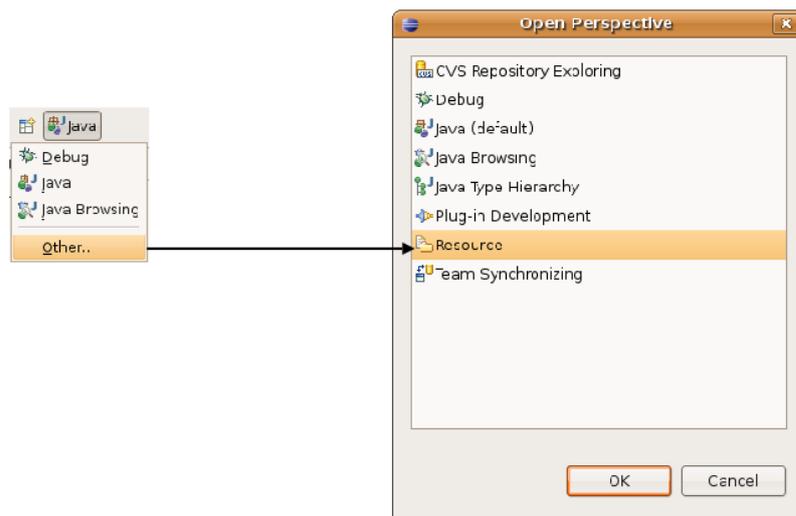


### 8.3 - Views e Perspective

Feche a tela de Welcome e você verá a tela abaixo. Nesta tela, destacamos as Views (em linha contínua) e as Perspectives (em linha pontilhada) do Eclipse.

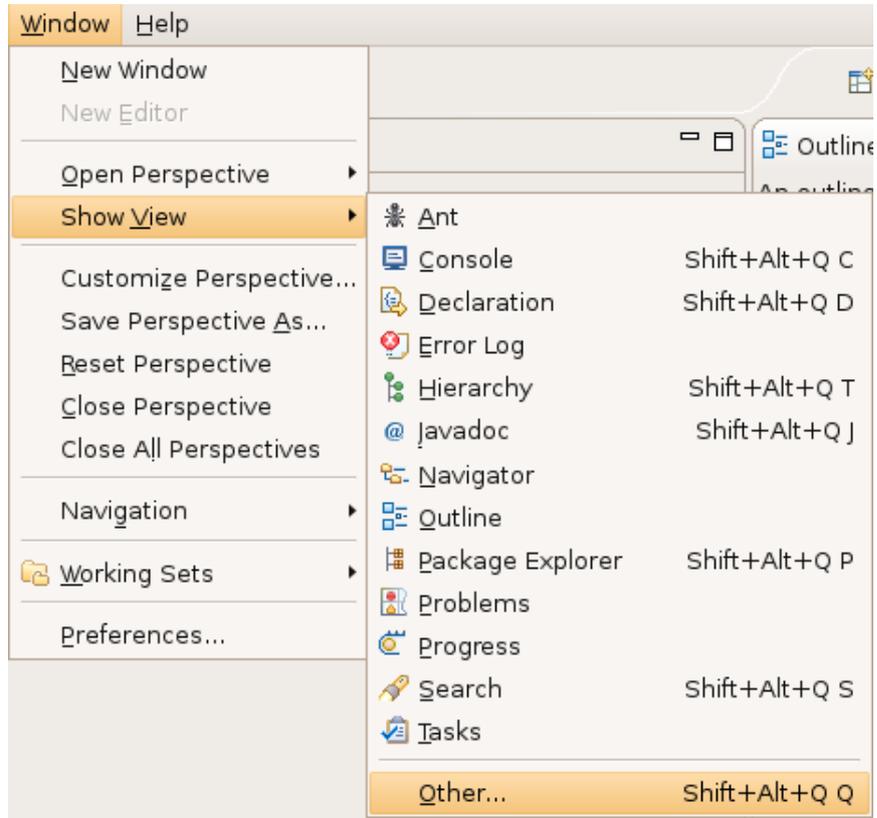


Mude para a perspectiva Resource, clicando no ícone ao lado da perspectiva Java, selecionando Other e depois Resource. Neste momento, trabalharemos com esta perspectiva, antes da de Java, pois ela possui um conjunto de Views mais simples.



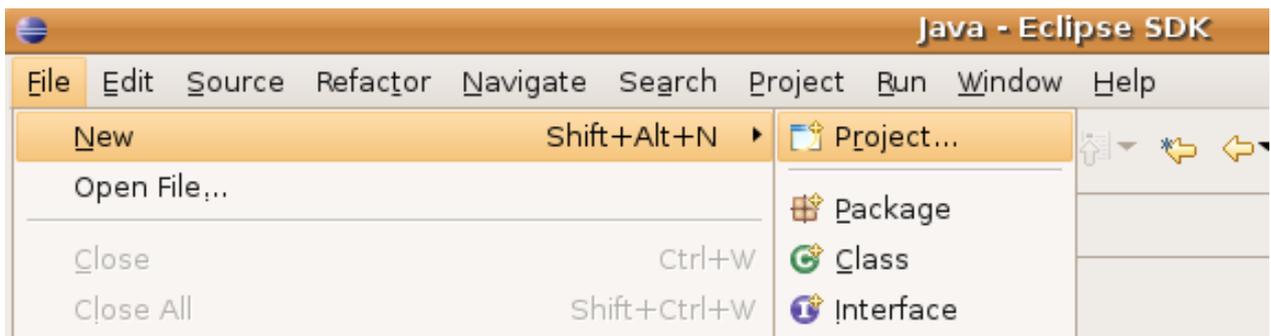
A View Navigator mostra a estrutura de diretório assim como está no sistema de arquivos. A View Outline mostra um resumo das classes, interfaces e enumerações declaradas no arquivo java atualmente editado (serve também para outros tipos de arquivos).

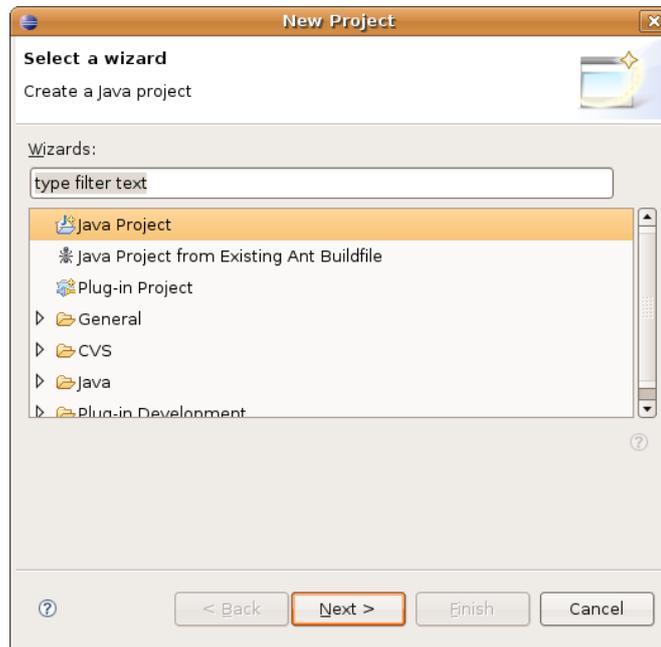
No menu Window -> Show View -> other, você pode ver as dezenas de Views que já vem embutidas no Eclipse. Acostume-se a sempre procurar novas Views, elas podem te ajudar em diversas tarefas.



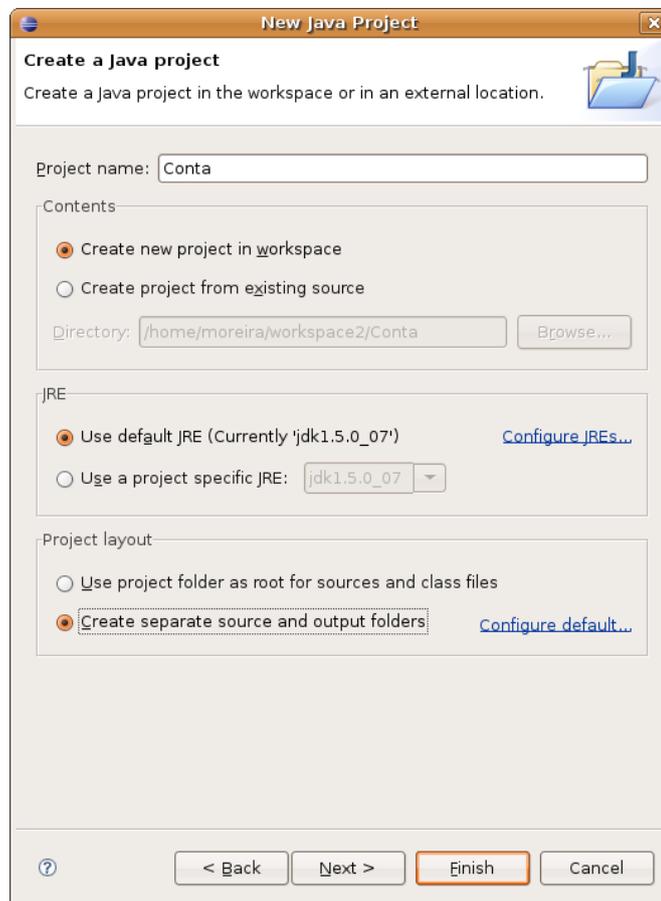
## 8.4 - Criando um projeto novo

Vá em File -> New -> Project. Selecciona Java Project e clique em Next.



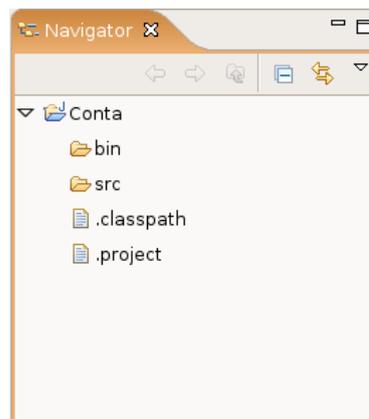


Você pode chegar nessa mesma tela clicando com o botão da direita no espaço da View Navigator e seguindo o mesmo menu. Nesta tela, configure seu projeto como na tela abaixo:

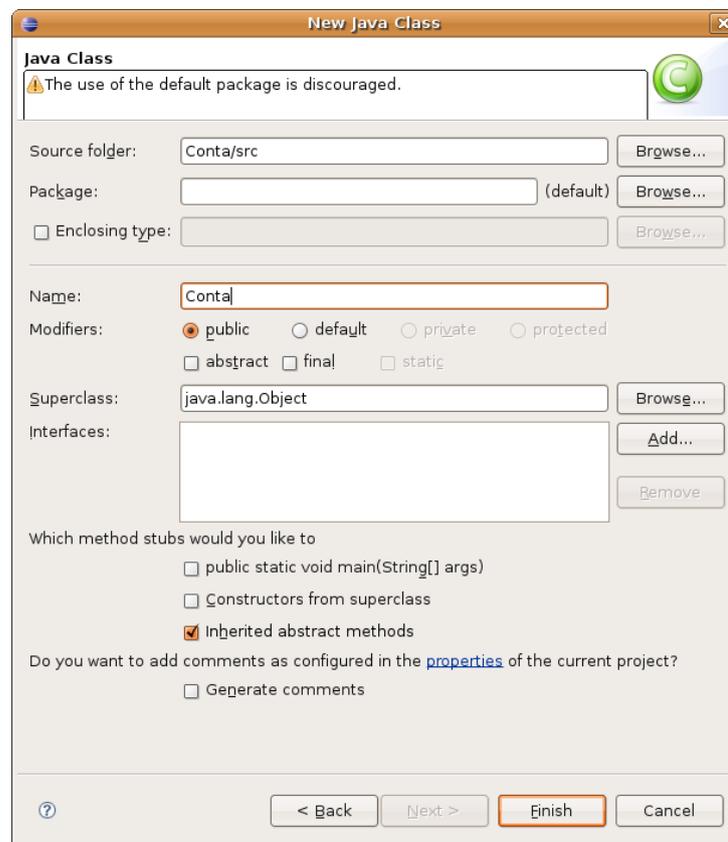


Isto é, marque “create separate source and output folders”, desta maneira seus arquivos java e arquivos class estarão em diretórios diferentes, para você trabalhar de uma maneira mais organizada.

Clique em Finish. O Eclipse pedirá para trocar a perspectiva para Java; escolha “No” para permanecer em Resource. Agora, na View Navigator, você verá o novo projeto e suas pastas e arquivos:

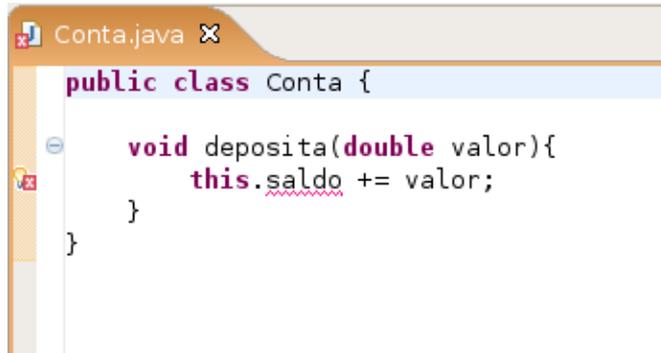


Vamos iniciar nosso projeto criando a classe Conta. Para isso, vá em File -> New -> Other -> Class. Clique em Next e crie a classe seguindo a tela abaixo:



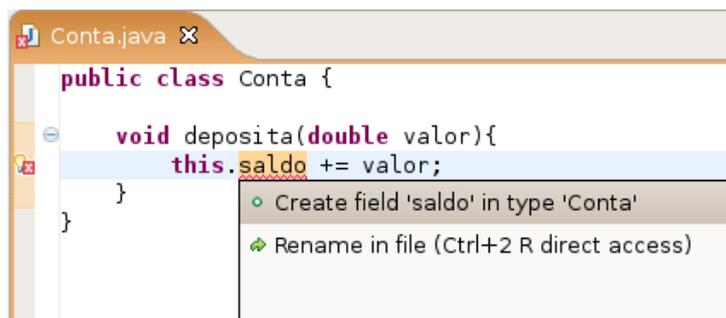
Clique em Finish. O Eclipse possui diversos wizards, mas usaremos o mínimo deles. O interessante é usar o *code assist* e *quick fixes* que a ferramenta possui e veremos em seguida. Não se atente às milhares de opções de cada wizard, a parte mais interessante do Eclipse não é essa.

Escreva o método `deposita` como abaixo e note que o Eclipse reclama de erro em `this.saldo` pois este atributo não existe.



```
Conta.java X
public class Conta {
    void deposita(double valor){
        this.saldo += valor;
    }
}
```

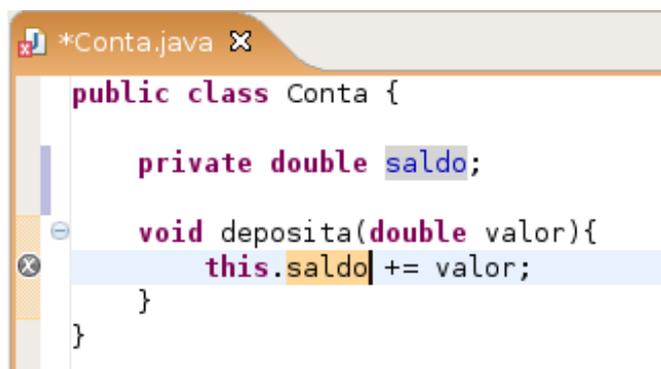
Vamos usar o recurso do Eclipse de **quick fix**. Coloque o cursor em cima do erro e aperte `Ctrl + 1`.



```
Conta.java X
public class Conta {
    void deposita(double valor){
        this.saldo += valor;
    }
}
```

- o Create field 'saldo' in type 'Conta'
- Rename in file (Ctrl+2 R direct access)

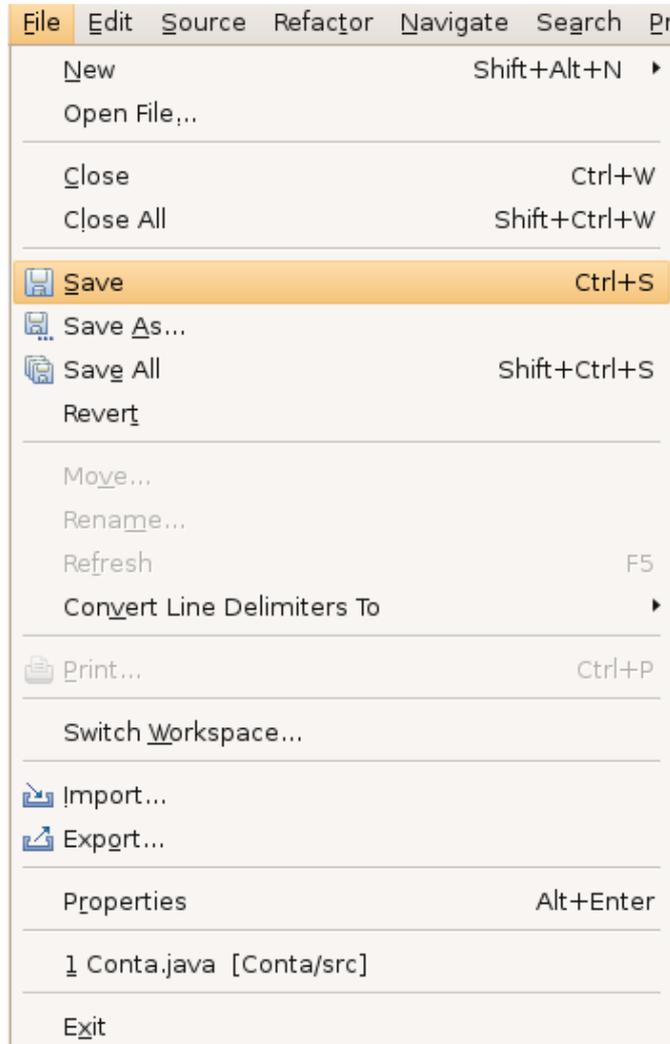
O Eclipse sugerirá possíveis formas de consertar o erro; uma delas é, justamente, criar o campo `saldo` na classe `Conta`, que é nosso objetivo. Clique nesta opção.



```
*Conta.java X
public class Conta {
    private double saldo;
    void deposita(double valor){
        this.saldo += valor;
    }
}
```

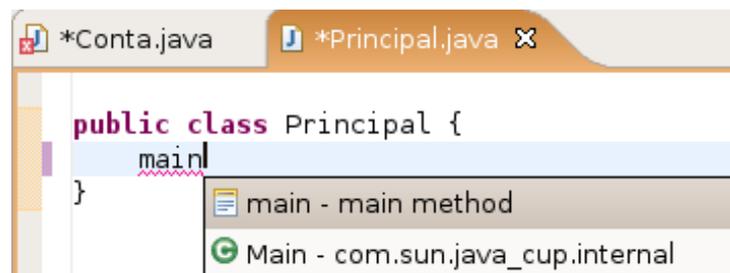
Este recurso de quick fixes, acessível pelo `Ctrl+1`, é uma das grandes facilidades do Eclipse e é extremamente poderoso. Através dele é possível corrigir boa parte dos erros na hora de programar e, como fizemos, economizar a digitação de certos códigos repetitivos. No nosso exemplo, não precisamos criar o campo antes; o Eclipse faz isso para nós. Ele até acerta a tipagem, já que estamos somando ele a um `double`. O `private` é colocado por motivos que já estudamos.

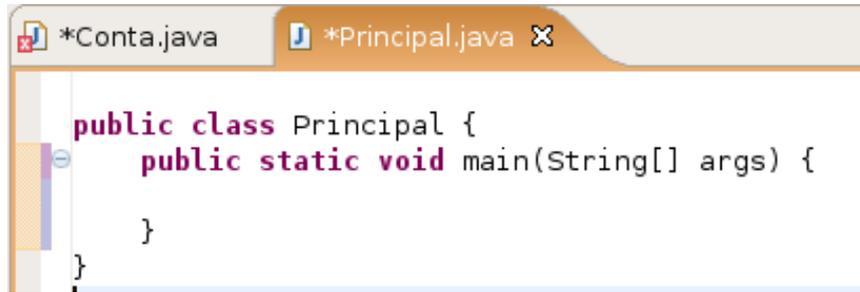
Vá ao menu `File -> Save` para gravar. `Control + S` tem o mesmo efeito.



## 8.5 - Criando o main

Crie uma nova classe chamada `Principal`. Vamos colocar um método `main` para testar nossa `Conta`. Em vez de digitar todo o método `main`, vamos usar o **code assist** do Eclipse. Escreva só `main` e aperte `Ctrl + Espaço` logo em seguida.





```
*Conta.java *Principal.java X
public class Principal {
    public static void main(String[] args) {

    }
}
```

O Eclipse sugerirá a criação do método `main` completo; selecione esta opção. O control + espaço é chamado de **code assist**. Assim como os quick fixes são de extrema importância. Experimente usar o code assist em diversos lugares.

Agora, dentro do método `main`, comece a digitar o seguinte código:

```
Conta conta = new Conta();
conta.deposita(100.0);
```

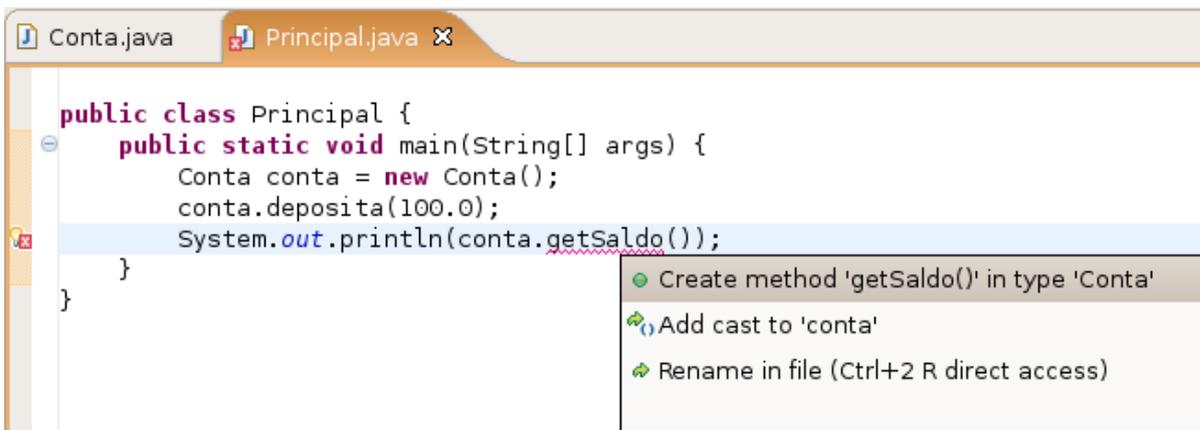
Observe que, na hora de invocar o método sobre o objeto `conta`, o Eclipse sugere os métodos possíveis. Este recurso é bastante útil, principalmente quando estivermos programando com classes que não são as nossas, como da API do Java. O Eclipse aciona este recurso quando você digita o ponto logo após um objeto (e você pode usar o Ctrl+Espaço para acioná-lo).

Agora, vamos imprimir o saldo com `System.out.println`. Mas, mesmo nesse código, o Eclipse nos ajuda. Escreva `syso` e aperte Ctrl+Espaço que o Eclipse escreverá `System.out.println()` para você.

Para imprimir, chame o `conta.getSaldo()`:

```
System.out.println(conta.getSaldo());
```

Note que o Eclipse acusará erro em `getSaldo()` porque este método não existe na classe `Conta`. Vamos usar Ctrl+1 em cima do erro para corrigir o problema:



```
Conta.java Principal.java X
public class Principal {
    public static void main(String[] args) {
        Conta conta = new Conta();
        conta.deposita(100.0);
        System.out.println(conta.getSaldo());
    }
}
```

- Create method 'getSaldo()' in type 'Conta'
- Add cast to 'conta'
- Rename in file (Ctrl+2 R direct access)

O Eclipse sugere criar um método `getSaldo()` na classe `Conta`. Selecione esta opção e o método será inserido automaticamente.

```
public Object getSaldo() {
    // TODO Auto-generated method stub
}
```

```
    return null;  
}
```

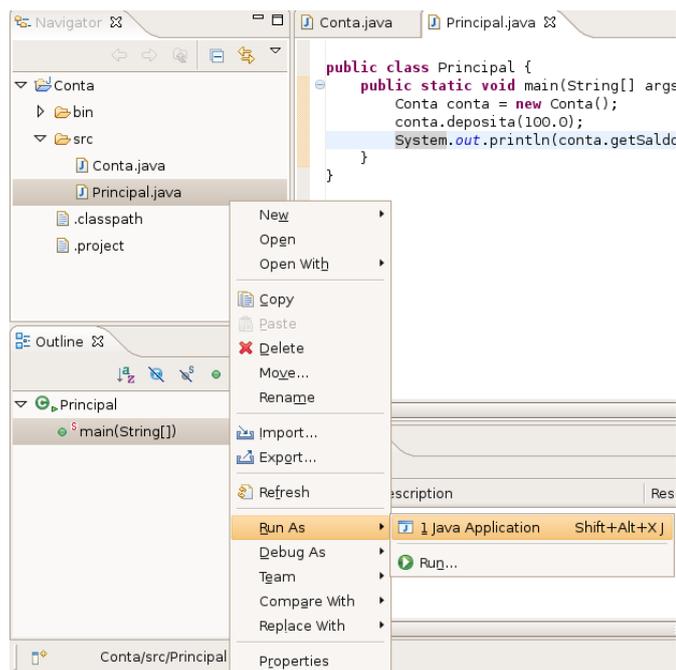
Implemente o método `getSaldo` como segue:

```
public double getSaldo() {  
    return this.saldo;  
}
```

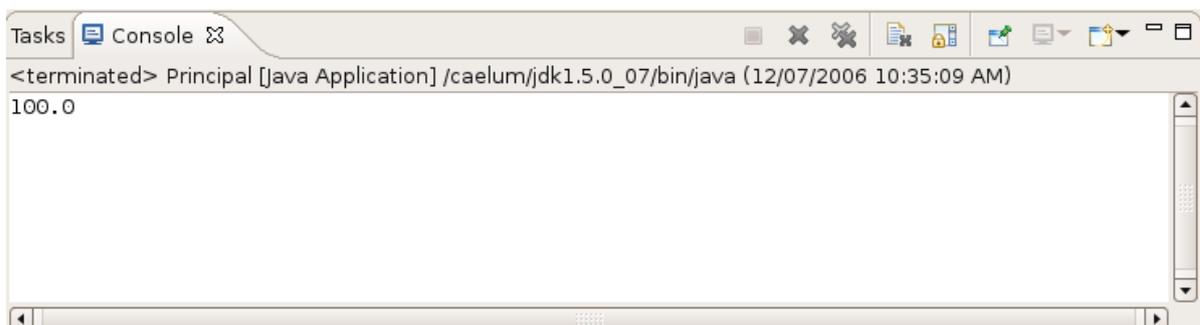
Esses pequenos recursos do Eclipse são de extrema utilidade. Dessa maneira, você pode programar sem se preocupar com métodos que ainda não existem, já que a qualquer momento ele pode te gerar o esqueleto (a parte da assinatura do método).

## 8.6 - Rodando o main

Vamos rodar o método `main` dessa nossa classe. No Eclipse, clique com o botão direito no arquivo `Principal.java` e vá em `Run as... Java Application`.



O Eclipse abrirá uma View chamada Console onde será apresentada a saída do seu programa:



Quando você precisar rodar de novo, basta clicar no ícone verde de play na toolbar, que roda o programa anterior. Ao lado desse ícone tem uma setinha onde são listados os 10 últimos executados.

## 8.7 - Pequenos truques

O Eclipse possui muitos atalhos úteis para o programador. Alguns bem interessantes de saber:

- **Ctrl + 1** Aciona o quick fixes com sugestões para correção de erros.
- **Ctrl + Espaço** Completa códigos
- **Ctrl + 3** Aciona modo de descoberta de menu. Experimente digitar **Ctrl+3** e depois digitar **ggas** e **enter**.
- **Ctrl + PgUp** e **Ctrl + PgDown** Navega nas abas abertas. Útil quando estiver editando vários arquivos ao mesmo tempo.
- **Ctrl + Shift + F** Formata o código segundo as convenções do Java
- **Ctrl + M** Expande a View atual para a tela toda (mesmo efeito de dar dois cliques no título da View)
- **Ctrl + Shift + L** Exibe todos os atalhos possíveis.
- **Ctrl + O** Exibe um outline para rápida navegação

Veremos mais no decorrer do curso, em especial quando virmos pacotes.

## 8.8 - Exercícios: Eclipse

- 1) Crie as classes `ContaCorrente`, `ContaPoupanca` e opcionalmente o `AtualizadorDeContas` no nosso projeto do Eclipse. Desta vez tente abusar do `control + espaço` e `control + 1`.

Por exemplo:

```
ContaCorr<ControlEspaco> <ControlEspaco> = new <ControlEspaco>();
```

Repare que até mesmo nomes de variáveis, ele cria para você!

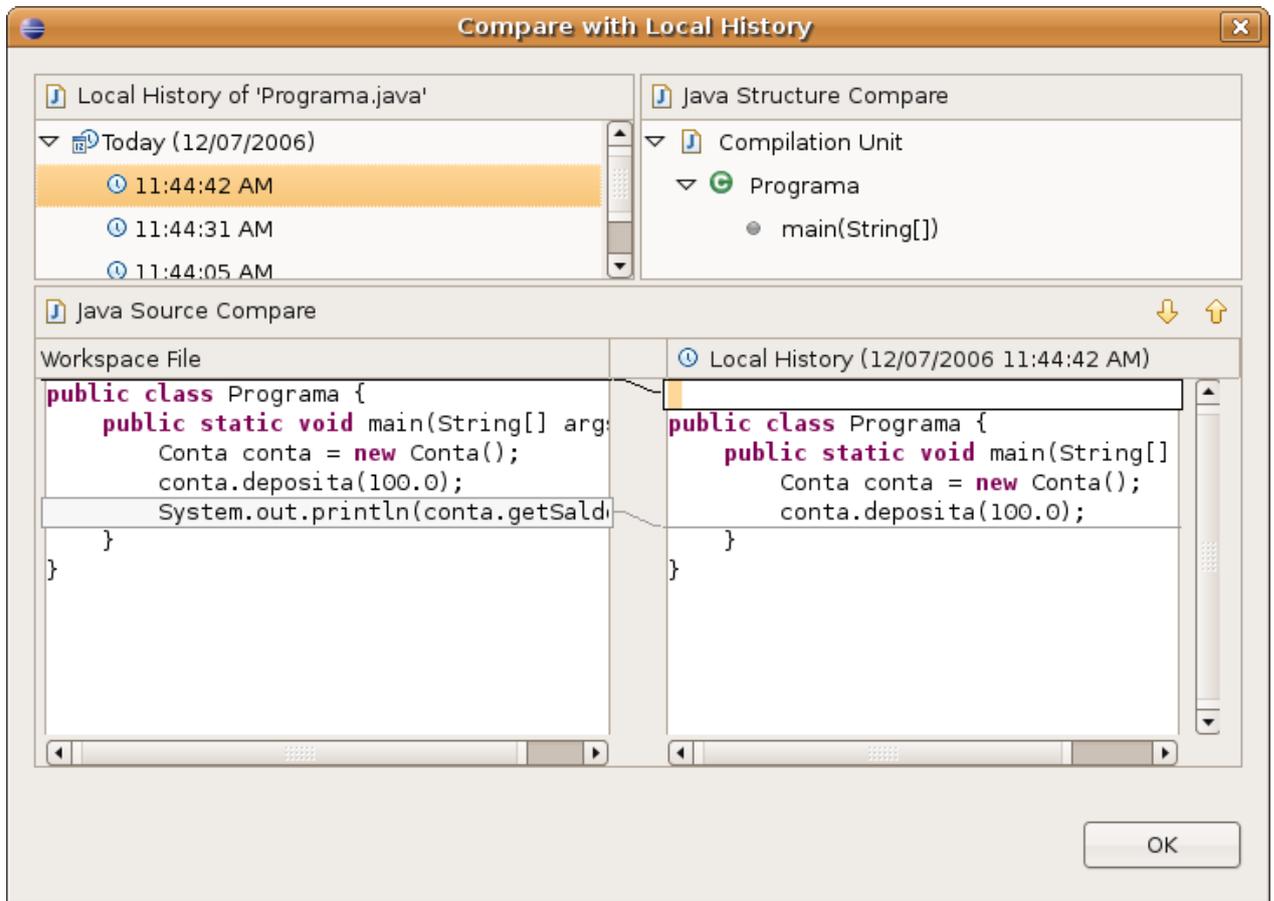
- 2) Imagine que queremos criar um setter do saldo para a classe `Conta`. Dentro da classe `Conta`, digite:

```
setSa<ControlEspaco>
```

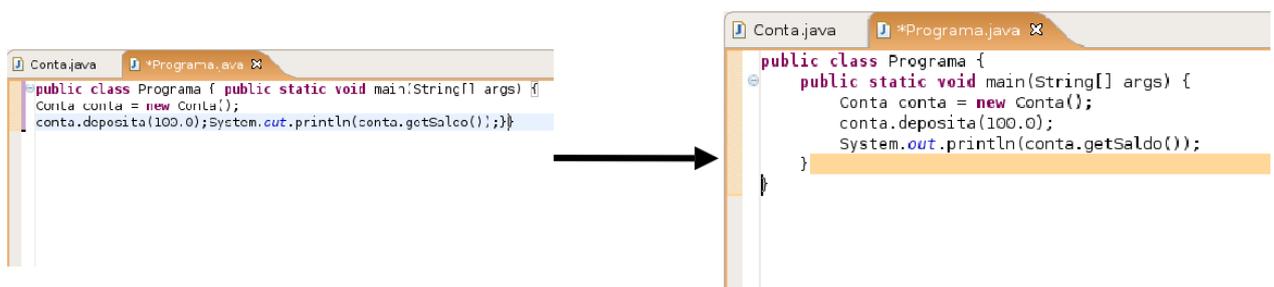
O mesmo vale no caso de você querer reescrever um método. Dentro de `ContaCorrente` faça:

```
atua<ControlEspaco>
```

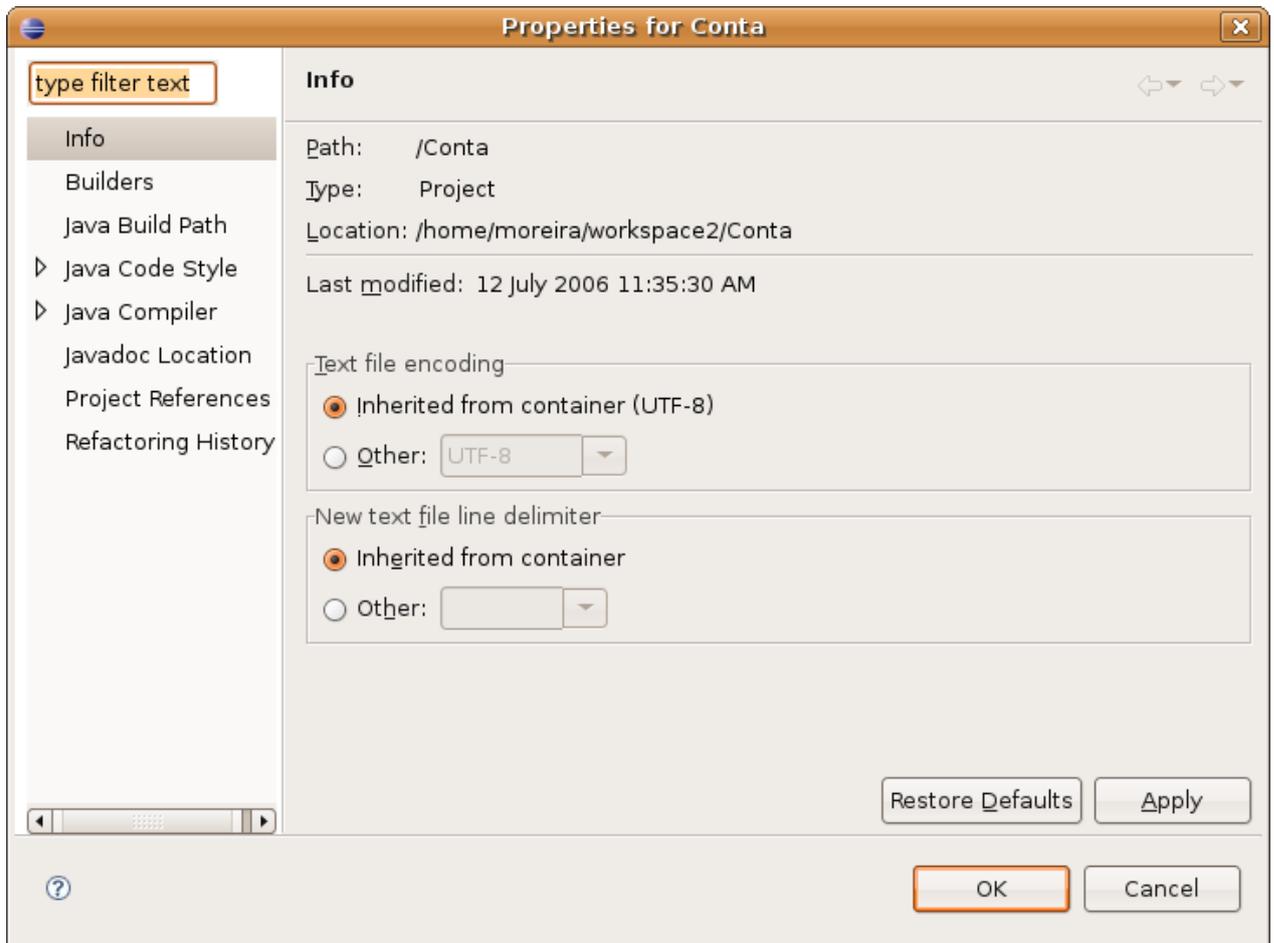
- 3) Vá na sua classe que tem o `main` e segure o `CONTROL` apertado enquanto você passa o mouse sobre o seu código. Repare que tudo virou hyperlink. Clique em um método que você está invocando na classe `Conta`.
- 4) Dê um clique da direita em um arquivo no navigator. Escolha `Compare With -> Local History`. O que é esta tela?



- 5) Use o Control + Shift + F para formatar o seu código. Dessa maneira, ele vai arrumar a bagunça de espaçamento e enters do seu código.



- 6) (opcional) O que são os arquivos .project e .classpath? Leia o conteúdo deles.
- 7) (opcional) Clique da direita no projeto, propriedades. É uma das telas mais importantes do Eclipse, onde você pode configurar diversas funcionalidades para o seu projeto, como compilador, versões, formatador, cvs e outros.



## 8.9 - Discussão em aula: Refactorings

O instrutor irá mostrar o que é uma **refatoração** e como usar esse poderoso recurso no Eclipse.

# Orientação a Objetos – Classes Abstratas

*“Dá-se importância aos antepassados quando já não temos nenhum.”*

– François Chateaubriand

Ao término desse capítulo, você será capaz de utilizar classes abstratas, quando necessário.

## 9.1 - Repetindo mais código?

Neste capítulo, aconselhamos que você passe a usar o Eclipse. Você já tem conhecimento suficiente dos erros de compilação do `javac` e agora pode aprender as facilidades que o Eclipse te traz ao ajudar você no código com os chamados quick fixes e quick assists.

Vamos recordar em como pode estar nossa classe `Funcionario`:

```
class Funcionario {  
  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 1.2;  
    }  
  
    // outros métodos aqui  
}
```

Considere, agora, o nosso `ControleDeBonificacao`:

```
class ControleDeBonificacoes {  
  
    private double totalDeBonificacoes = 0;  
  
    public void registra(Funcionario f) {  
        System.out.println("Adicionando bonificacao do funcionario: " + f);  
        this.totalDeBonificacoes += funcionario.getBonificacao();  
    }  
  
    public double getTotalDeBonificacoes() {  
        return this.totalDeBonificacoes;  
    }  
}
```

Nosso método `registra` recebe qualquer referência do tipo `Funcionario`, isto é, podem ser objetos do tipo `Funcionario` e qualquer de seus subtipos: `Gerente`, `Diretor` e, eventualmente, alguma nova subclasse que venha ser escrita, sem prévio conhecimento do autor da `ControleDeBonificacao`.

Estamos utilizando aqui a classe `Funcionario` para o polimorfismo. Se não fosse ela, teríamos um grande prejuízo: precisaríamos criar um método `bonifica` para receber cada um dos tipos de `Funcionario`, um para `Gerente`, um para `Diretor`, etc. Repare que perder esse poder é muito pior do que a pequena vantagem que a herança traz em herdar código.

Porém, em alguns sistemas, como é o nosso caso, usamos uma classe com apenas esses intuitos: de economizar um pouco código e ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.

Faz sentido ter um objeto do tipo `Funcionario`? Essa pergunta é diferente de saber se faz sentido ter uma referência do tipo `Funcionario`: nesse caso, faz sim e é muito útil.

Referenciando `Funcionario` temos o polimorfismo de referência, já que podemos receber qualquer coisa que seja um `Funcionario`. Porém, dar `new` em `Funcionario` pode não fazer sentido, isto é, não queremos receber um objeto do tipo `Funcionario`, mas sim que aquela referência seja ou um `Gerente`, ou um `Diretor`, etc. Algo mais **concreto** que um `Funcionario`.

```
ControleDeBonificacoes cdb = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
cdb.adiciona(f); // faz sentido?
```

Vejamos um outro caso em que não faz sentido ter um objeto daquele tipo, apesar da classe existir: imagine a classe `Pessoa` e duas filhas, `PessoaFisica` e `PessoaJuridica`. Quando puxamos um relatório de nossos clientes (uma array de `Pessoa` por exemplo), queremos que cada um deles seja ou uma `PessoaFisica`, ou uma `PessoaJuridica`. A classe `Pessoa`, nesse caso, estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas: não faz sentido permitir instanciá-la.

Para resolver esses problemas, temos as classes abstratas.

## 9.2 - Classe abstrata

O que, exatamente, vem a ser a nossa classe `Funcionario`? Nossa empresa tem apenas `Diretores`, `Gerentes`, `Secretárias`, etc. Ela é uma classe que apenas idealiza um tipo, define apenas um rascunho.

Para o nosso sistema, é inadmissível que um objeto seja apenas do tipo `Funcionario` (pode existir um sistema em que faça sentido ter objetos do tipo `Funcionario` ou apenas `Pessoa`, mas, no nosso caso, não).

Usamos a palavra chave `abstract` para impedir que ela possa ser instanciada. Esse é o efeito direto de se usar o modificador `abstract` na declaração de uma classe:

```
abstract class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e metodos comuns a todos Funcionarios
```

```
}
```

E, no meio de um código:

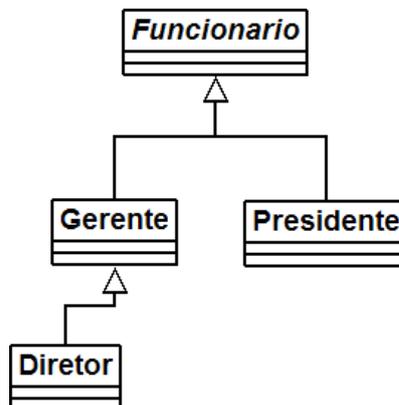
```
Funcionario f = new Funcionario(); // não compila!!!
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
    Cannot instantiate the type Funcionario  
  
    at br.com.caelum.empresa.TestaFuncionario.main(TestaFuncionario.java:5)
```

O código acima não compila. O problema é instanciar a classe - criar referência, você pode. Se ela não pode ser instanciada, para que serve? Somente para o polimorfismo e herança dos atributos e métodos.

Vamos então herdar dessa classe, reescrevendo o método `getBonificacao`:

```
class Gerente extends Funcionario {  
  
    public String getBonificacao() {  
        return this.salario * 1.4 + 1000;  
    }  
}
```



Mas qual é a real vantagem de uma classe abstrata? Poderíamos ter feito isto com uma herança comum. Por enquanto, a única diferença é que não podemos instanciar um objeto do tipo `Funcionario`, que já é de grande valia, dando mais consistência ao sistema.

Fique claro que a nossa decisão de transformar `Funcionario` em uma classe abstrata dependeu do nosso negócio. Pode ser que, em um sistema com classes similares, faça sentido uma classe análoga a `Funcionario` ser concreta.

## 9.3 - Métodos abstratos

Se o método `getBonificacao` não fosse reescrito, ele seria herdado da classe mãe, fazendo com que devolvesse o salário mais 20%.

Levando em consideração que cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado, faz algum sentido ter esse método na classe `Funcionario`? Será que existe uma bonificação padrão para todo tipo de `Funcionario`? Parece não, cada classe filha terá um método diferente de bonificação pois, de acordo com nosso sistema, não existe uma regra geral: queremos que cada pessoa que escreve a classe de um `Funcionario` diferente (subclasses de `Funcionario`) reescreva o método `getBonificacao` de acordo com as suas regras.

Poderíamos, então, jogar fora esse método da classe `Funcionario`? O problema é que, se ele não existisse, não poderíamos chamar o método apenas com uma referência a um `Funcionario`, pois ninguém garante que essa referência aponta para um objeto que possui esse método.

Existe um recurso em Java que, em uma classe abstrata, podemos escrever que determinado método será **sempre** escrito pelas classes filhas. Isto é, um **método abstrato**.

Ele indica que todas as classes filhas (concretas, isto é, que não forem abstratas) devem reescrever esse método ou não compilarão. É como se você herdasse a responsabilidade de ter aquele método.

### Como declarar um método abstrato

Às vezes, não fica claro como declarar um método abstrato.

Basta escrever a palavra chave `abstract` na assinatura do mesmo e colocar um ponto e vírgula em vez de abre e fecha chaves!

```
abstract class Funcionario {  
  
    abstract double getBonificacao();  
  
    // outros atributos e metodos  
  
}
```

Repare que não colocamos o corpo do método e usamos a palavra chave `abstract` para definir o mesmo. Por que não colocar corpo algum? Porque esse método nunca vai ser chamado, sempre que alguém chamar o método `getBonificacao`, vai cair em uma das suas filhas, que realmente escreveram o método.

Qualquer classe que estender a classe `Funcionario` será obrigada a reescrever este método, tornando-o "concreto". Se não reescreverem esse método, um erro de compilação ocorrerá.

O método do `ControleDeBonificacao` estava assim:

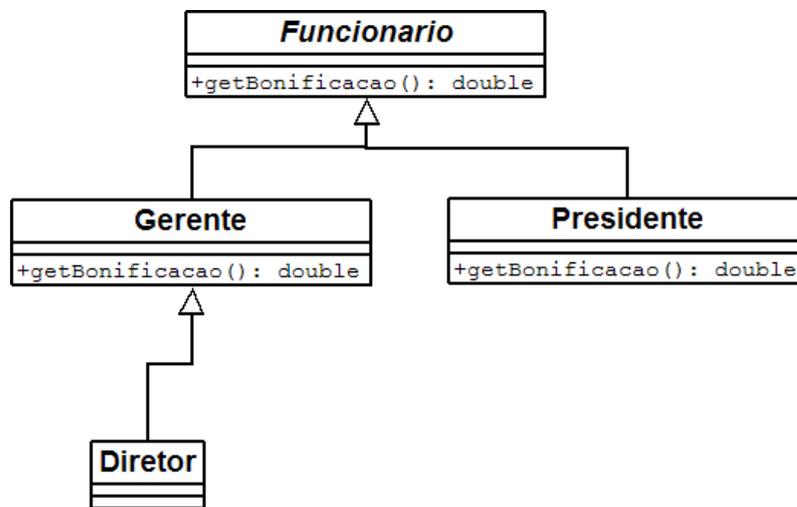
```
public void registra(Funcionario f) {  
    System.out.println("Adicionando bonificacao do funcionario: " + f);  
    this.totalDeBonificacoes += funcionario.getBonificacao();  
}
```

Como posso acessar o método `getBonificacao` se ele não existe na classe `Funcionario`?

Já que o método é abstrato, **com certeza** suas subclasses têm esse método, o que garante que essa invocação de método não vai falhar. Basta pensar que uma referência do tipo `Funcionario` nunca aponta para um objeto que não tem o método `getBonificacao`, pois não é possível instanciar uma classe abstrata, apenas as concretas. Um método abstrato obriga a classe em que ele se encontra ser abstrata, o que garante a coerência do código acima compilar.

## 9.4 - Aumentando o exemplo

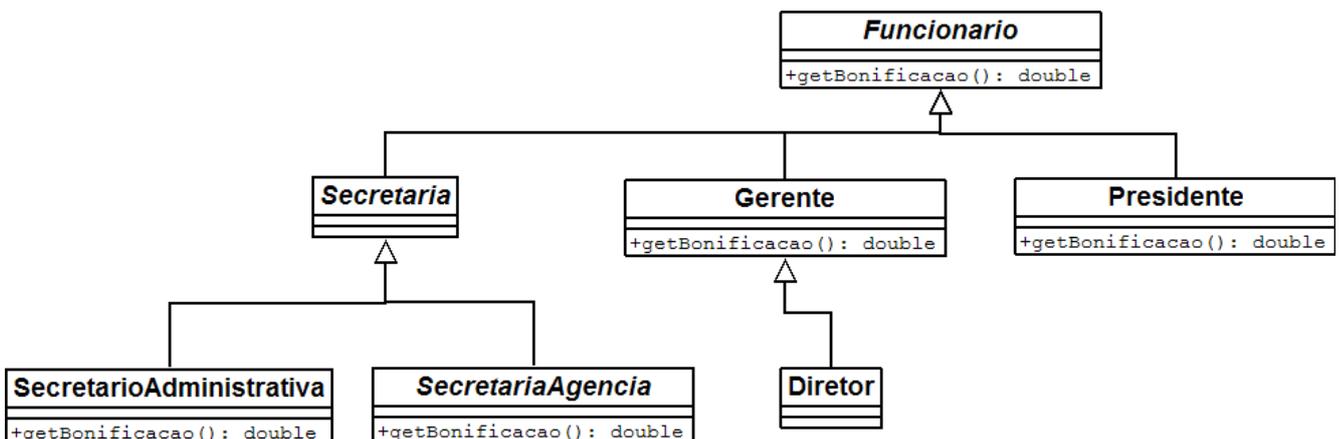
E agora se, no nosso exemplo de empresa, tivéssemos o seguinte diagrama de classes com os seguintes métodos:



Ou seja, tenho a classe abstrata `Funcionario`, com o método abstrato `getBonificacao`; as classes `Gerente` e `Presidente` estendendo `Funcionario` e implementando o método `getBonificacao`; e, por fim, a classe `Diretor`, que estende `Gerente`, mas não implementa o método `getBonificacao`.

Essas classes vão compilar? Vão rodar?

A resposta é sim e, além de tudo, farão exatamente o que nós queremos, pois, quando `Gerente` e `Presidente` possuem os métodos perfeitamente implementados, e a classe `Diretor`, que não possui o método implementado, vai usar a implementação de `Gerente`.



E esse diagrama, no qual incluímos uma classe abstrata `Secretaria` sem o método `getBonificacao`, que é estendida por mais duas classes (`SecretariaAdministrativa`, `SecretariaAgencia`) que, por sua vez, implementam o método `getBonificacao`, vai compilar? Vai rodar?

De novo, a resposta é sim, pois `Secretaria` é uma classe abstrata e, por isso, o Java tem certeza de que ninguém vai conseguir instanciá-la e, muito menos, chamar o método `getBonificacao` dela. Lembrando que, nesse caso, não precisamos nem ao menos escrever o método abstrato `getBonificacao` na classe `Secretaria`.

Se eu não reescrever um método abstrato da minha classe mãe, o código não compilará. Mas posso, em vez disso, declarar a classe como abstrata!

java.io

Classes abstratas não possuem nenhum segredo no aprendizado, mas quem está aprendendo orientação a objetos pode ter uma enorme dificuldade para saber quando utilizá-las, o que é muito normal.

Estudaremos o pacote `java.io`, que usa bastantes classes abstratas, sendo um exemplo real de uso desse recurso, que vai melhorar o entendimento delas. (classe `InputStream` e suas filhas)

## 9.5 - Para saber mais...

- Uma classe que estende uma classe normal também pode ser abstrata! Ela não poderá ser instanciada, mas sua classe pai sim!
- Uma classe abstrata não precisa necessariamente ter um método abstrato.

## 9.6 - Exercícios: Classes Abstratas

1) Repare que a nossa classe `Conta` é uma excelente candidata para uma classe abstrata. Por quê? Que métodos seriam interessantes candidatos a serem abstratos?

Transforme a classe `Conta` em abstrata, repare o que acontece no seu `main` já existente do `TestaContas`.

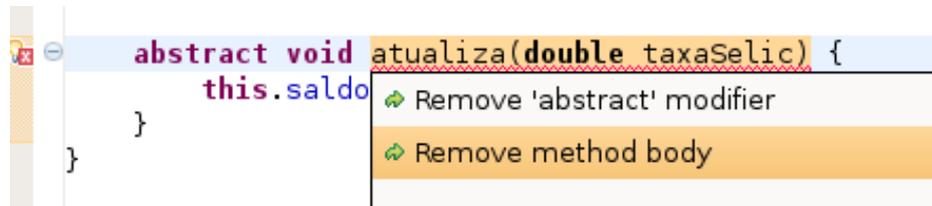
```
abstract class Conta {  
    // ...  
}
```

2) Para que o código do `main` volte a compilar, troque o `new Conta()` por `new ContaCorrente()`.

Se agora não podemos dar `new` em `Conta`, qual é a utilidade de ter um método que recebe uma referência a `Conta` como argumento? Aliás, posso ter isso?

3) Apenas pare efeito de entender melhor o `abstract`, remova (delete) o método `atualiza()` da `ContaPoupanca`, dessa forma ele herdará o método diretamente de `Conta`.

Transforme o método `atualiza()` da classe `Conta` em abstrato. Repare que, ao colocar a palavra chave `abstract` ao lado do método, o Eclipse rapidamente vai sugerir que você deve remover o corpo (body) do método com um quick fix:

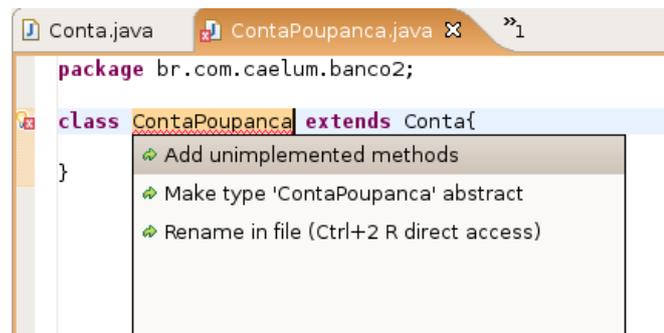


Sua classe Conta deve ficar parecido com:

```
abstract class Conta {
    // atributos e metodos que já existiam
    abstract void atualiza(double taxaSelic);
}
```

Qual é o problema com a classe ContaPoupanca?

- 4) Reescreva o método atualiza() na classe ContaPoupanca para que a classe possa compilar normalmente. O eclipse também sugere isso como um quick fix:



- 5) (opcional) Existe outra maneira de a classe ContaCorrente compilar se você não reescrever o método abstrato?
- 6) (opcional) Pra que ter o método atualiza na classe Conta se ele não faz nada? O que acontece se simplesmente apagarmos esse método da classe Conta e deixarmos o método atualiza nas filhas?
- 7) (opcional) Posso chamar um método abstrato de dentro de um outro método da própria classe abstrata? Um exemplo: o mostra do Funcionario pode invocar this.getBonificacao()?

## 9.7 - Desafios

- 1) Não podemos dar new em Conta, mas por que, então, podemos dar new em Conta[10], por exemplo?

# Orientação a Objetos - Interfaces

*“Uma imagem vale mil palavras. Uma interface vale mil imagens.”*  
– Ben Shneiderman

Ao término desse capítulo, você será capaz de:

- dizer o que é uma interface e as diferenças entre herança e implementação;
- escrever uma interface em Java;
- utilizá-las como um poderoso recurso para diminuir acoplamento entre as classes.

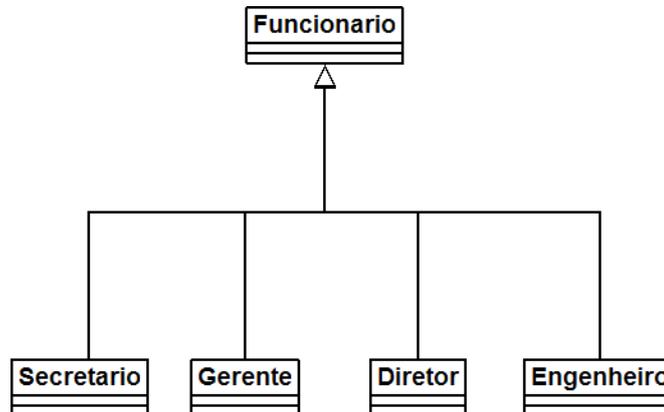
## 10.1 - Aumentando nosso exemplo

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores do Banco. Então, teríamos uma classe `Diretor`:

```
class Diretor extends Funcionario {  
  
    public boolean autentica(int senha) {  
        // verifica aqui se a senha confere com a recebida como parametro  
    }  
  
}
```

E a classe `Gerente`:

```
class Gerente extends Funcionario {  
  
    public boolean autentica(int senha) {  
        // verifica aqui se a senha confere com a recebida como parametro  
        // no caso do gerente verifica tambem se o departamento dele  
        // tem acesso  
    }  
  
}
```



Repare que o método de autenticação de cada tipo de `Funcionario` pode variar muito. Mas vamos aos problemas. Considere o `SistemaInterno` e seu controle: precisamos receber um `Diretor` ou `Gerente` como argumento, verificar se ele se autentica e colocá-lo dentro do sistema.

```

class SistemaInterno {

    void login(Funcionario funcionario) {
        // chamar o método autentica? não da! Nem todo Funcionario tem
    }
}
  
```

O `SistemaInterno` aceita qualquer tipo de `Funcionario`, tendo ele acesso ao sistema ou não, mas note que nem todo `Funcionario` possui o método `autentica`. Isso nos impede de chamar esse método com uma referência apenas a `Funcionario` (haveria um erro de compilação). O que fazer então?

```

class SistemaInterno {

    void login(Funcionario funcionario) {
        funcionario.autentica(); // não compila
    }
}
  
```

Uma possibilidade é criar dois métodos `login` no `SistemaInterno`: um para receber `Diretor` e outro para receber `Gerente`. Já vimos que essa não é uma boa escolha. Por quê?

```

class SistemaInterno {

    // design problemático
    void login(Diretor funcionario) {
        funcionario.autentica();
    }

    // design problemático
    void login(Gerente funcionario) {
        funcionario.autentica();
    }
}
  
```

Cada vez que criarmos uma nova classe de `Funcionario` que é *autenticável*, precisaríamos adicionar um novo método de login no `SistemaInterno`.

#### Métodos com mesmo nome

Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isto é, que exista uma maneira de distinguir no momento da chamada.

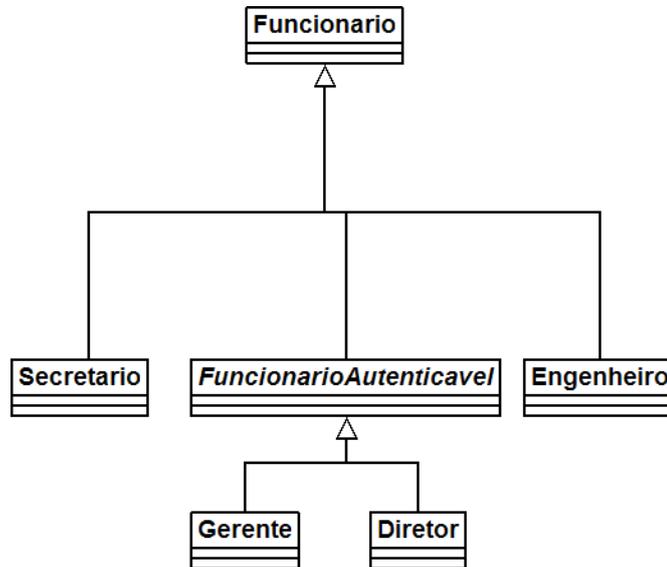
Isso se chama **sobrecarga** de método. (**Overloading**. Não confundir com **overriding**, que é um conceito muito mais poderoso, no caso).

Uma solução mais interessante seria criar uma classe no meio da árvore de herança, `FuncionarioAutenticavel`:

```
class FuncionarioAutenticavel extends Funcionario {  
  
    public boolean autentica(int senha) {  
        // faz autenticacao padrao  
    }  
  
    // outros atributos e metodos  
  
}
```

As classes `Diretor` e `Gerente` passariam a estender de `FuncionarioAutenticavel`, e o `SistemaInterno` receberia referências desse tipo, como a seguir:

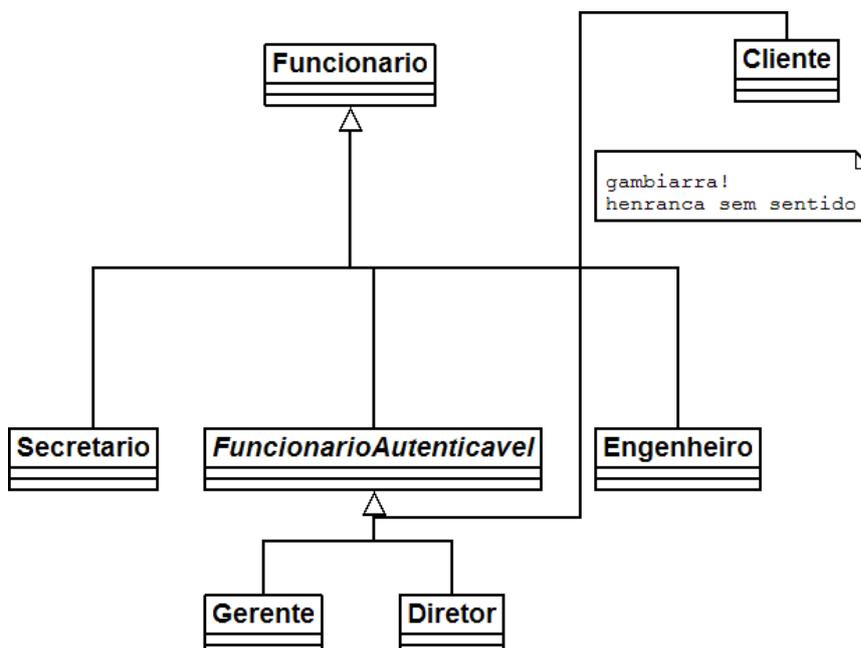
```
class SistemaInterno {  
  
    void login(FuncionarioAutenticavel fa) {  
  
        int senha = //pega senha de um lugar, ou de um scanner de polegar  
  
        // aqui eu posso chamar o autentica!  
        // Pois todo FuncionarioAutenticavel tem  
        boolean ok = fa.autentica(senha);  
  
    }  
  
}
```



Repare que `FuncionarioAutenticavel` é uma forte candidata a classe abstrata. Mais ainda, o método `autentica` poderia ser um método abstrato.

O uso de herança resolve esse caso, mas vamos a uma outra situação um pouco mais complexa: precisamos que todos os clientes também tenham acesso ao `SistemaInterno`. O que fazer? Uma opção é criar outro método `login` em `SistemaInterno`: mas já descartamos essa anteriormente.

Uma outra, que é comum entre os novatos, é fazer uma herança sem sentido para resolver o problema, por exemplo, fazer `Cliente` extends `FuncionarioAutenticavel`. Realmente, resolve o problema, mas trará diversos outros. `Cliente` definitivamente **não é** `FuncionarioAutenticavel`. Se você fizer isso, o `Cliente` terá, por exemplo, um método `getBonificacao`, um atributo `salario` e outros membros que não fazem o menor sentido para esta classe! Não faça herança quando a relação não é estritamente “é um”.



Como resolver essa situação? Note que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar/desenhar bem a nossa estrutura de classes.

## 10.2 - Interfaces

O que precisamos para resolver nosso problema? Arranjar uma forma de poder referenciar `Diretor`, `Gerente` e `Cliente` de uma mesma maneira, isto é, achar um fator comum.

Se existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema.

Toda classe define 2 ítems:

- o que uma classe faz (as assinaturas dos métodos)
- como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

Podemos criar um “contrato” que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

contrato `Autenticavel`:

```
quem quiser ser Autenticavel precisa saber fazer:  
1. autenticar dada uma senha, devolvendo um booleano
```

Quem quiser, pode “assinar” esse contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um `Gerente` assinar esse contrato, podemos nos referenciar a um `Gerente` como um `Autenticavel`.

Podemos criar esse contrato em Java!

```
interface Autenticavel {  
  
    boolean autentica(int senha);  
  
}
```

Chama-se `interface` pois é a maneira pela qual poderemos conversar com um `Autenticavel`. Interface é a maneira através da qual conversamos com um objeto.

Lemos a interface da seguinte maneira: “*quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano*”. Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe **o que o objeto deve fazer**, e não **como ele faz**, nem **o que ele tem**. **Como ele faz** vai ser definido em uma **implementação** dessa interface.

E o `Gerente` pode “assinar” o contrato, ou seja, **implementar** a interface. No momento em que ele implementa essa interface, ele precisa escrever os métodos pedidos pela interface (muito parecido com o efeito de herdar métodos abstratos, aliás, métodos de uma interface são públicos e abstratos, sempre). Para implementar usamos a palavra chave `implements` na classe:

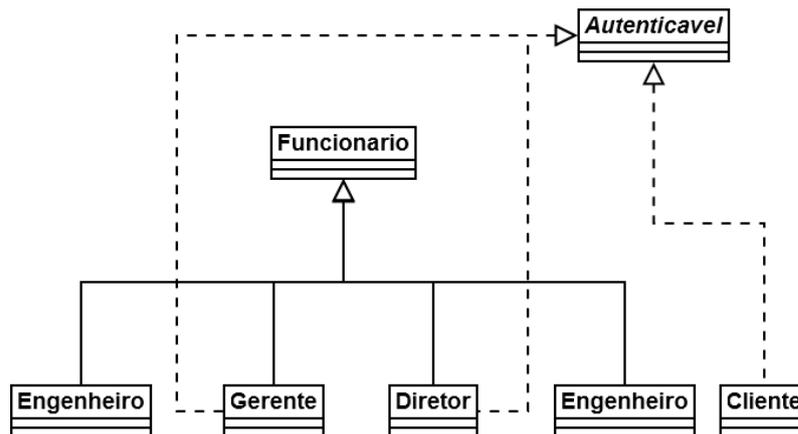
```
class Gerente extends Funcionario implements Autenticavel {

    private int senha;

    // outros atributos e métodos

    public boolean autentica(int senha) {
        if(this.senha != senha) {
            return false;
        }
        // pode fazer outras possiveis verificacoes, como saber se esse
        // departamento do gerente tem acesso ao Sistema

        return true;
    }
}
```



O implements pode ser lido da seguinte maneira: “A classe Gerente se compromete a ser tratada como Autenticavel, sendo obrigada a ter os métodos necessários, definidos neste contrato”.

A partir de agora, podemos tratar um Gerente como sendo um Autenticavel. Ganhamos polimorfismo! Temos mais uma forma de referenciar a um Gerente. Quando crio uma variável do tipo Autenticavel, estou criando uma referência para qualquer objeto de uma classe que implementa Autenticavel, direta ou indiretamente:

```
Autenticavel a = new Gerente();
//posso aqui chamar o metodo autentica!
```

Novamente, a utilização mais comum, aqui, seria receber como argumento.

Voltamos ao nosso SistemaInterno:

```
class SistemaInterno {
    void login(Autenticavel a) {
        int senha = //pega senha de um lugar, ou de um scanner de polegar
        boolean ok = fa.autentica(senha);
        // aqui eu posso chamar o autentica!
    }
}
```

```

    // não necessariamente é um Funcionario! Mais ainda, eu não sei
    // que objeto a referência "a" está apontando exatamente! Flexibilidade.
}
}

```

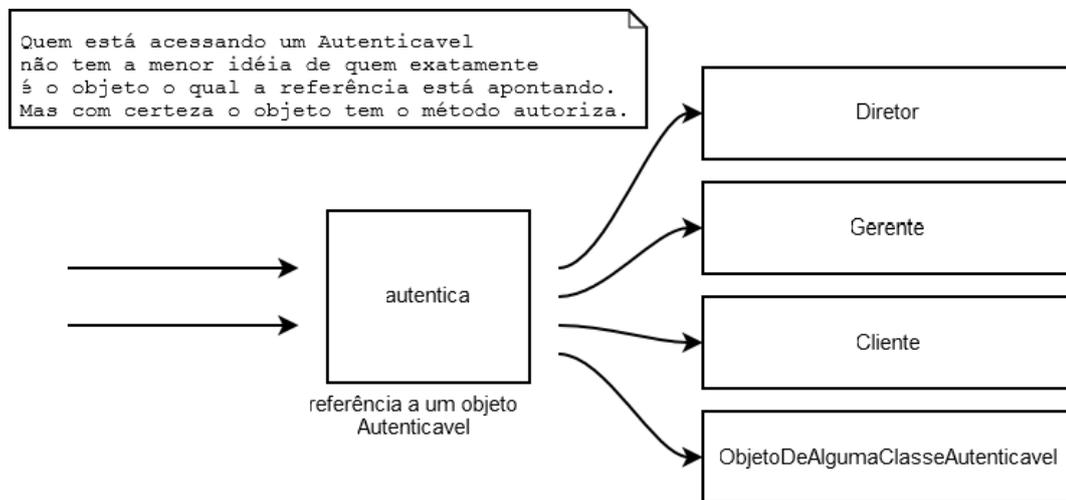
Pronto! E já podemos passar qualquer Autenticavel para o SistemaInterno. Então precisamos fazer com que o Diretor também implemente essa interface.

```

class Diretor extends Funcionario implements Autenticavel {

    // metodos e atributos, alem de obrigatoriamente ter o autentica
}

```



Agora podemos passar um Diretor. No dia em que tivermos mais um funcionário com acesso ao sistema, basta que ele implemente essa interface, para se encaixar no sistema.

Qualquer Autenticavel passado para o SistemaInterno está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método autentica que é o necessário para nosso SistemaInterno funcionar corretamente. Aliás, qualquer outra classe que futuramente implemente essa interface poderá ser passada como argumento aqui.

```

Autenticavel diretor = new Director();
Autenticavel gerente = new Gerente();

```

Ou, se agora achamos que o Fornecedor precisa ter acesso: basta que ele implemente Autenticavel. Olhe só o tamanho do desacoplamento: quem escreveu o SistemaInterno só precisa saber que ele é Autenticavel.

```

class SistemaInterno {
    void login(Autenticavel a) {
        // não importa se ele é um gerente ou diretor
        // será que é um fornecedor?
        // Eu, o programador do SistemaInterno, não me preocupo
        // Invocarei o método autentica
    }
}

```

}

Não faz diferença se é um `Diretor`, `Gerente`, `Cliente` ou qualquer classe que venha por aí. Basta seguir o contrato! Mais ainda, cada `Autenticavel` pode se autenticar de uma maneira completamente diferente de outro.

Lembre-se: a interface define que todos vão saber se autenticar (o que ele faz), enquanto a implementação define como exatamente vai ser feito (como ele faz).

A maneira como os objetos se comunicam num sistema orientado a objetos é muito mais importante do que como eles executam. **O que um objeto faz** é mais importante do que **como ele faz**. Aqueles que seguem essa regra, terão sistemas mais fáceis de manter e modificar. Como você já percebeu, esta é uma das idéias principais que queremos passar e, provavelmente, a mais importante de todo esse curso.

#### Herança entre interfaces

Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende que outros contratos sejam fechados antes deste valer. Você não herda métodos e atributos, mas sim responsabilidades.

### 10.3 - Dificuldade no aprendizado de interfaces

Interfaces representam uma barreira no aprendizado do Java: parece que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do método) escrita nas nossas classes implementadoras. Essa é uma maneira errada de se pensar. O objetivo do uso de uma interface é deixar seu código mais flexível e possibilitar a mudança de implementação sem maiores traumas. **Não é apenas um código de prototipação, um cabeçalho!**

Os mais radicais dizem que toda classe deve ser “interfaceada”, isto é, só devemos nos referir a objetos através de suas interfaces. Se determinada classe não tem uma interface, ela deveria ter. Os autores deste material acham tal medida radical demais, porém o uso de interfaces em vez de herança é amplamente aconselhado. Você pode encontrar mais informações sobre o assunto nos livros *Design Patterns*, *Refactoring* e *Effective Java*.

No livro *Design Patterns*, logo no início, os autores citam 2 regras “de ouro”. Uma é “evite herança, prefira composição” e a outra, “ programe voltado a interface e não à implementação”.

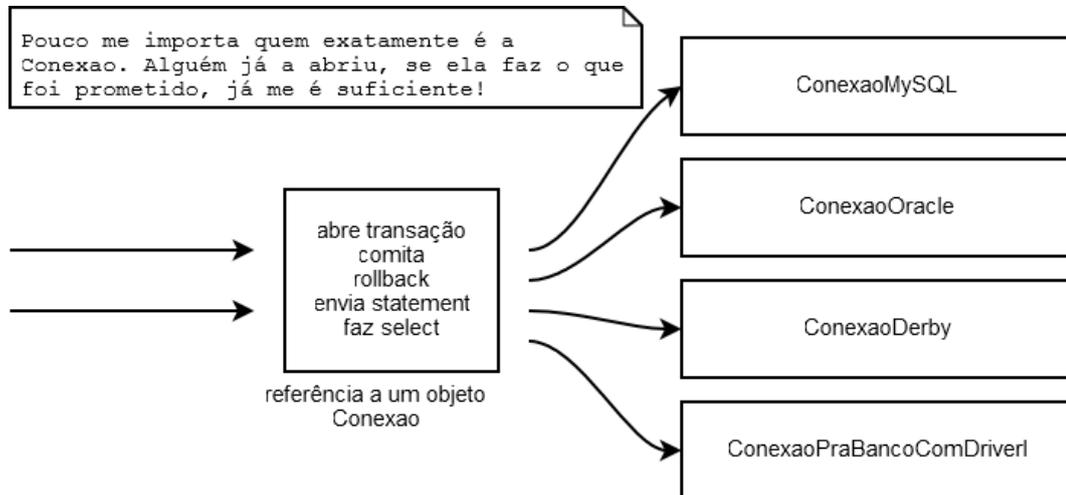
Veremos o uso de interfaces no capítulo de coleções, o que melhora o entendimento do assunto. O exemplo da interface `Comparable` também é muito esclarecedor, onde enxergamos o reaproveitamento de código através das interfaces, além do encapsulamento. Para o método `Collections.sort()`, pouco importa quem vai ser passado como argumento. Para ele, basta que a coleção sejam de objetos comparáveis. Ele pode ordenar `Elefante`, `Conexao` OU `ContaCorrente`, desde que implementem `Comparable`.

### 10.4 - Exemplo interessante: conexões com o banco de dados

Como fazer com que todas as chamadas para bancos de dados diferentes respeitem a mesma regra? Usando interfaces!

Imagine uma interface `Conexao` contendo todos os métodos necessários para a comunicação e troca de dados com um banco de dados. Cada banco de dados fica encarregado de criar a sua implementação para essa interface.

Quem for usar uma *Conexao* não precisa se importar com qual objeto exatamente está trabalhando, já que ele vai cumprir o papel que toda *Conexao* deve ter. Não importa se é uma conexão com um Oracle ou MySQL.



Apesar do `java.sql.Connection` não trabalhar bem assim, a idéia é muito similar, porém as conexões vêm de uma *factory* chamada `DriverManager`.

Conexão a banco de dados está fora do escopo desse treinamento, mas é um dos primeiros tópicos abordados no curso FJ-21, juntamente com DAO.

## 10.5 - Um pouco mais...

- 1) Posso substituir toda minha herança por interfaces? Qual é a vantagem e a desvantagem?
- 2) Uma interface também pode declarar constantes (não atributos de objeto). Qual é a utilidade?

## 10.6 - Exercícios: Interfaces

- 1) A sintaxe do uso de interfaces pode parecer muito estranha, à primeira vista. Vamos começar com um exercício para praticar a sintaxe:

```
interface AreaCalculavel {
    double calculaArea();
}
```

Queremos, agora, criar algumas classes que são `AreaCalculavel`:

```
class Quadrado implements AreaCalculavel {
    private int lado;

    public Quadrado(int lado) {
        this.lado = lado;
    }

    public double calculaArea() {
```

```
        return this.lado * this.lado;
    }
}

class Retangulo implements AreaCalculavel {
    private int largura;
    private int altura;

    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }

    public double calculaArea() {
        return this.largura * this.altura;
    }
}
```

Repare que, aqui, se você tivesse usado herança, não ia ganhar muita coisa, já que cada implementação é totalmente diferente da outra: um Quadrado, um Retangulo e um Circulo têm atributos e métodos **bem** diferentes.

Mas, mesmo que eles tivessem atributos em comum, utilizar interfaces é uma maneira muito mais elegante de modelar suas classes. Elas também trazem vantagens em não acoplar as classes. Uma vez que herança através de classes traz muito acoplamento, muitos autores clássicos dizem que, em muitos casos, **herança quebra o encapsulamento**, pensamento com o qual os autores dessa apostila concordam plenamente.

Crie uma classe de Teste. Repare no polimorfismo. Poderíamos estar passando esses objetos como argumento para alguém que aceitasse `AreaCalculavel` como argumento:

```
class Teste {
    public static void main(String[] args) {
        AreaCalculavel a = new Retangulo(3,2);
        System.out.println(a.calculaArea());
    }
}
```

Opcionalmente, crie a classe `Circulo`:

```
class Circulo implements AreaCalculavel {
    // ... atributos (raio) e métodos (calculaArea)
}
```

Utilize `Math.PI * raio * raio` para calcular a área.

- 2) Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso, vamos criar uma interface:

```
interface Tributavel {
    double calculaTributos();
}
```

Lemos essa interface da seguinte maneira: “todos que quiserem ser *tributável* precisam saber *calcular tributos*, devolvendo um `double`”.

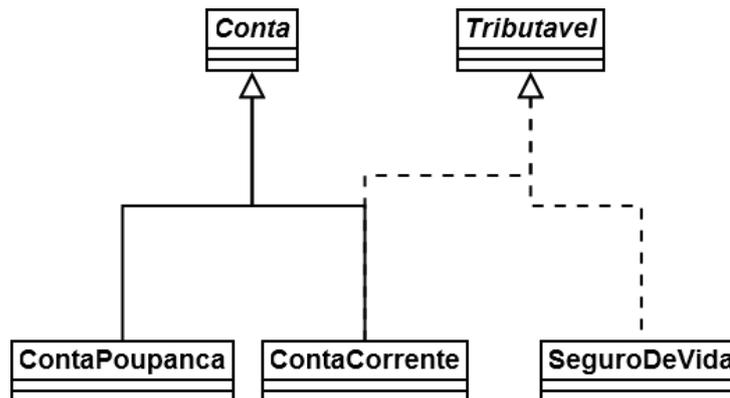
Alguns bens são tributáveis e outros não, ContaPoupanca não é tributável, já para ContaCorrente você precisa pagar 1% da conta e o SeguroDeVida tem uma taxa fixa de 42 reais. (faça a mudança em ContaCorrente e crie a classe SeguroDeVida):

```
class ContaCorrente extends Conta implements Tributavel {

    // outros atributos e metodos

    public double calculaTributos() {
        return this.saldo * 0.01;
    }
}

class SeguroDeVida implements Tributavel {
    public double calculaTributos() {
        return 42;
    }
}
```



Vamos criar uma classe TestaTributavel com um método main para testar o nosso exemplo:

```
class TestaTributavel {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        Tributavel t = cc;
        System.out.println(t.calculaTributos());
    }
}
```

Tente, agora, chamar o método getSaldo através da referência t, o que ocorre? Por quê?

A linha em que atribuímos cc a um Tributavel é apenas para você enxergar que é possível fazê-lo. Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade será útil para o próximo exercício.

- 3) (opcional) Crie um GerenciadorDeImpostoDeRenda, que recebe todos os tributáveis de uma pessoa e soma seus valores, e inclua nele um método para devolver seu total:

```
class GerenciadorDeImpostoDeRenda {
    private double total;
```

```
void adiciona(Tributavel t) {
    System.out.println("Adicionando tributavel: " + t);

    this.total = this.total + t.calculaTributos();
}

public double getTotal() {
    return total;
}
}
```

Crie um `main` para instanciar diversas classes que implementam `Tributavel` e passar como argumento para um `GerenciadorDeImpostoDeRenda`. Repare que você não pode passar qualquer tipo de conta para o método `adiciona`, apenas a que implementa `Tributavel`. Além disso, pode passar o `SeguroDeVida`.

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

        GerenciadorDeImpostoDeRenda gerenciador = new GerenciadorDeImpostoDeRenda();

        SeguroDeVida sv = new SeguroDeVida();
        gerenciador.adiciona(sv);

        ContaCorrente cc = new ContaCorrente();
        cc.deposita(1000);
        gerenciador.adiciona(cc);

        System.out.println(gerenciador.getTotal());
    }
}
```

Repare que, de dentro do `GerenciadorDeImpostoDeRenda`, você não pode acessar o método `getSaldo`, por exemplo, pois você não tem a garantia de que o `Tributavel` que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface `Tributavel`.

É interessante enxergar que as interfaces (como aqui, no caso, `Tributavel`) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, `SeguroDeVida` e `ContaCorrente` são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

Se amanhã o governo começar a tributar até mesmo `PlanoDeCapitalizacao`, basta que essa classe implemente a interface `Tributavel`! Repare no grau de desacoplamento que temos: a classe `GerenciadorDeImpostoDeRenda` nem imagina que vai trabalhar como `PlanoDeCapitalizacao`. Para ela, a única coisa que importa é que o objeto respeite o contrato de um tributável, isso é, a interface `Tributavel`. Novamente: programe voltado a interface, não a implementação.

- 4) (opcional) Use o método `printf` para imprimir o saldo com exatamente duas casas decimais:

```
System.out.printf("O saldo é: %.2f", conta.getSaldo());
```

## 10.7 - Exercícios avançados

- 1) (Opcional) Transforme a classe `Conta` em uma interface. Atenção: faça isso num projeto à parte pois usaremos a `Conta` como classe em exercícios futuros.

```
interface Conta {
    double getSaldo();
    void deposita(double valor);
    void retira(double valor);
    void atualiza(double taxaSelic);
}
```

Adapte `ContaCorrente` e `ContaPoupanca` para essa modificação:

```
class ContaCorrente implements Conta {
    // ...
}

class ContaPoupanca implements Conta {
    // ...
}
```

Algum código vai ter de ser copiado e colado? Isso é tão ruim?

- 2) (Opcional) Subinterfaces:

Às vezes, é interessante criarmos uma interface que herda de outras interfaces.

```
interface ContaTributavel extends Conta, Tributavel {
}
```

Dessa maneira, quem for implementar essa nova interface precisa implementar todos os métodos herdados das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

```
class ContaCorrente implements ContaTributavel {
    // metodos
}
```

```
Conta c = new ContaCorrente();
Tributavel t = new ContaCorrente();
```

Repare que o código pode parecer estranho, pois a interface não declara método algum, só herda os métodos abstratos declarados nas outras interfaces.

Ao mesmo tempo que uma interface pode herdar de mais de uma outra interface, classes só podem possuir uma classe mãe (herança simples).

## 10.8 - Discussão em aula: Favorecer composição em relação à herança

Discuta com o instrutor e seus colegas, alternativas à herança. Falaremos de herança versus composição e porque herança é muitas vezes considerada maléfica.

Numa entrevista, James Gosling, “pai do java”, fala sobre uma linguagem puramente de delegação e chega a dizer:.

*Rather than subclassing, just use pure interfaces. It's not so much that class inheritance is particularly bad. It just has problems.* - James Gosling, na mesma entrevista.

<http://www.artima.com/intv/gosling3P.html>

# Controlando os erros com Exceções

*“Quem pensa pouco, erra muito”*  
– Leonardo da Vinci

Ao término desse capítulo, você será capaz de:

- controlar erros e tomar decisões baseadas nos mesmos;
- criar novos tipos de erros para melhorar o tratamento deles em sua aplicação ou biblioteca;
- assegurar que um método funcionou como diz em seu “contrato”.

## 11.1 - Motivação

Voltando às `Contas` que criamos no capítulo 6, o que aconteceria ao tentar chamar o método `saca` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca` não saberá que isso aconteceu.

Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Em Java, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário de que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma `Conta` a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
//      o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como `boolean` e retornar `true`, se tudo ocorreu da maneira planejada, ou `false`, caso contrário:

```
boolean saca(double quantidade) {
    if (quantidade > this.saldo + this.limite) { //posso sacar até      saldo+limite
        System.out.println("Não posso sacar fora do limite!");
        return false;
    } else {
```

```
        this.saldo = this.saldo - quantidade;
        return true;
    }
}
```

Um novo exemplo de chamada ao método acima:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
    System.out.println("Não saquei");
}
```

Mas e se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução é alterar o retorno de `boolean` para `int` e retornar o código do erro que ocorreu. Isso é considerado uma má prática (conhecida também como uso de “*magic numbers*”).

Além de você perder o retorno do método, o valor retornado é “mágico” e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando.

Repare o que aconteceria se fosse necessário retornar um outro valor. O exemplo abaixo mostra um caso onde, através do retorno, não será possível descobrir se ocorreu um erro ou não, pois o método retorna um cliente.

```
public Cliente procuraCliente(int id) {

    if (idInvalido) {
        // avisa o método que chamou este que ocorreu um erro
    } else {
        Cliente cliente = new Cliente();
        cliente.setId(id);
        // cliente.setNome(.....);
        return cliente;
    }
}
```

Por esse e outros motivos, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do argumento do saque inválido ou do `id` inválido de um cliente é uma **exceção** à regra.

### Exceção

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

## 11.2 - Exercício para começar com os conceitos

1) Para aprendermos os conceitos básicos das exceptions do Java, teste o seguinte código você mesmo:

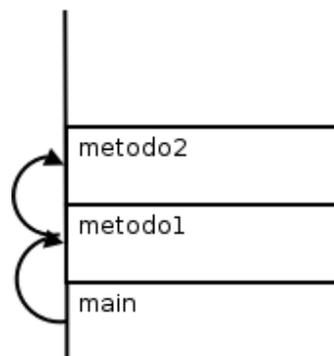
```
class TesteErro {
    public static void main(String[] args) {
        System.out.println("inicio do main");
        metodo1();
        System.out.println("fim do main");
    }

    static void metodo1() {
        System.out.println("inicio do metodo1");
        metodo2();
        System.out.println("fim do metodo1");
    }

    static void metodo2() {
        System.out.println("inicio do metodo2");
        int[] array = new int[10];
        for(int i = 0; i <= 15; i++) {
            array[i] = i;
            System.out.println(i);
        }
        System.out.println("fim do metodo2");
    }
}
```

Repare o método `main` chamando `metodo1` e esse, por sua vez, chamando o `metodo2`. Cada um desses métodos pode ter suas próprias variáveis locais, sendo que, por exemplo, o `metodo1` não enxerga as variáveis declaradas dentro do `main`.

Como o Java (e muitas das outras linguagens) faz isso? Toda invocação de método é empilhada... em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (stack). Basta jogar fora um gomo da pilha (stackframe):



Porém, o nosso `metodo2` propositalmente possui um enorme problema: está acessando um índice de array indevida para esse caso: o índice estará fora dos limites da array quando chegar em 10!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?

```
Console X
<terminated> Teste [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:44:42 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Teste.metodo2(Teste.java:18)
    at Teste.metodo1(Teste.java:10)
    at Teste.main(Teste.java:4)
```

Essa é o conhecido **rastro da pilha** (*stacktrace*). É uma saída importantíssima para o programador - tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa *stacktrace*.

Por que isso aconteceu? O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é **lançada** (*thrown*) a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o `metodo2` não toma nenhuma medida diferente do que vimos até agora.

Como o `metodo2` não está **tratando** esse problema, a JVM pára a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* pra baixo, onde será feita nova verificação: o `metodo1` está se precavendo de um problema chamado `ArrayIndexOutOfBoundsException`? Não... volta para o `main`, onde também não há proteção, então a JVM morre (na verdade, quem morre é apenas a `Thread` corrente, veremos mais para frente).

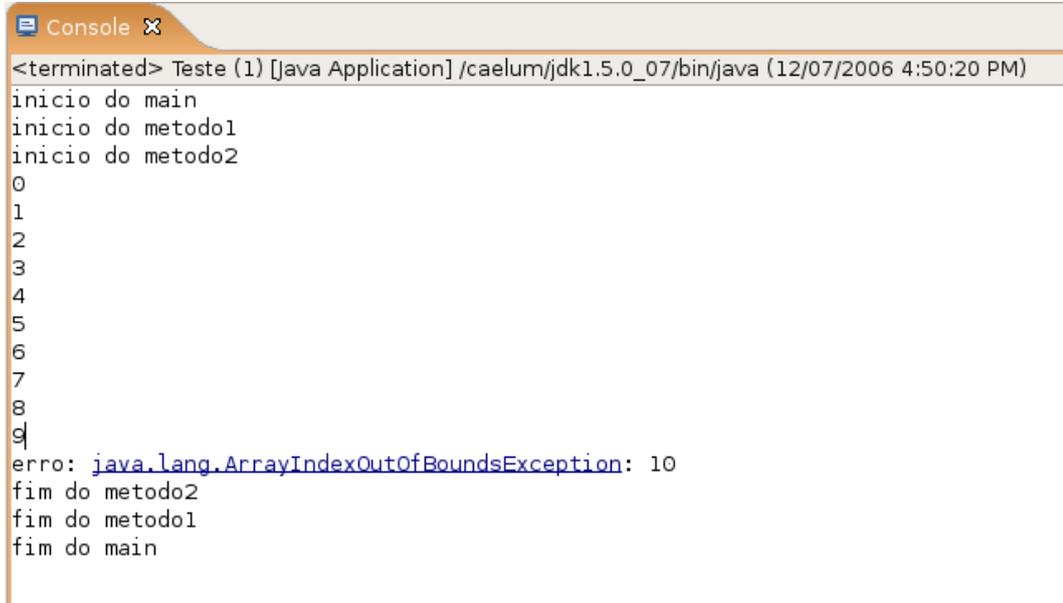
Obviamente, aqui estamos forçando o erro, e não faria sentido tomarmos cuidado com ele. Seria fácil arrumar um problema desses. Basta navegarmos na array no máximo até o seu `length`.

Porém, apenas para entender o controle de fluxo de uma `Exception`, vamos colocar o código que vai **tentar** (*try*) executar o bloco perigoso e, caso o problema seja do tipo `ArrayIndexOutOfBoundsException`, ele será **pego** (*caught*). Repare que é interessante que cada exceção no Java tenha um tipo... ela pode ter atributos e métodos.

- 2) Adicione um `try/catch` em volta do `for`, pegando `ArrayIndexOutOfBoundsException`. O que o código imprime agora?

```
try {
    for(int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
```

}



```
Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:50:20 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo2
fim do metodo1
fim do main
```

Em vez de fazer o try em torno do for inteiro, tente apenas com o bloco de dentro do for:

```
for(int i = 0; i <= 15; i++) {
    try {
        array[i] = i;
        System.out.println(i);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
}
```

Qual é a diferença?

```
Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:52:43 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
erro: java.lang.ArrayIndexOutOfBoundsException: 11
erro: java.lang.ArrayIndexOutOfBoundsException: 12
erro: java.lang.ArrayIndexOutOfBoundsException: 13
erro: java.lang.ArrayIndexOutOfBoundsException: 14
erro: java.lang.ArrayIndexOutOfBoundsException: 15
fim do metodo2
fim do metodo1
fim do main
```

Agora retire o try/catch e coloque ele em volta da chamada do `metodo2`.

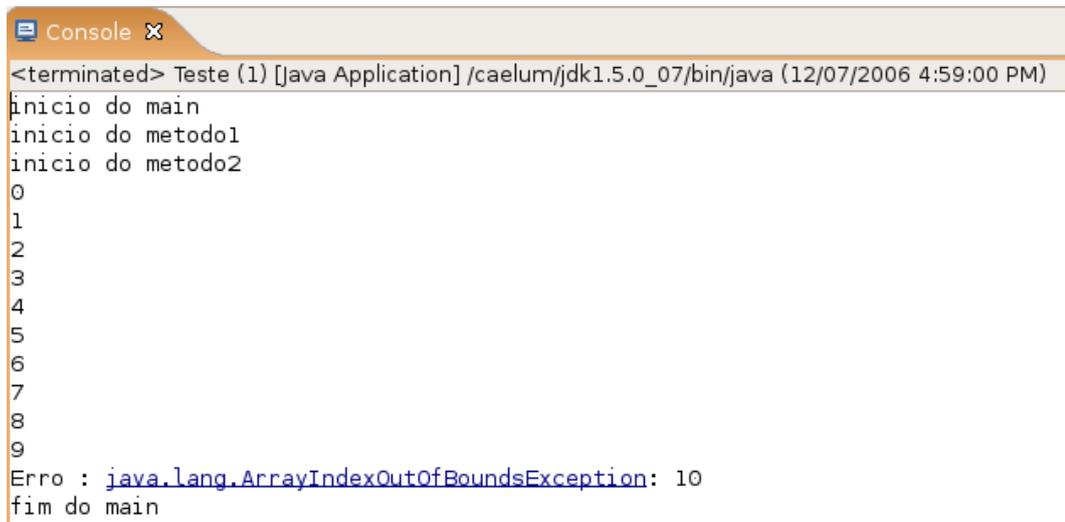
```
System.out.println("inicio do metodo1");
try {
    metodo2();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");
```

```
Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:56:54 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo1
fim do main
```

Faça a mesma coisa, retirando o try/catch novamente e colocando em volta da chamada do `metodo1`. Rode

os códigos, o que acontece?

```
System.out.println("inicio do main");
try {
    metodo1();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erro : "+e);
}
System.out.println("fim do main");
```



```
Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:59:00 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Erro : java.lang.ArrayIndexOutOfBoundsException: 10
fim do main
```

Repare que, a partir do momento que uma exception foi *caught* (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

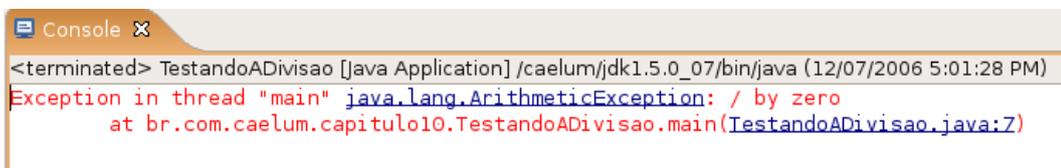
## 11.3 - Exceções de Runtime mais comuns

Que tal tentar dividir um número por zero? Será que o computador consegue fazer aquilo que nós definimos que não existe?

```
public class TestandoADivisao {

    public static void main(String args[]) {
        int i = 5571;
        i = i / 0;
        System.out.println("O resultado " + i);
    }
}
```

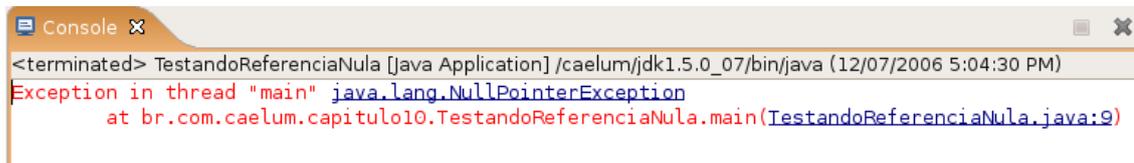
Tente executar o programa acima. O que acontece?



```
Console X
<terminated> TestandoADivisao [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:01:28 PM)
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at br.com.caelum.capitulo10.TestandoADivisao.main(TestandoADivisao.java:7)
```

```
public class TestandoReferenciaNula {  
  
    public static void main(String args[]) {  
        Conta c = null;  
        System.out.println("Saldo atual " + c.getSaldo());  
    }  
}
```

Agora, tente executar este programa. O que acontece?



Repare que um `ArrayIndexOutOfBoundsException` ou um `NullPointerException` poderia ser facilmente evitado com o `for` corretamente escrito ou com `ifs` que checariam os limites da array.

Outro caso em que também ocorre tal tipo de exceção é quando um cast errado é feito (veremos mais pra frente). Em todos os casos, tais erros provavelmente poderiam ser evitados pelo programador. É por esse motivo que o Java não te obriga a dar o `try/catch` nessas exceções e chamamos essas exceções de *unchecked* (em outras palavras, o compilador não checa se você está tratando essas exceções).

#### Erros

Os erros em Java são um tipo de exceção que também pode ser tratado. Eles representam problemas na máquina virtual e não devem ser tratados em 99% dos casos, já que provavelmente o melhor a se fazer é deixar a JVM encerrar (ou apenas a Thread em questão).

## 11.4 - Outro tipo de exceção: Checked Exceptions

Fica claro, com os exemplos de código acima, que não é necessário declarar que você está tentando fazer algo onde um erro possa ocorrer. Os dois exemplos, com ou sem o `try/catch`, compilaram e rodaram. Em um, o erro terminou o programa e, no outro, foi possível tratá-lo.

Mas não é só esse tipo de exceção que existe em Java. Um outro tipo, obriga a quem chama o método ou construtor a tratar o erro. Um exemplo que podemos mostrar agora, é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, **não** se preocupe com isto agora):

```
public static void metodo() {  
  
    new java.io.FileReader("arquivo.txt");  
}
```

O código acima não compila e o compilador avisa que é necessário tratar o `FileNotFoundException` que pode ocorrer:

```
Teste.java:3: unreported exception java.io.FileNotFoundException; must be caught
  or declared to be thrown
      new java.io.FileReader("arquivo.txt");
      ^
1 error
```

Para compilar e fazer o programa funcionar, precisamos tratar o erro de um de dois jeitos. O primeiro, é tratá-lo com o try e catch do mesmo jeito que usamos no exemplo anterior, com uma array:

```
public static void metodo() {

    try {
        new java.io.FileReader("arquivo.txt");
    } catch (java.io.FileNotFoundException e) {
        System.out.println("Nao foi possivel abrir o arquivo para leitura");
    }

}
```

A segunda forma de tratar esse erro, é delegar ele para quem chamou o nosso método, isto é, passar para a frente.

```
public static void metodo() throws java.io.FileNotFoundException {

    new java.io.FileReader("arquivo.txt");

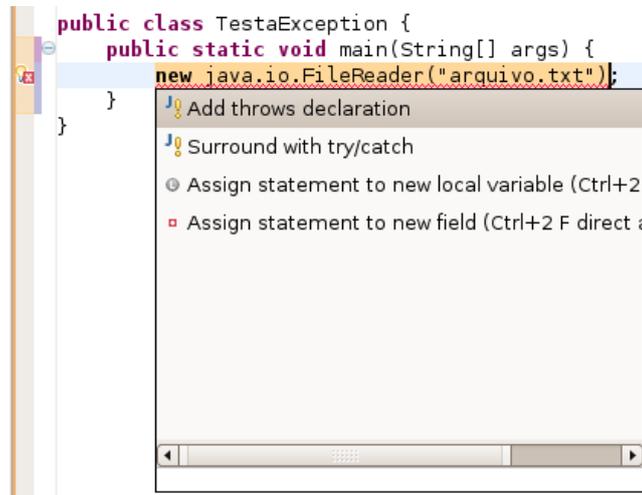
}
```

No Eclipse é bem simples fazer tanto um try/catch como um throws:

Tente digitar esse código no eclipse:

```
public class TestaException {
    public static void main(String[] args) {
        new java.io.FileReader("arquivo.txt");
    }
}
```

O Eclipse vai reclamar :



E você tem duas opções:

- 1) Add throws declaration, que vai gerar:

```
public class TestaException {
    public static void main(String[] args) throws FileNotFoundException {
        new java.io.FileReader("arquivo.txt");
    }
}
```

- 2) Surround with try/catch, que vai gerar:

```
public class TestaException2 {
    public static void main(String[] args) {
        try {
            new java.io.FileReader("arquivo.txt");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

No início, existe uma grande tentação de sempre passar o erro pra frente para outros tratarem dele. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

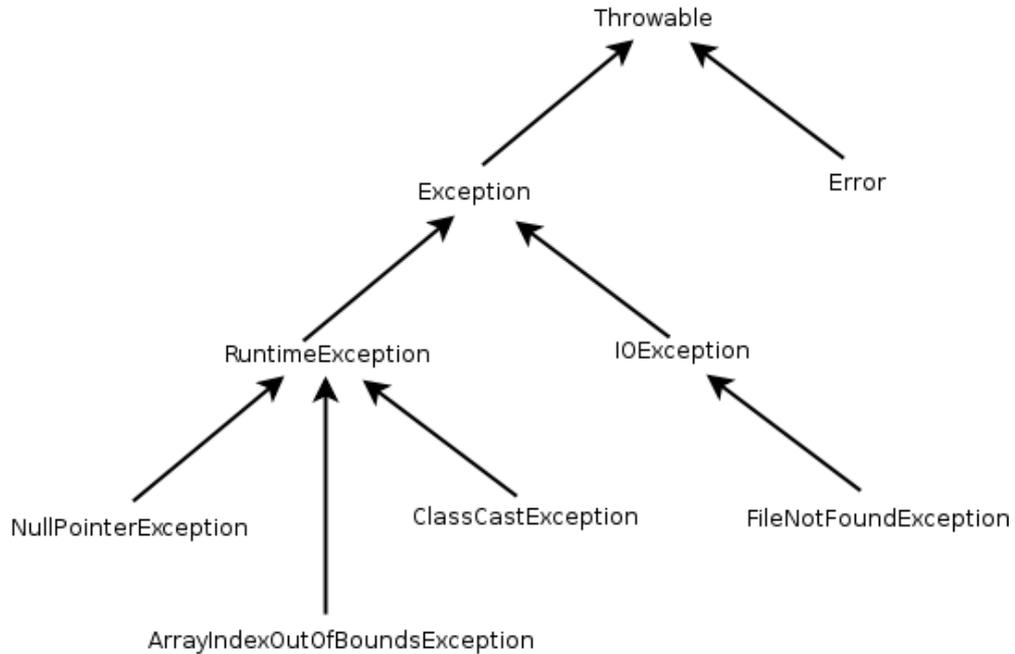
#### Boas práticas no tratamento de exceções

No blog da Caelum há um extenso artigo discutindo as boas práticas em relação ao tratamento de exceções.

<http://blog.caelum.com.br/2006/10/07/lidando-com-exceptions/>

## 11.5 - Um pouco da grande família Throwable

Uma pequena parte da Família Throwable:



## 11.6 - Mais de um erro

É possível tratar mais de um erro quase que ao mesmo tempo:

1) Com o try e catch:

```
try {
    objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
    // ..
} catch (SQLException e) {
    // ..
}
```

2) Com o throws:

```
public void abre(String arquivo) throws IOException, SQLException {
    // ..
}
```

3) Você pode, também, escolher tratar algumas exceções e declarar as outras no throws:

```
public void abre(String arquivo) throws IOException {
    try {
        objeto.metodoQuePodeLancarIOeSQLException();
    } catch (SQLException e) {
```

```
    // ..  
  }  
}
```

É desnecessário declarar no `throws` as exceptions que são unchecked, porém é permitido e, às vezes, facilita a leitura e a documentação do seu código.

## 11.7 - Lançando exceções

Lembre-se do método `saca` da nossa classe `Conta`. Ele devolve um `boolean` caso consiga ou não sacar:

```
boolean saca(double valor){  
    if (this.saldo < valor){  
        return false;  
    } else {  
        this.saldo-=valor;  
        return true;  
    }  
}
```

Podemos, também, lançar uma `Exception`, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um `if` no retorno de um método.

A palavra chave **throw** lança uma `Exception` (diferente de `throws`, que apenas avisa da possibilidade daquele método lançá-la).

```
void saca(double valor){  
    if (this.saldo < valor){  
        throw new RuntimeException();  
    } else {  
        this.saldo-=valor;  
    }  
}
```

No nosso caso, lança uma do tipo unchecked. `RuntimeException` é a exception mãe de todas as exceptions unchecked. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos, então, usar uma `Exception` mais específica:

```
void saca(double valor){  
    if (this.saldo < valor){  
        throw new IllegalArgumentException();  
    } else {  
        this.saldo-=valor;  
    }  
}
```

`IllegalArgumentException` diz um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma `Exception` unchecked pois estende de `RuntimeException` e já faz parte da biblioteca do java. (`IllegalArgumentException` é a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc).

E agora, para pegar esse erro, não usaremos um if/else e sim um try/catch, porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println("Saldo Insuficiente");
}
```

Podíamos melhorar ainda mais e passar para o construtor da IllegalArgumentException o motivo da exceção:

```
void saca(double valor){
    if (this.saldo < valor){
        throw new IllegalArgumentException("Saldo insuficiente");
    } else {
        this.saldo-=valor;
    }
}
```

O método getMessage() definido na classe Throwable (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da IllegalArgumentException.

```
try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

## 11.8 - O que colocar dentro do try?

Imagine que vamos sacar dinheiro de diversas contas:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

Conta cp = new ContaPoupanca();
cp.deposita(100);

// sacando das contas:

cc.saca(50);
System.out.println("consegui sacar da corrente!");

cp.saca(50);
System.out.println("consegui sacar da poupanca!");
```

Podemos escolher vários lugares para colocar try/catch:

```
try {
    cc.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
System.out.println("consegui sacar da corrente!");

try {
    cp.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
System.out.println("consegui sacar da poupanca!");
```

Essa não parece uma opção boa, pois a mensagem “consegui sacar” será impressa mesmo que o `catch` seja acionado. Sempre que temos algo que depende da linha de cima para ser correto, devemos agrupá-lo no `try`:

```
try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

try {
    cp.saca(50);
    System.out.println("consegui sacar da poupanca!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Mas há ainda uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```
try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
    cp.saca(50);
    System.out.println("consegui sacar da poupanca!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

**O que você vai colocar dentro do `try` influencia muito a execução do programa!** Pense direito nas linhas que dependem uma da outra para a execução correta da sua lógica de negócios.

## 11.9 - Criando seu próprio tipo de exceção

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções. Dessa maneira, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar a nossa:

Voltamos para o exemplo das Contas, vamos criar a nossa Exceção de SaldoInsuficienteException:

```
public class SaldoInsuficienteException extends RuntimeException{

    SaldoInsuficienteException(String message){
        super(message);
    }
}
```

E agora, em vez de lançar um `IllegalArgumentException`, vamos lançar nossa própria exception, com uma mensagem que dirá "Saldo Insuficiente":

```
void saca(double valor){
    if (this.saldo < valor){
        throw new SaldoInsuficienteException("Saldo Insuficiente, tente um valor menor");
    } else {
        this.saldo-=valor;
    }
}
```

E, para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
public static void main(String[] args) {
    Conta cc = new ContaCorrente();
    cc.deposita(10);

    try {
        cc.saca(100);
    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
    }
}
```

Podemos transformar essa `Exception` de `unchecked` para `checked`, obrigando, assim, quem chama esse método a dar `try-catch`, ou `throws`:

```
public class SaldoInsuficienteException extends Exception{

    SaldoInsuficienteException(String message){
        super(message);
    }
}
```

#### Catch e throws

Existe uma péssima prática de programação em java que é a de escrever o `catch` e o `throws` com `Exception`.

Existem códigos que sempre usam `Exception` pois isso cuida de todos os possíveis erros. O maior problema disso é generalizar o erro. Se alguém joga algo do tipo `Exception` para quem o chamou, quem recebe não sabe qual o tipo específico de erro ocorreu e não vai saber como tratar o mesmo.

## 11.10 - Para saber mais: finally

Os blocos `try` e `catch` podem conter uma terceira cláusula chamada `finally` que indica o que deve ser feito após o término do bloco `try` ou de um `catch` qualquer.

É interessante colocar algo que é imprescindível de ser executado, caso o que você queria fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no `finally`, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

No exemplo a seguir, o bloco `finally` será executado - não importa se tudo ocorrer ok ou com algum problema:

```
try {
    // bloco try
} catch (IOException ex) {
    // bloco catch 1
} catch (SQLException sqlEx) {
    // bloco catch2
} finally {
    // bloco finally
}
```

## 11.11 - Um pouco mais...

- 1) É possível criar um bloco `try` e `finally`, sem `catch`. Isso significa que, se não ocorrer erro algum, o bloco do `finally` será executado. Se ocorrer algum erro, ele também será executado, e o erro será jogado para quem chamou o método.
- 2) Procure informações sobre a palavra-chave `assert` e tente utilizar o código abaixo:

```
int i = 1;
// tenho certeza que i vale 1
assert i == 1;
i = 2;
// tenho certeza que i vale 2
assert i == 2;
i++;
// vai gerar um erro pois i não vale 4
assert i == 4;
```

## 11.12 - Exercícios: Exceções

- 1) Na classe `Conta`, modifique o método `deposita(double x)`: Ele deve lançar uma `exception` chamada `IllegalArgumentException`, que já faz parte da biblioteca do java, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

```
void deposita(double valor){
    if (valor < 0) {
        throw new IllegalArgumentException();
    }
}
```

```
    } else {  
        this.saldo += valor * 0.9962;  
    }  
}
```

- 2) Crie uma classe `TestaDeposita` com o método `main`. Crie uma `ContaPoupanca` e tente depositar valores inválidos:

```
public static void main(String[] args) {  
    Conta cc = new ContaPoupanca();  
    cc.deposita(-100);  
}
```

O que acontece? Uma `IllegalArgumentException` é lançada uma vez que tentamos depositar um valor inválido. Adicione o `try/catch` para tratar o erro:

```
public static void main(String[] args) {  
    Conta cc = new ContaPoupanca();  
  
    try {  
        cc.deposita(-100);  
    } catch( IllegalArgumentException e) {  
        System.out.println("Você tentou depositar um valor inválido");  
    }  
}
```

Atenção: se a sua classe `ContaCorrente` está reescrevendo o método `deposita` e não utiliza do `super.deposita`, ela não lançará a `exception` no caso do valor negativo! Você pode resolver isso utilizando o `super`, ou fazendo apenas o teste com `ContaPoupanca`.

- 3) Ao lançar a `IllegalArgumentException`, passe via construtor uma mensagem a ser exibida. Lembre que a `String` recebida como parâmetro é acessível depois via o método `getMessage()` herdado por todas as `Exceptions`.

```
void deposita(double valor){  
    if (valor < 0){  
        throw new IllegalArgumentException("Você tentou depositar um valor negativo");  
    } else {  
        this.saldo += valor * 0.9962;  
    }  
}
```

- 4) Agora, altere sua classe `TestaDeposita` para exibir a mensagem da exceção através da chamada do `getMessage()`:

```
public static void main(String[] args) {  
    Conta cc = new ContaPoupanca();  
  
    try {  
        cc.deposita(-100);  
    } catch(IllegalArgumentException e){  
        System.out.println(e.getMessage());  
    }  
}
```

```
}
```

- 5) Crie sua própria Exception, `ValorInvalidoException`. Para isso, você precisa criar uma classe com esse nome que estenda de `RuntimeException`.

```
class ValorInvalidoException extends RuntimeException {  
  
}
```

Lance-a em vez de `IllegalArgumentException`.

Atenção: nem sempre é interessante criarmos um novo tipo de exception! Depende do caso. Neste aqui, seria melhor ainda utilizarmos `IllegalArgumentException`.

- 6) (opcional) Coloque um construtor na classe `ValorInvalidoException` que receba valor inválido que ele tentou passar (isto é, ele vai receber um `double` valor).

Dessa maneira, na hora de dar o `throw new ValorInvalidoException` você vai precisar passar esse valor como argumento:

```
if (valor < 0){  
    throw new ValorInvalidoException(valor);  
}
```

- 7) (opcional) A classe `Exception` (na verdade, `Throwable`) tem definida nela um método `getMessage` que pode ser reescrito nas suas subclasses para falar um pouco mais sobre o erro. Reescreva-o na sua classe `ValorInvalidoException` para algo como:

```
public String getMessage() {  
    return "Não posso retirar valor negativo: " + valor;  
}
```

Agora, ao imprimir a exception, repare que essa mensagem é que vai aparecer.

- 8) (opcional) Declare a classe `ValorInvalidoException` como filha direta de `Exception` em vez de `RuntimeException`. O que acontece?

Você vai precisar avisar que o seu método `saca()` throws `ValorInvalidoException`, pois ela é uma checked exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try-catch` ou `throws`.

## 11.13 - Desafios

- 1) O que acontece se acabar a memória da java virtual machine? Como forçar isso?

# Pacotes - Organizando suas classes e bibliotecas

*“Uma discussão prolongada significa que ambas as partes estão erradas”*  
– Voltaire

Ao término desse capítulo, você será capaz de:

- separar suas classes em pacotes e
- preparar arquivos simples para distribuição.

## 12.1 - Organização

Quando um programador utiliza as classes feitas por outro, surge um problema clássico: como escrever duas classes com o mesmo nome?

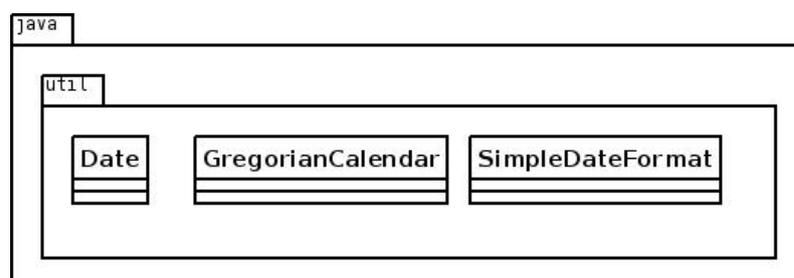
Por exemplo: pode ser que a minha classe de `Data` funcione de um certo jeito, e a classe `Data` de um colega, de outro jeito. Pode ser que a classe de `Data` de uma **biblioteca** funcione ainda de uma terceira maneira diferente.

Como permitir que tudo isso realmente funcione? Como controlar quem quer usar qual classe de `Data`?

Pensando um pouco mais, notamos a existência de um outro problema e da própria solução: o sistema operacional não permite a existência de dois arquivos com o mesmo nome sob o mesmo diretório, portanto precisamos organizar nossas classes em diretórios diferentes.

Os diretórios estão diretamente relacionados aos chamados **pacotes** e costumam agrupar classes de funcionalidades similares ou relacionadas.

Por exemplo, no pacote `java.util` temos as classes `Date`, `SimpleDateFormat` e `GregorianCalendar`; todas elas trabalham com datas de formas diferentes.



## 12.2 - Diretórios

Se a classe `Cliente` está no pacote `banco`, ela deverá estar no diretório com o mesmo nome: `banco`. Se ela se localiza no pacote `br.com.caelum.banco`, significa que está no diretório `br/com/caelum/banco`.

A classe `Cliente`, que se localiza nesse último diretório mencionado, deve ser escrita da seguinte forma:

```
package br.com.caelum.banco;

class Cliente {
    // ...
}
```

Fica fácil notar que a palavra chave `package` indica qual o pacote/diretório contém esta classe.

Um pacote pode conter nenhum ou mais subpacotes e/ou classes dentro dele.

### Padrão da nomenclatura dos pacotes

O padrão da sun para dar nome aos pacotes é relativo ao nome da empresa que desenvolveu a classe:

```
br.com.nomedaempresa.nomedoprojeto.subpacote
br.com.nomedaempresa.nomedoprojeto.subpacote2
br.com.nomedaempresa.nomedoprojeto.subpacote2.subpacote3
```

Os pacotes só possuem letras minúsculas, não importa quantas palavras estejam contidas nele. Esse padrão existe para evitar ao máximo o conflito de pacotes de empresas diferentes. As classes do pacote padrão de bibliotecas não seguem essa nomenclatura, que foi dada para bibliotecas de terceiros.

## 12.3 - Import

Para usar uma classe do mesmo pacote, basta fazer referência a ela como foi feito até agora simplesmente escrevendo o próprio nome da classe. Se existe uma classe `Banco` dentro do pacote `br.com.caelum.banco`, ela deve ser escrita assim:

```
package br.com.caelum.banco;

class Banco {
    String nome;
    Cliente clientes[];
}
```

A classe `Cliente` deve ficar no mesmo pacote da seguinte maneira:

```
package br.com.caelum.banco;

class Cliente {
    String nome;
    String endereco;
}
```

A novidade chega ao tentar utilizar a classe `Banco` (ou `Cliente`) em uma outra classe que esteja em outro pacote, por exemplo, no pacote `br.com.caelum.util`:

```
package br.com.caelum.banco.util;

class TesteDoBanco {

    public static void main(String args[]) {
        br.com.caelum.banco.Banco meuBanco = new br.com.caelum.banco.Banco();
        meuBanco.nome = "Banco do Brasil";
        System.out.println(meuBanco.nome);
    }
}
```

Repare que precisamos referenciar a classe `Banco` com todo o nome do pacote na sua frente. Esse é o conhecido *Full Qualified Name* de uma classe. Em outras palavras, esse é o verdadeiro nome de uma classe, por isso duas classes com o mesmo nome em pacotes diferentes não conflitam.

Mesmo assim, ao tentar compilar a classe anterior, surge um erro reclamando que a classe `Banco` não está visível.

Acontece que as classes só são visíveis para outras no **mesmo pacote** e, para permitir que a classe `TesteDoBanco` veja e acesse a classe `Banco` em outro pacote, precisamos alterar essa última e transformá-la em pública:

```
package br.com.caelum.banco;

public class Banco {
    String nome;
    Cliente clientes[] = new Cliente[2];
}
```

A palavra chave `public` libera o acesso para classes de outros pacotes. Do mesmo jeito que o compilador reclamou que a classe não estava visível, agora ele reclama que o atributo/variável membro também não está. É fácil deduzir como resolver o problema - algo que já foi visto anteriormente:

```
package br.com.caelum.banco;

public class Banco {
    public String nome;
    public Cliente clientes[] = new Cliente[2];
}
```

Agora já podemos testar nosso exemplo anterior.

Voltando ao código do `TesteDoBanco`, é necessário escrever todo o pacote para identificar qual classe queremos usar? O exemplo que usamos ficou bem complicado de ler:

```
br.com.caelum.banco.Banco meuBanco = new br.com.caelum.banco.Banco();
```

Existe uma maneira mais simples de se referenciar a classe `Banco`: basta **importá-la** do pacote `br.com.caelum.banco`:

```
package br.com.caelum.banco.util;

import br.com.caelum.banco.Banco; // agora podemos nos referenciar
// a Banco diretamente

class TesteDoBanco {

    public static void main(String args[]) {
        Banco meuBanco = new Banco();
        meuBanco.nome = "Banco do Brasil";
    }
}
```

package, import, class

É muito importante manter a ordem! Primeiro, aparece uma (ou nenhuma) vez o `package`; depois, pode aparecer um ou mais `imports`; e, por último, as declarações de classes.

import x.y.z.\*;

É possível importar um pacote inteiro (todas as classes do pacote, **exceto os subpacotes**) através do `*`:

```
import java.util.*;
```

Importar todas as classes de um pacote não implica em perda de performance em tempo de execução, mas pode trazer problemas com classes de mesmo nome! Além disso, importar de um em um é considerado boa prática, pois facilita a leitura para outros programadores.

## 12.4 - Acesso aos atributos, construtores e métodos

Os modificadores de acesso existentes em Java são quatro, e até o momento já vimos três, mas só explicamos dois.

- `public` – Todos podem acessar aquilo que for definido como `public`. Classes, atributos, construtores e métodos podem ser `public`.
- `protected` – Aquilo que é `protected` pode ser acessado por todas as classes do mesmo pacote e por todas as classes que o estendam. Somente atributos, construtores e métodos podem ser `protected`.
- **padrão (sem nenhum modificador)** – Se nenhum modificador for utilizado, todas as classes do mesmo pacote têm acesso ao atributo, construtor, método ou classe.
- `private` – A única classe capaz de acessar os atributos, construtores e métodos privados é a própria classe. Classes, como conhecemos, não podem ser `private`, mas atributos, construtores e métodos sim.

### Classes públicas

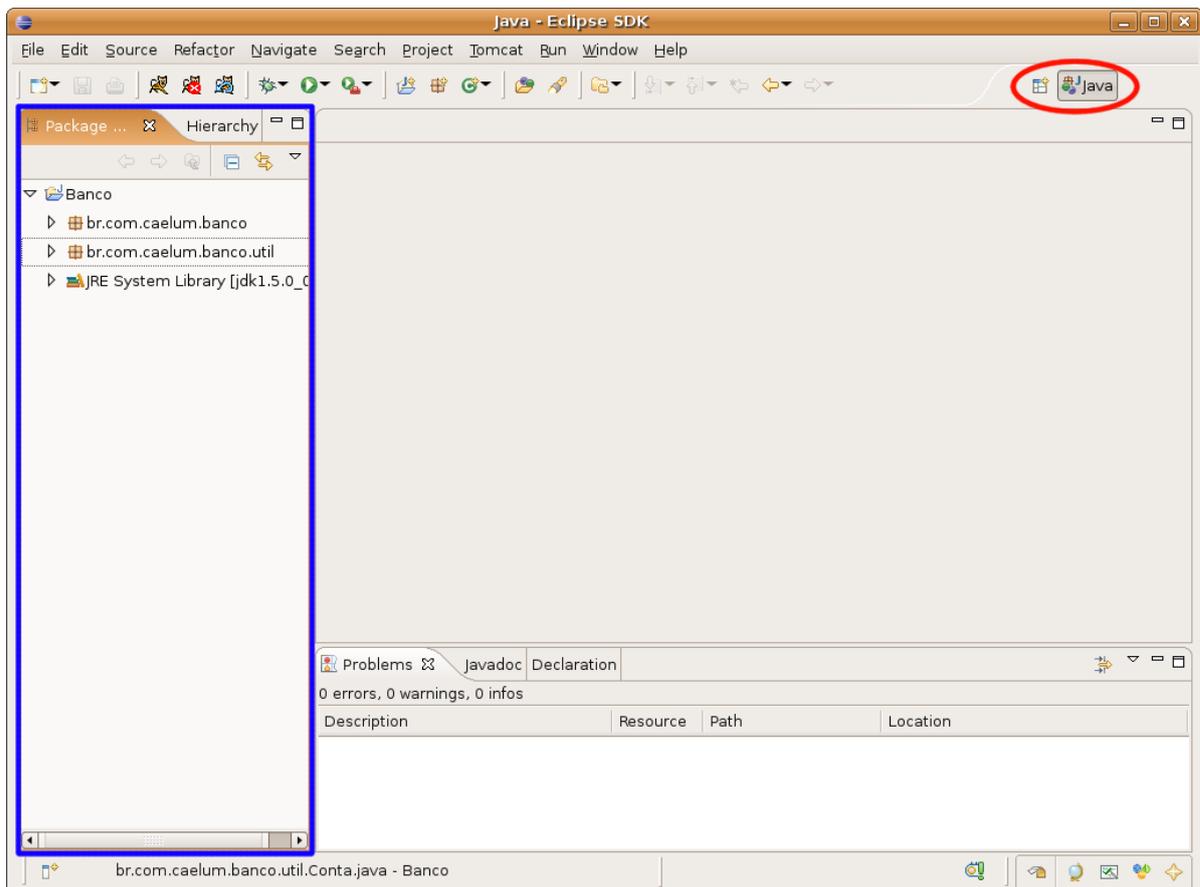
Para melhor organizar seu código, o Java não permite mais de uma classe pública por arquivo e o arquivo deve ser `NomeDaClasse.java`.

Uma vez que outros programadores irão utilizar essa classe, quando precisarem olhar o código da mesma, fica mais fácil encontrá-la sabendo que ela está no arquivo de mesmo nome.

Classes podem ser `protected` ou `private`, mas esse é um tópico avançado que não será estudado nesse instante (classes internas).

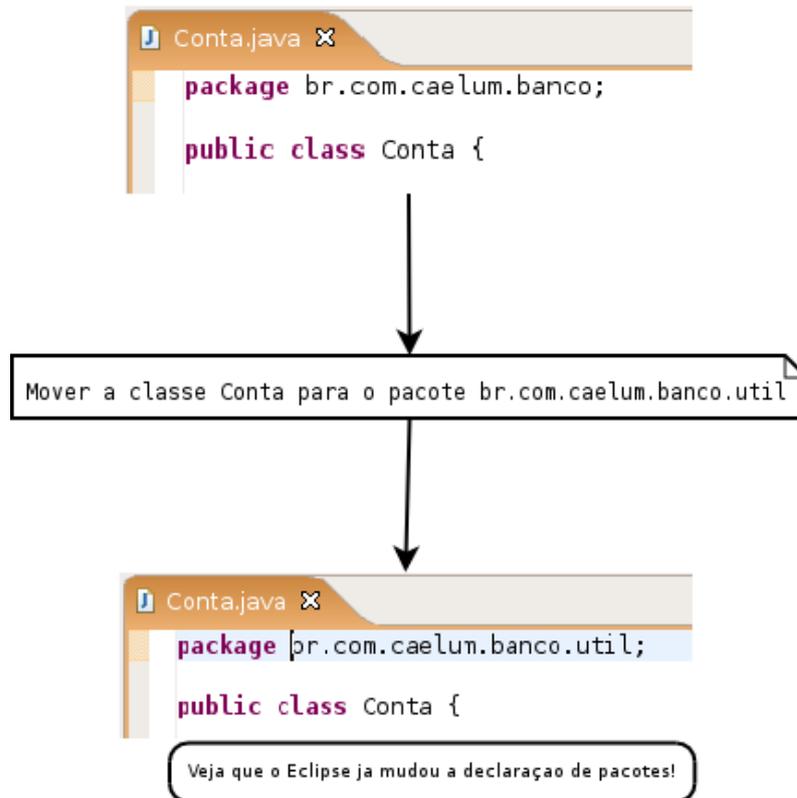
## 12.5 - Usando o Eclipse com pacotes

Você pode, agora, usar a perspectiva Java do eclipse. A view principal de navegação é o package explorer, que agrupa classes pelos pacotes em vez de diretórios (você pode usá-la em conjunto com a *Navigator*, basta também abri-la pelo *Window/Show View/Package Explorer*)



Antes de movermos nossas classes, declare-as como públicas e coloque-as em seus respectivos arquivos: um arquivo para cada classe.

Você pode mover uma classe de pacote arrastando-a para o destino desejado. Repare que o Eclipse já declara `packages` e `imports` necessários:



No Eclipse nunca precisamos declarar um `import`, pois ele sempre vai sugerir isso quando usarmos o `Ctrl+Espaço` no nome de uma classe.

Você também pode usar o `Ctrl+1` no caso da declaração de pacote possuir algum erro.

## 12.6 - Exercícios: Pacotes

**Atenção:** utilize os recursos do Eclipse para realizar essas mudanças. Use a view `package-explorer`. Também utilize os quick fixes quando o Eclipse reclamar dos diversos problemas de compilação.

- 1) Ponha o seu sistema de Contas para utilizar pacotes. Respeite a convenção de código da Sun, por exemplo:  
`br.com.empresa.banco`: colocar classes com o método `main` aqui  
`br.com.empresa.banco.conta`: colocar `Conta`, suas filhas e `exceptions` aqui  
`br.com.empresa.banco.sistema`: colocar `AtualizadorDeContas` aqui
- 2) Coloque cada classe em seu respectivo arquivo `.java`. Independente de ela ser pública (boa prática).
- 3) O código não vai compilar prontamente, pois muitos métodos que declaramos são `package-friendly` quando, na verdade, precisaríamos que eles fossem `public`.

O mesmo vale para as classes: algumas delas precisarão ser públicas. Use o recurso de quick fix do Eclipse aqui - ele mesmo vai sugerir que o modificador de acesso deve ser público.

# Ferramentas: jar e javadoc

*“Perder tempo em aprender coisas que não interessam, priva-nos de descobrir coisa interessantes”*  
– Carlos Drummond de Andrade

Ao término desse capítulo, você será capaz de:

- criar o JAR do seu aplicativo;
- colocar um JAR no build path do seu projeto;
- ler um javadoc;
- criar o javadoc do seu aplicativo.

## 13.1 - Arquivos, bibliotecas e versões

Assim que um programa fica pronto, é meio complicado enviar dezenas ou centenas de classes para cada cliente que quer utilizá-lo.

O jeito mais simples de trabalhar com um conjunto de classes é compactá-los em um arquivo só. O formato de compactação padrão é o ZIP com a extensão do arquivo compactado JAR.

### O arquivo .jar

O arquivo **jar** ou **Java ARchive**, possui um conjunto de classes (e arquivos de configurações) compactados, no estilo de um arquivo zip. O arquivo jar pode ser criado com qualquer compactador zip disponível no mercado, inclusive o programa jar que vem junto com o sdk.

Para criar um arquivo jar do nosso programa de banco, basta ir ao diretório onde estão contidas as classes e usar o comando a seguir para criar o arquivo `banco.jar` com todas as classes dos pacotes `br.com.caelum.util` e `br.com.caelum.banco`:

```
jar -cvf banco.jar br/com/caelum/util/*.class br/com/caelum/banco/*.class
```

Para usar esse arquivo `banco.jar` para rodar o `TesteDoBanco` basta rodar o java com o arquivo jar como argumento:

```
java -classpath banco.jar br.com.caelum.util.TesteDoBanco
```

Para adicionar mais arquivos .jar, que podem ser bibliotecas, ao programa basta rodar o java da seguinte maneira:

```
java -classpath biblioteca1.jar;biblioteca2.jar NomeDaClasse
```

Vale lembrar que o ponto e vírgula utilizado só é válido em ambiente windows. Em linux, é o dois pontos - varia de acordo com o sistema operacional.

Há também um arquivo de manifesto que contém informações do seu jar como, por exemplo, qual classe ele vai rodar quando o jar for chamado. Mas não se preocupe pois, com o eclipse, esse arquivo é gerado automaticamente.

#### Bibliotecas

Diversas bibliotecas podem ser controladas de acordo com a versão por estarem sempre compactadas em um arquivo `.jar`. Basta verificar o nome da biblioteca (por exemplo `log4j-1.2.13.jar`) para descobrir a versão dela.

Então é possível rodar dois programas ao mesmo tempo, cada um utilizando uma versão da biblioteca através do parâmetro `-classpath` do java.

#### Criando um `.jar` automaticamente

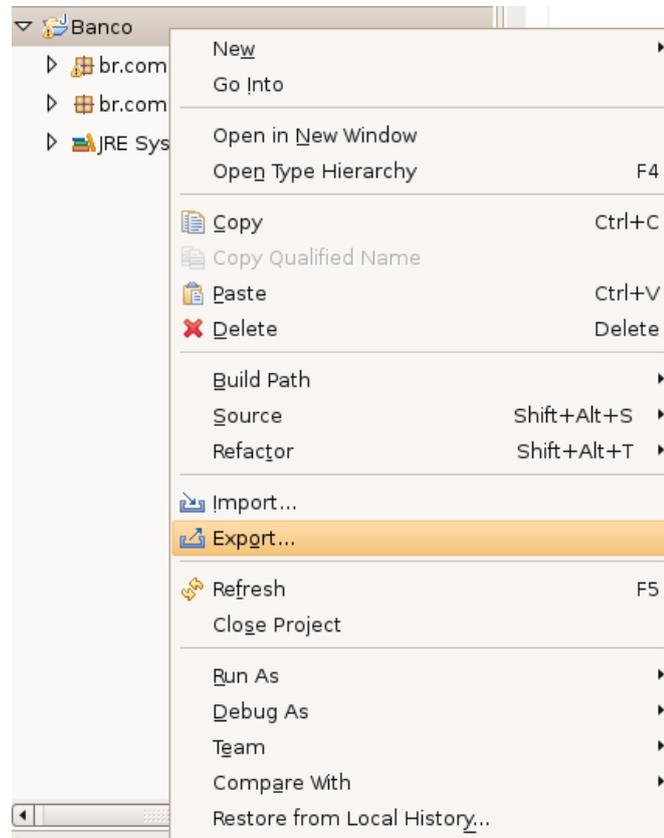
Existem diversas ferramentas que servem para automatizar o processo de deploy, que consiste em compilar, gerar documentação, bibliotecas etc. As duas mais famosas são o **ANT** e o **MAVEN**, ambos são projetos do grupo Apache.

O Eclipse pode gerar facilmente um jar, porém, se o seu build é complexo e precisa preparar e copiar uma série de recursos, as ferramentas indicadas acima possuem sofisticadas maneiras de rodar um script batch.

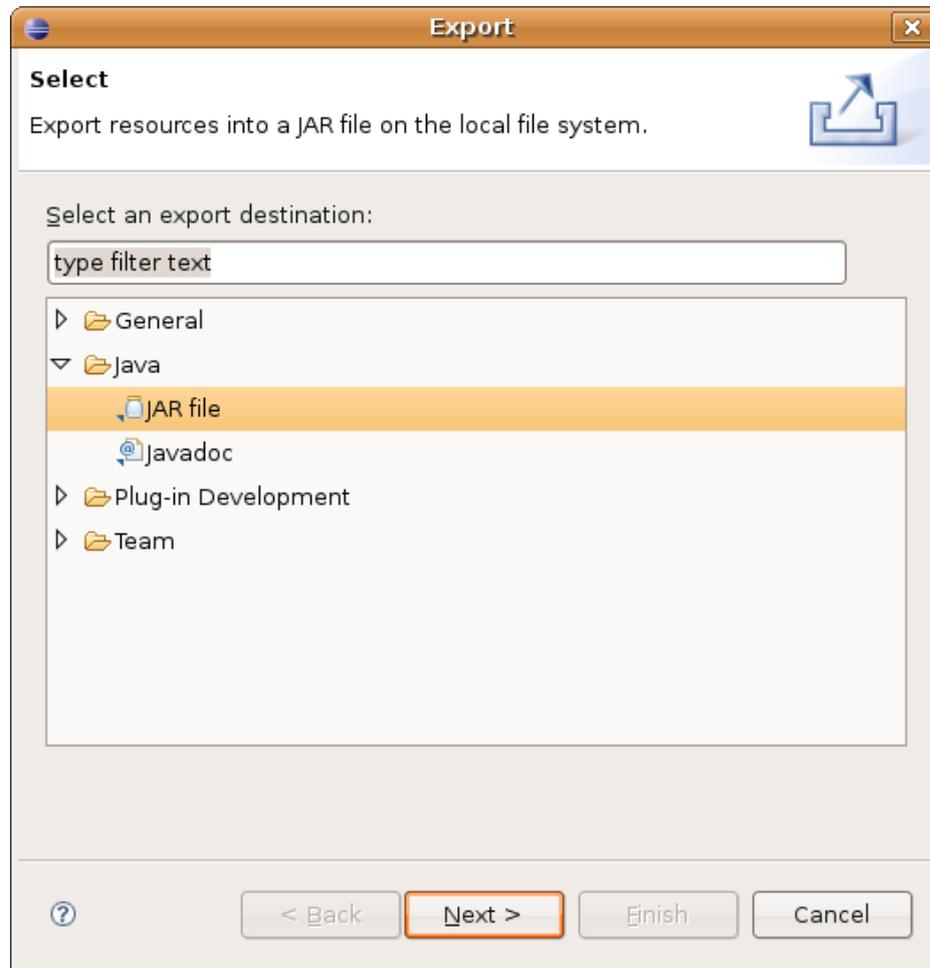
## 13.2 - Gerando o jar pelo Eclipse

Neste exemplo, vamos gerar o arquivo JAR do nosso projeto a partir do eclipse.

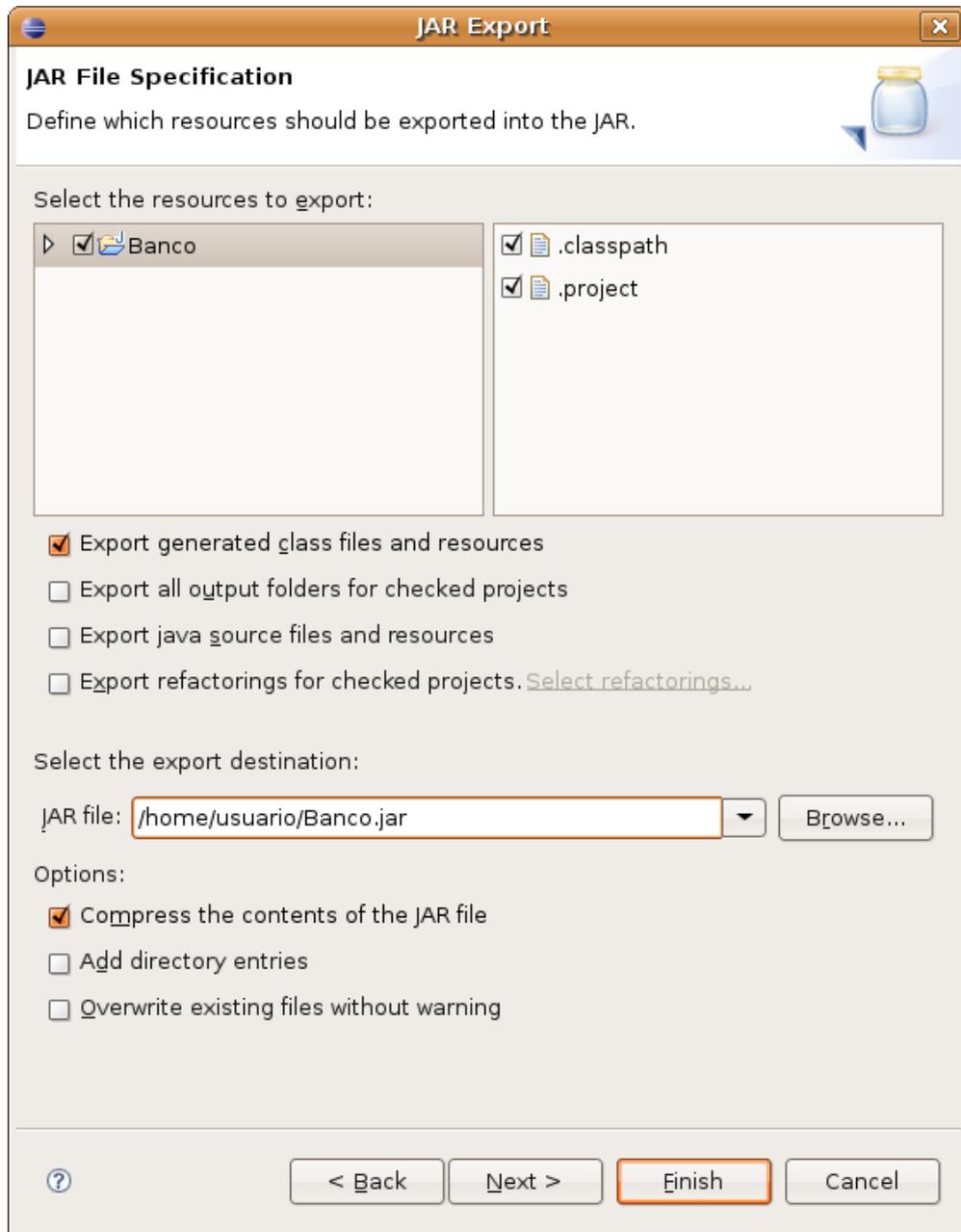
- 1) Clique com o botão direito em cima do nome do seu projeto e selecione a opção Export.



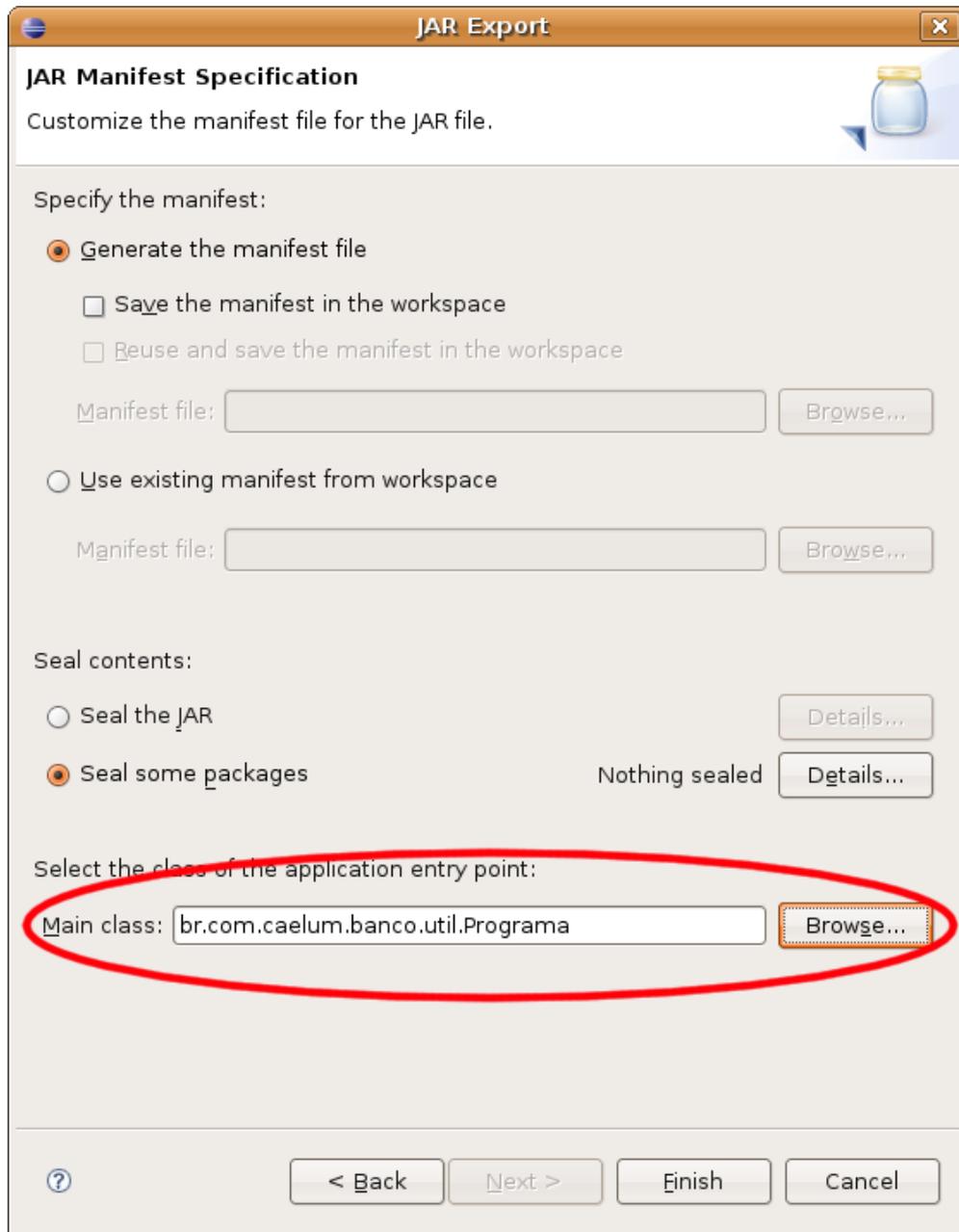
2) Na tela Export (como mostra a figura abaixo), selecione a opção “JAR file” e aperte o botão “Next”.



3) Na opção “JAR file:”, selecione o local que você deseja salvar o arquivo JAR. E aperte “Next”.



- 4) Na próxima tela, simplesmente clique em next, pois não há nenhuma configuração a ser feita.
- 5) Na tela abaixo, na opção “select the class of the application entry point”, você deve escolher qual classe será a classe que vai rodar automaticamente quando você executar o JAR.



6) Entre na linha de comando: `java -jar banco.jar`

#### Usando jars e classpath

No capítulo 19 vamos ver como usar jars de terceiros na prática e aprender o que é e como configurar o classpath.

### 13.3 - Javadoc

Como vamos saber o que cada classe tem no Java? Quais são seus métodos, o que eles fazem?

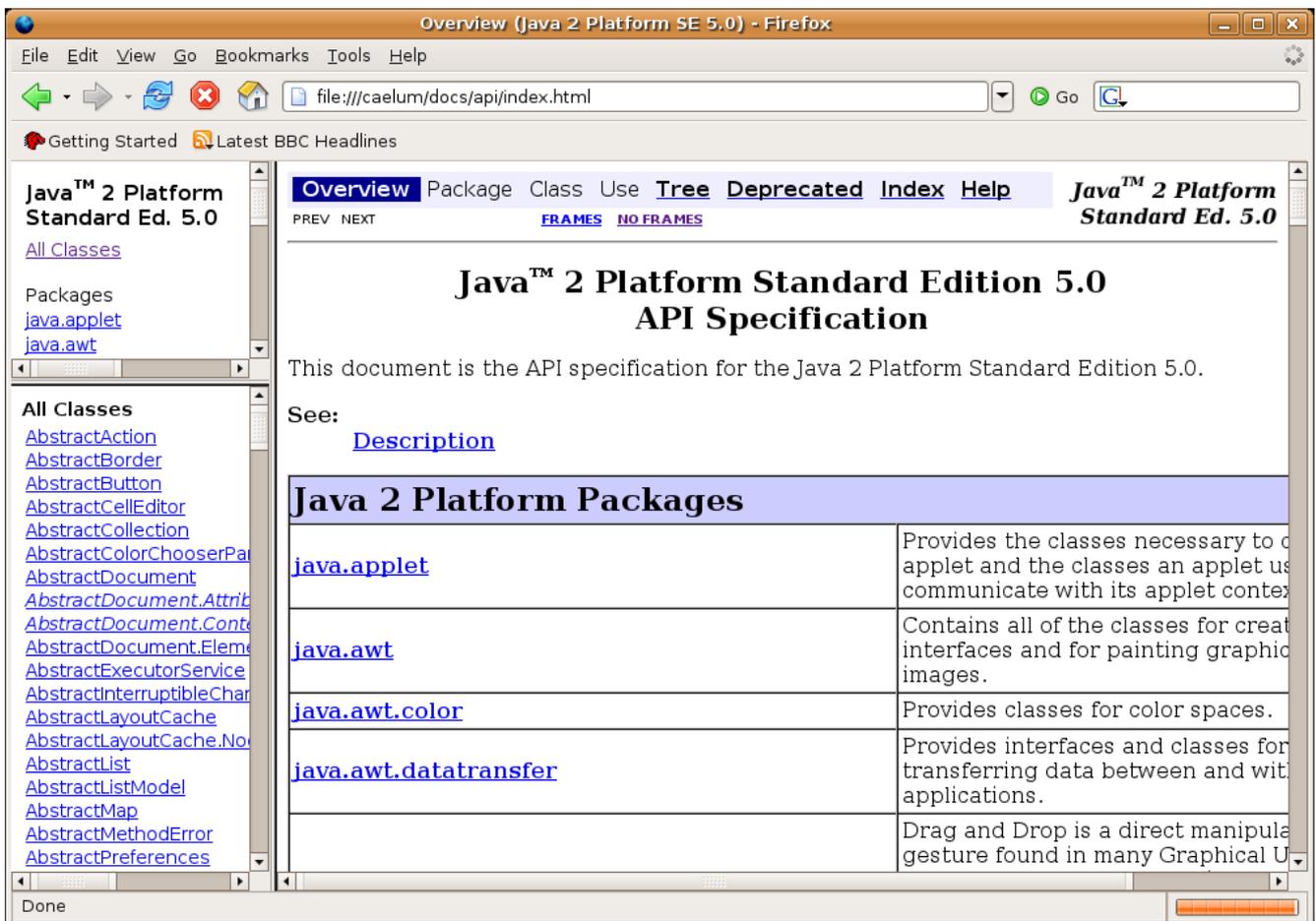
Aqui na Caelum, você pode acessar a documentação digitando na barra de endereço do Browser:

*/caelum/docs/api/index.html*

E, a partir da Internet, você pode acessar através do link:

<http://java.sun.com/javase/6/docs/api/>

No site da Sun, você pode (e deve) baixar a documentação das bibliotecas do Java, freqüentemente referida como **“javadoc”** ou API (sendo na verdade a documentação da API).



Nesta documentação, no quadro superior esquerdo, você encontra os pacotes e, no inferior esquerdo, está a listagem das classes e interfaces do respectivo pacote (ou de todos, caso nenhum tenha sido especificado). Clicando-se em uma classe ou interface, o quadro da direita passa a detalhar todos atributos e métodos.

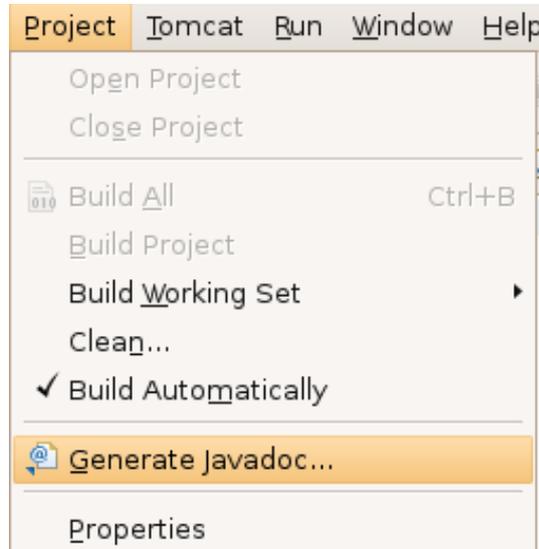
Repare que métodos e atributos privados não estão aí. O importante é documentar o que sua classe faz, e não como ela faz: detalhes de implementação, como atributos e métodos privados, não interessam ao desenvolvedor que usará a sua biblioteca (ou, ao menos, não deveriam interessar).

Você também consegue gerar esse javadoc a partir da linha de comando, com o comando: `javadoc`.

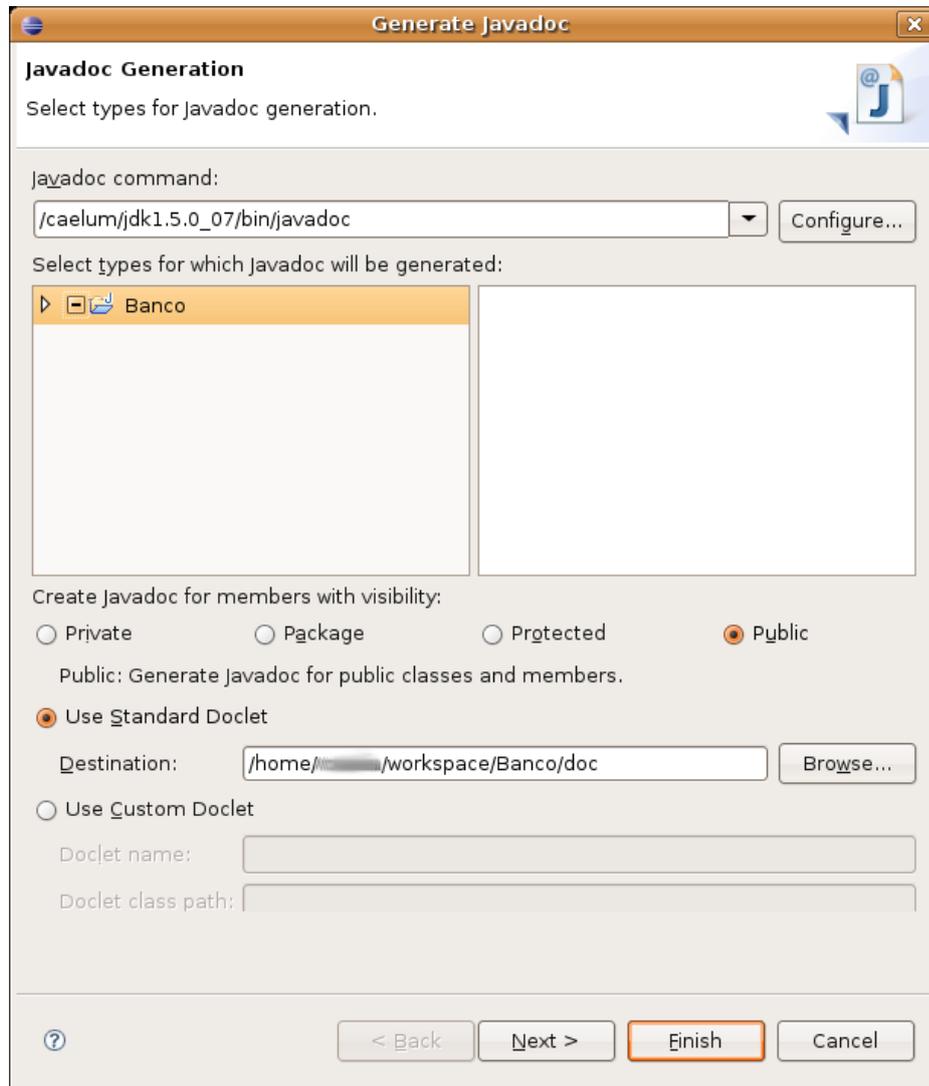
## **13.4 - Gerando o Javadoc**

Para gerar o Javadoc a partir do Eclipse é muito simples, siga os passos abaixo:

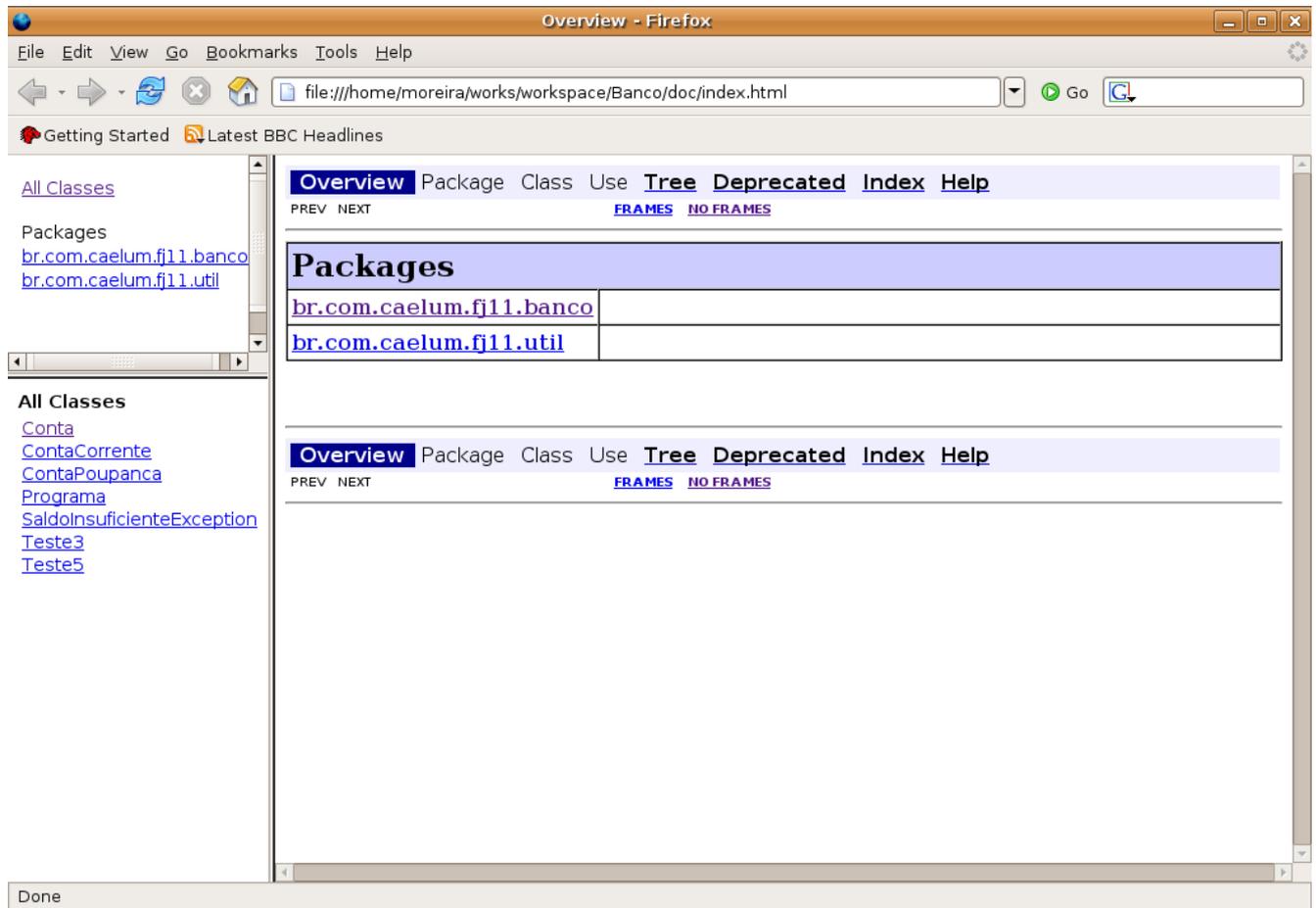
- 1) Na barra de menu, selecione o menu Project, depois a opção “Generate Javadoc...”. (apenas disponível se estiver na perspectiva Java, mas você pode acessar o mesmo wizard pelo export do projeto).



- 2) Em seguida, aparecerão as opções para gerar a documentação do seu sistema, selecione todas as classes do seu sistema e deixe as outras opções como estão. Não esqueça de marcar o caminho da opção “Destination”, pois é lá que estará sua documentação.



- 3) Abra a documentação através do caminho que você marcou e abra o arquivo index.html, que vai chamar uma página semelhante a essa da figura abaixo.



Para colocarmos comentários na documentação, devemos adicionar ao código, sob forma de comentário, abrindo o texto com `/**` e fechando com `*/` e, nas outras linhas, apenas colocando `*`. Também podemos definir algumas coisas neste texto, como: autor, versão, parâmetros, retorno, etc. Agora, adicione alguns comentários ao seu Projeto como abaixo:

```
/**
 * Classe responsavel por moldar as Contas do Banco
 *
 * @author Manoel Santos da Silva
 */

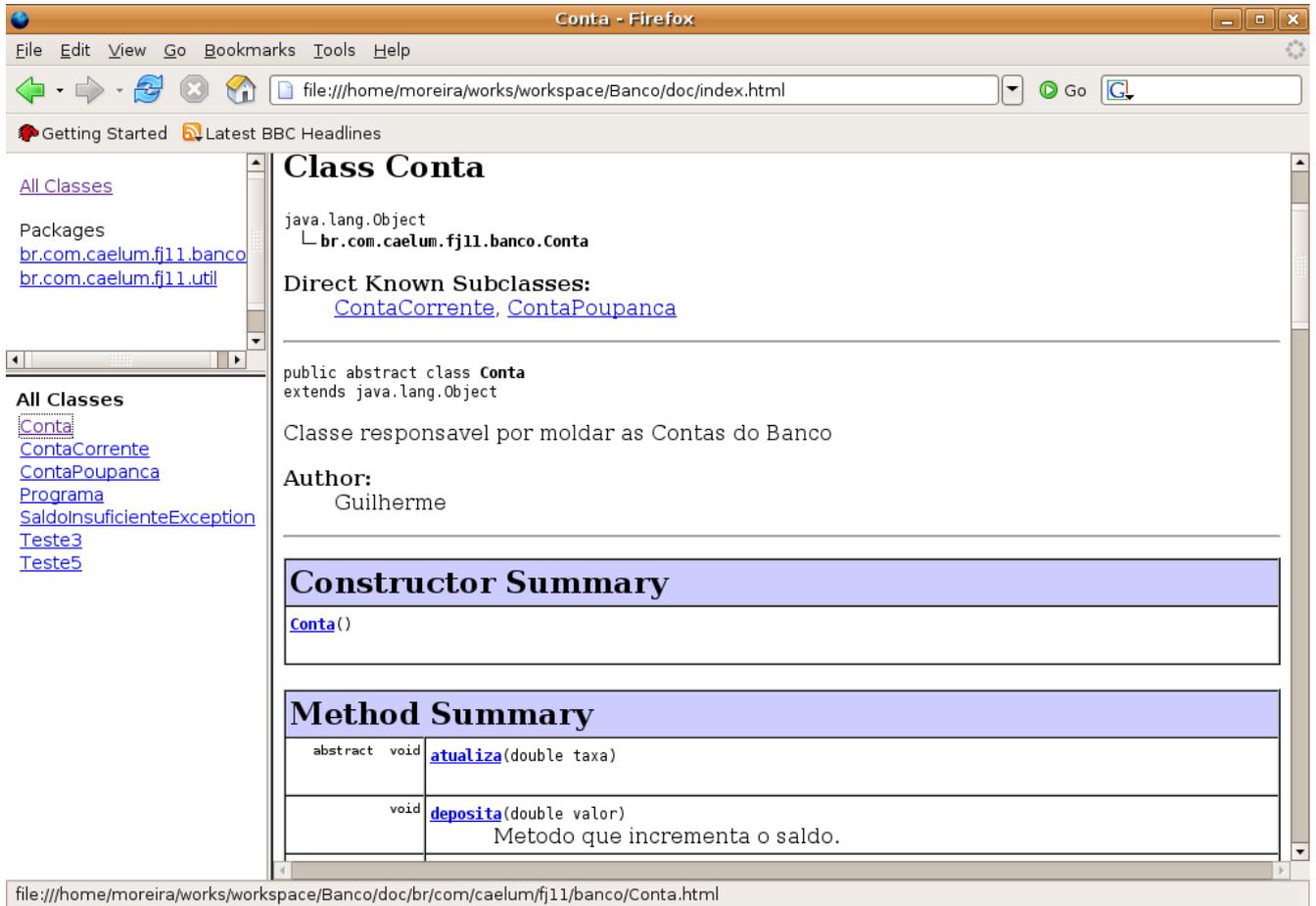
public abstract class Conta{
```

Ou adicione alguns comentários em algum método seu:

```
/**
 * Metodo que incrementa o saldo.
 * @param valor
 */

public void deposita(double valor) {
```

Veja como ficou:



**Class Conta**

java.lang.Object  
└─ br.com.caelum.fj11.banco.Conta

**Direct Known Subclasses:**  
[ContaCorrente](#), [ContaPoupanca](#)

---

public abstract class **Conta**  
extends java.lang.Object

Classe responsavel por moldar as Contas do Banco

**Author:**  
Guilherme

---

**Constructor Summary**

<a href="#">Conta</a> ()
--------------------------

---

**Method Summary**

abstract void	<a href="#">atualiza</a> (double taxa)
void	<a href="#">deposita</a> (double valor) Metodo que incrementa o saldo.

file:///home/moreira/works/workspace/Banco/doc/index.html

file:///home/moreira/works/workspace/Banco/doc/br.com/caelum/fj11/banco/Conta.html

## 13.5 - Exercícios: Jar e Javadoc

- 1) Gere um jar do seu sistema com o arquivo de manifesto. Execute-o com `java -jar`:

```
java -jar Projeto.jar
```

Se o windows ou linux foi configurado para trabalhar com a extensão .jar, basta você dar um duplo clique no arquivo, que ele será "executado" (o arquivo de manifesto será lido para que ele possa descobrir qual é a classe com main que o java deve processar).

- 2) Gere o javadoc do seu sistema.

# O pacote `java.lang`

*“Nossas cabeças são redondas para que os pensamentos possam mudar de direção.”*

– Francis Picaba

Ao término desse capítulo, você será capaz de:

- utilizar as principais classes do pacote `java.lang` e ler a documentação padrão de projetos java;
- usar a classe `System` para obter informações do sistema;
- utilizar a classe `String` de uma maneira eficiente e conhecer seus detalhes;
- usar as classes wrappers (como `Integer`) e boxing;
- utilizar os métodos herdados de `Object` para generalizar seu conceito de objetos.

## 14.1 - Pacote `java.lang`

Já usamos, por diversas vezes, as classes `String` e `System`. Vimos o sistema de pacotes do java e nunca precisamos dar um `import` nessas classes. Isso ocorre porque elas estão dentro do pacote `java.lang`, que é **automaticamente importado** para você. É o **único pacote** com esta característica.

Vamos ver um pouco de suas principais classes.

## 14.2 - Um pouco sobre a classe `System`

A classe `System` possui uma série de atributos e métodos estáticos. Já usamos o atributo `System.out`, para imprimir. Ela também possui o atributo `in`, que lê da entrada padrão alguns bytes.

```
int i = System.in.read();
```

O código acima deve estar dentro de um bloco de `try` e `catch`, pois pode lançar uma exceção `IOException`. Será útil ficar lendo de `byte` em `byte`?

O `System` conta também com um método que simplesmente desliga a virtual machine, retornando um código de erro para o sistema operacional, é o `exit`.

```
System.exit(0);
```

Veremos também um pouco mais sobre a classe `System` no próximo capítulo e no de `Threads`. Consulte a documentação do Java e veja outros métodos úteis da `System`.

### A classe Runtime

A classe `Runtime` possui um método para fazer uma chamada ao sistema operacional e rodar algum programa:

```
Runtime rt = Runtime.getRuntime();  
Process p = rt.exec("dir");
```

Isto deve ser evitado ao máximo, já que gera uma dependência da sua aplicação com o sistema operacional em questão, perdendo a portabilidade entre plataformas. Em muitos casos, isso pode ser substituído por chamadas as bibliotecas do java. Nesse caso, por exemplo, você tem um método `list` na classe `File` do pacote de entrada e saída, que veremos posteriormente.

O método `exec` te retorna um `Process`, onde você é capaz de pegar a saída do programa, enviar dados para a entrada, entre outros.

## 14.3 - java.lang.Object

Sempre quando declaramos uma classe, essa classe é **obrigada** a herdar de outra. Isto é, para toda classe que declaramos, existe uma superclasse. Porém, criamos diversas classes sem herdar de ninguém:

```
class MinhaClasse {  
  
}
```

Quando o Java não encontra a palavra chave `extends`, ele considera que você está herdando da classe `Object`, que também se encontra dentro do pacote `java.lang`. Você até mesmo pode escrever essa herança, que é a mesma coisa:

```
class MinhaClasse extends Object {  
  
}
```

**Todas as classes, sem exceção, herdam de `Object`**, seja direta ou indiretamente.

Podemos também afirmar que qualquer objeto em Java é um `Object`, podendo ser referenciado como tal. Então, qualquer objeto possui todos os métodos declarados na classe `Object` e veremos alguns deles logo após o casting.

## 14.4 - Casting de referências

A habilidade de poder se referir a qualquer objeto como `Object` nos traz muitas vantagens. Podemos criar um método que recebe um `Object` como argumento, isto é, qualquer coisa! Melhor, podemos armazenar qualquer objeto:

```
public class GuardadorDeObjetos {  
    private Object[] arrayDeObjetos = new Object[100];  
    private int posicao= 0;  
  
    public void adicionaObjeto(Object object) {  
        this.arrayDeObjetos[this.posicao] = object;  
        this.posicao++;  
    }  
}
```

```
public Object pegaObjeto(int indice) {  
    return this.arrayDeObjetos[indice];  
}  
}
```

Mas e no momento que retirarmos uma referência a esse objeto, como vamos acessar os métodos e atributos desse objeto? Se estamos referenciando-o como `Object`, não podemos acessá-lo como sendo `Conta`. Veja o exemplo a seguir:

```
GuardadorDeObjetos guardador = new GuardadorDeObjetos();  
Conta conta = new Conta();  
guardador.adicionaObjeto(conta);  
  
// ...  
  
Object object = guardador.pegaObjeto(0); // pega a conta referenciado como objeto  
object.getSaldo(); // classe Object nao tem método getSaldo! não compila!
```

Poderíamos então atribuir essa referência de `Object` para `Conta`? Tentemos:

```
Conta contaResgatada = object;
```

Nós temos certeza de que esse `Object` se refere a uma `Conta`, já que fomos nós que o adicionamos na classe que guarda objetos. Mas o Java não tem garantias sobre isso! Essa linha acima não compila, pois nem todo `Object` é uma `Conta`.

Para conseguir realizar essa atribuição, devemos “avisar” o Java que realmente queremos fazer isso, sabendo do risco que podemos estar correndo. Fazemos o **casting de referências**, parecido com quando o fizemos com os tipos primitivos:

```
Conta contaResgatada = (Conta) object;
```

Agora o código passa a compilar, mas será que roda? Esse código roda sem nenhum problema, pois em tempo de execução ele verificará se essa referência realmente está se referindo a uma `Conta`, e está! Se não estivesse, uma exceção do tipo `ClassCastException` seria lançada.

Poderíamos fazer o mesmo com `Funcionario` e `Gerente`. Tendo uma referência para um `Funcionario` que temos certeza ser um `Gerente`, podemos fazer a atribuição, desde que o casting exista, pois nem todo `Funcionario` é um `Gerente`.

```
Funcionario funcionario = new Gerente();  
  
// ... e depois  
  
Gerente gerente = funcionario; // não compila!  
// nem todo Funcionario é um Gerente
```

O correto então seria:

```
Gerente gerente = (Gerente) funcionario;
```

E agora vamos misturar algumas coisas:

```
Object object = new Conta();  
  
// ... e depois  
  
Gerente gerente = (Gerente) object;
```

Esse código compila? Roda?

Compila, porém não roda, ele vai lançar uma `Exception` (`ClassCastException`) em tempo de execução. É importante diferenciar tempo de compilação e tempo de execução.

Neste exemplo, nós garantimos ao java que nosso Objeto `object` era um `Gerente` com o Casting, por isso compilou, mas na hora de rodar, quando ele foi receber um `Gerente`, ele recebeu uma `Conta`, daí ele reclamou lançando `ClassCastException`!

## 14.5 - Métodos do `java.lang.Object`: `equals` e `toString`

O primeiro método interessante é o `toString`. As classes podem reescrever esse método para mostrar uma mensagem, uma `String`, que o represente. Você pode usá-lo assim:

```
Conta c = new Conta();  
System.out.println(c.toString());
```

O método `toString` do `Object` retorna o nome da classe @ um número de identidade:

```
Conta@34f5d74a
```

Mas isso não é interessante para nós. Então podemos reescrevê-lo:

```
class Conta {  
    private double saldo;  
    // outros atributos...  
  
    public Conta(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public String toString() {  
        return "Uma conta com valor: " + this.saldo;  
    }  
}
```

Chamando o `toString` agora:

```
Conta c = new Conta(100);  
System.out.println(c.toString()); //imprime: Uma conta com valor: 100.
```

E o melhor, se for apenas para jogar na tela, você nem precisa chamar o `toString`! Ele já é chamado para você:

```
Conta c = new Conta(100);  
System.out.println(c); // O toString é chamado pela classe PrintStream
```

Gera o mesmo resultado!

Você ainda pode concatenar `Strings` em Java com o operador `+`. Se o Java encontra um objeto no meio da concatenação, ele também chama o `toString` dele.

```
Conta c = new Conta(100);
System.out.println("descrição: " + c);
```

O outro método muito importante é o `equals`. Quando comparamos duas variáveis referência no Java, o `==` verifica se as duas referem-se ao mesmo objeto:

```
Conta c1 = new Conta(100);
Conta c2 = new Conta(100);
if (c1 != c2) {
    System.out.println("objetos referenciados são diferentes!");
}
```

Mas, e se fosse preciso comparar os atributos? Quais atributos ele deveria comparar? O Java por si só não faz isso, mas existe um método na classe `Object` que pode ser reescrito para criarmos esse critério de comparação. Esse método é o `equals`.

O `equals` recebe um `Object` como argumento e deve verificar se ele mesmo é igual ao `Object` recebido para retornar um `boolean`. Se você não reescrever esse método, o comportamento herdado é fazer um `==` com o objeto recebido como argumento.

```
public class Conta {
    private double saldo;
    // outros atributos...

    public Conta(double saldo) {
        this.saldo = saldo;
    }

    public boolean equals(Object object) {
        Conta outraConta = (Conta) object;
        if (this.saldo == outraConta.saldo) {
            return true;
        }
        return false;
    }

    public String toString() {
        return "Uma conta com valor: " + this.saldo;
    }
}
```

Um exemplo clássico do uso do `equals` é para datas. Se você criar duas datas, isto é, dois objetos diferentes, contendo 31/10/1979, ao comparar com o `==` receberá `false`, pois são referências para objetos diferentes. Seria correto, então, reescrever este método, fazendo as comparações dos atributos, e o usuário passaria a invocar `equals` em vez de comparar com `==`.

Você poderia criar um método com outro nome em vez de reescrever `equals` que recebe `Object`, mas ele é importante pois muitas bibliotecas o chamam através do polimorfismo, como veremos no capítulo do `java.util`.

O método `hashCode()` anda de mãos dadas com o método `equals()` e é de fundamental entendimento no caso de você utilizar suas classes com estruturas de dados que usam tabelas de espalhamento. Também falaremos dele no capítulo de `java.util`.

## 14.6 - Integer e classes wrappers (box)

Uma pergunta bem simples que surge na cabeça de todo programador ao aprender uma nova linguagem é: “Como transformar um número em `String` e vice-versa?”.

Cuidado! Usamos aqui o termo “transformar” porém o que ocorre não é uma transformação entre os tipos, e sim uma forma de conseguirmos uma `String` dado um `int` e vice-versa. O jeito mais simples de transformar um número em `String` é concatená-lo da seguinte maneira:

```
int i = 100;
String s = "" + i;
System.out.println(s);
```

```
double d = 1.2;
String s2 = "" + d;
System.out.println(s2);
```

Para formatar o número de uma maneira diferente, com vírgula e número de casas decimais devemos utilizar outras classes de ajuda (`NumberFormat`, `Formatter`).

Para transformar uma `String` em número, utilizamos as classes de ajuda para os tipos primitivos correspondentes. Por exemplo, para transformar a `String` `s` em um número inteiro utilizamos o método estático da classe `Integer`:

```
String s = "101";
int i = Integer.parseInt(s);
```

As classes `Double`, `Short`, `Long`, `Float` etc contêm o mesmo tipo de método, como `parseDouble` e `parseFloat` que retornam um `double` e `float` respectivamente.

Essas classes também são muito utilizadas para fazer o **wrapping** (embrulho) de tipos primitivos como objetos, pois referências e tipos primitivos são incompatíveis. Imagine que precisamos passar como argumento um inteiro para o nosso guardador de carros. Um inteiro não é um `Object`, como fazer?

```
int i = 5;
Integer x = new Integer(i);
```

E, dado um `Integer`, podemos pegar o `int` que está dentro dele (desembrulhá-lo):

```
int i = 5;
Integer x = new Integer(i);
int numeroDeVolta = x.intValue();
```

## 14.7 - Autoboxing no Java 5.0

Esse processo de wrapping e unwrapping é entediante. O Java 5.0 em diante traz um recurso chamado de **autoboxing**, que faz isso sozinho para você, custando legibilidade:

```
Integer x = 5;  
int y = x;
```

No Java 1.4 esse código é inválido. No Java 5.0 em diante ele compila perfeitamente. É importante ressaltar que isso não quer dizer que tipos primitivos e referências agora são a mesma coisa, isso é simplesmente um “açúcar sintático” (*syntax sugar*) para facilitar a codificação.

Você pode fazer todos os tipos de operações matemáticas com os wrappers agora, porém corre o risco de tomar um `NullPointerException`.

Você pode fazer o autoboxing diretamente para `Object` também, possibilitando passar um tipo primitivo para um método que receber `Object` como argumento:

```
Object o = 5;
```

## 14.8 - java.lang.String

`String` é uma classe em Java. Variáveis do tipo `String` guardam referências a objetos, e não um valor, como acontece com os tipos primitivos.

Aliás, podemos criar uma `String` utilizando-nos do `new`:

```
String x = new String("fj11");  
String y = new String("fj11");
```

Criamos aqui, dois objetos diferentes. O que acontece quando comparamos essas duas referências utilizando o `==`?

```
if (x == y) {  
    System.out.println("mesmo objeto");  
}  
else {  
    System.out.println("objetos diferentes!");  
}
```

Temos aqui dois objetos diferentes! E, então, como faríamos para verificar se o conteúdo do objeto é o mesmo? Utilizamos o método `equals`, que foi reescrito pela `String`, para fazer a comparação de char em char.

```
if (x.equals(y)) {  
    System.out.println("consideramos iguais no critério de igualdade");  
}  
else {  
    System.out.println("consideramos diferentes no critério de igualdade");  
}
```

Aqui, a comparação retorna verdadeiro. Por quê? Pois quem implementou a classe `String` decidiu que este seria o melhor critério de comparação. Você pode descobrir os critérios de igualdade de cada classe pela documentação.

Podemos também concatenar `Strings` usando o `+`. Podemos concatenar `Strings` com qualquer outra coisa, até mesmo objetos e números:

```
int total = 5;
System.out.println("o total gasto é: " + total);
```

A classe `String` conta também com um método `split`, que divide a `String` em um array de `Strings`, dado determinado critério.

```
String frase = "java é demais";
String palavras[] = frase.split(" ");
```

Se quisermos comparar duas `Strings`, utilizamos o método `compareTo`, que recebe uma `String` como argumento e devolve um inteiro indicando se a `String` vem antes, é igual ou vem depois da `String` recebida. Se forem iguais, é devolvido 0; se for anterior à `String` do argumento, devolve um inteiro negativo; e, se for posterior, um inteiro positivo.

Fato importante: uma `String` é imutável. O java cria um pool de `Strings` para usar como cache e, se a `String` não fosse imutável, mudando o valor de uma `String` afetaria todas as `Strings` de outras classes que tivessem o mesmo valor.

Repare no código abaixo:

```
String palavra = "fj11";
palavra.toUpperCase();
System.out.println(palavra);
```

Pode parecer estranho, mas ele imprime “fj11” em minúsculo. Todo método que parece alterar o valor de uma `String`, na verdade, cria uma nova `String` com as mudanças solicitadas e a retorna. O código realmente útil ficaria assim:

```
String palavra = "fj11";
String outra = palavra.toUpperCase();
System.out.println(outra);
```

Ou você pode eliminar a criação de outra variável temporária, se achar conveniente:

```
String palavra = "fj11";
palavra = palavra.toUpperCase();
System.out.println(palavra);
```

Isso funciona da mesma forma para **todos** os métodos que parecem alterar o conteúdo de uma `String`.

O funcionamento do pool interno de `Strings` do java tem uma série de detalhes e você pode encontrar mais informações sobre isto na documentação da classe `String` e no seu método `intern()`.

#### Outros métodos da classe `String`

Existem diversos métodos da classe `String` que são extremamente importantes. Recomendamos sempre consultar o javadoc relativo a essa classe para aprender cada vez mais sobre a mesma. Por exemplo, o método `charAt(i)`, retorna o caractere existente na posição `i` da string, o **método** `length` retorna o número de caracteres na mesma e o método `substring` que recebe um `int` e devolve a `SubString` a partir da posição passada por aquele `int`.

O `indexOf` recebe um `char` ou uma `String` e devolve o índice em que aparece pela primeira vez na `String` principal (há também o `lastIndexOf` que devolve o índice da última ocorrência).

O `toUpperCase` e o `toLowerCase` devolvem uma nova `String` toda em maiúscula e toda em minúscula, respectivamente.

A partir do Java 6, temos ainda o método `isEmpty`, que devolve `true` se a `String` for vazia ou `false` caso contrário.

Alguns métodos úteis para buscas são o `contains` e o `matches`.

Há muitos outros métodos, recomendamos que você sempre consulte o javadoc da classe.

#### `java.lang.StringBuffer` e `StringBuilder`

Como a classe `String` é imutável, trabalhar com uma mesma `String` diversas vezes pode ter um efeito colateral: gerar inúmeras `Strings` temporárias. Isto prejudica a performance da aplicação consideravelmente.

No caso de você trabalhar muito com a manipulação de uma mesma `String` (por exemplo, dentro de um laço), o ideal é utilizar a classe `StringBuffer`. A classe `StringBuffer` representa uma sequência de caracteres. Diferentemente da `String`, ela é mutável, e não possui aquele pool.

A classe `StringBuilder` tem exatamente os mesmos métodos, com a diferença dela não ser **thread-safe**. Veremos sobre este conceito no capítulo de `Threads`.

## 14.9 - `java.lang.Math`

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

```
double d = 4.6;  
long i = Math.round(d);
```

```
int x = -4;  
int y = Math.abs(x);
```

Consulte a documentação para ver a grande quantidade de métodos diferentes.

No Java 5.0, podemos tirar proveito do `import static` aqui:

```
import static java.lang.Math.*;
```

Isso elimina a necessidade de usar o nome da classe, sob o custo de legibilidade:

```
double d = 4.6;  
long i = round(d);  
int x = -4;  
int y = abs(x);
```

## 14.10 - Exercícios: java.lang

- 1) Teste os exemplos desse capítulo, para ver que uma `String` é imutável. Por exemplo:

```
public class TestaString {  
  
    public static void main(String[] args) {  
        String s = "fj11";  
        s.replaceAll("1", "2");  
        System.out.println(s);  
    }  
  
}
```

Como fazer para ele imprimir fj22?

- 2) Utilize-se da documentação do Java e descubra de que classe é o objeto referenciado pelo atributo `out` da `System`.

Repare que poderíamos escrever:

```
PrintStream saida = System.out;  
saida.println("ola");
```

- 3) Crie e imprima uma referência para `Conta` (`ContaCorrente` OU `ContaPoupanca`, no caso de sua `Conta` ser abstrata):

```
Conta conta = new ContaCorrente();  
System.out.println(conta);
```

O que acontece?

- 4) Reescreva o método `toString` da sua classe `Conta` fazendo com que uma mensagem mais propícia seja devolvida. Lembre-se de aproveitar dos recursos do Eclipse para isto: digitando apenas o começo do nome do método a ser reescrito e pressionando `Ctrl+espaço`, ele vai sugerir reescrever o método, poupando você do trabalho de escrever a assinatura do mesmo, e de cometer algum engano.

```
public abstract class Conta {  
  
    private double saldo;  
  
    public String toString() {  
        return "esse objeto é uma conta com saldo R$" + this.saldo;  
    }  
  
    // restante da classe  
  
}
```

Agora imprima novamente uma referência a `Conta`. O que aconteceu?

- 5) Reescreva o método `equals` da classe `Conta` para que duas contas com o mesmo número de conta sejam consideradas iguais. Esboço:

```
abstract class Conta {
    private int numero;

    public boolean equals(Object obj) {
        Conta outraConta = (Conta) obj;

        return this.numero == outraConta.numero;
    }

    // restante
}
```

Compare duas instâncias de `Conta` com `==`, depois com `equals`, sendo que as instâncias são diferentes mas possuem o mesmo número.

- 6) Um `double` não está sendo suficiente para guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que poderia usar? (consulte a documentação, tente adivinhar onde você pode encontrar algo desse tipo pelos nomes dos pacotes, veja como é intuitivo).
- 7) (opcional) Agora faça com que o `equals` da sua classe `Conta` também leve em consideração a `String` do nome do cliente a qual ela pertence. Teste-a.
- 8) (opcional) Crie o `GuardadorDeObjetos` visto nesse capítulo, coloque uma `ContaPoupanca` na primeira posição, e tente recuperar essa conta fazendo um casting para `ContaCorrente`:
- 9) (opcional) Crie a classe `GuardadorDeObjetos` desse capítulo. Teste-a adicionando uma `ContaCorrente` e depois devolvendo como `ContaPoupanca`, usando casting. Repare na exception que é lançada:

```
GuardadorDeObjetos guardador = new GuardadorDeObjetos();
ContaCorrente cc = new ContaCorrente();
guardador.adicionaObjeto(cc);

// vai precisar do casting
// Ctrl+1 pro Eclipse gerar para você
ContaPoupanca cp = guardador.pegar(0);
```

Teste também o autoboxing do java 5.0, passando um inteiro para nosso guardador.

- 10) (opcional) Escreva um método que usa os métodos `charAt` e `length` de uma `String` para imprimir a mesma caractere a caractere, com cada caractere em uma linha diferente.
- 11) (opcional) Reescreva o método do exercício anterior, mas agora imprima a string de trás para a frente.
- 12) (opcional) Pesquise a classe `StringBuilder` (ou `StringBuffer` no java 1.4). Ela é mutável. Por que usá-la em vez da `String`? Quando usá-la?

## 14.11 - Desafio

- 1) Converta uma `String` para um número sem usar as bibliotecas do java que já fazem isso. Isso é, uma `String` `x = "762"` deve gerar um `int i = 762`.

Para ajudar, saiba que um char pode ser “transformado” em int com o mesmo valor numérico fazendo:

```
char c = '3';  
int i = c - '0'; // i vale 3!
```

Aqui estamos nos aproveitando do conhecimento da tabela unicode: os números de 0 a 9 estão em sequência! Você poderia usar o método estático `Character.getNumericValue(char)` em vez disso.

# Pacote java.io

*“A benevolência é sobretudo um vício do orgulho e não uma virtude da alma.”*  
– Doantien Alphonse François (Marquês de Sade)

Ao término desse capítulo, você será capaz de:

- ler e escrever bytes, caracteres e Strings de/para a entrada e saída padrão;
- ler e escrever bytes, caracteres e Strings de/para arquivos;
- utilizar buffers para agilizar a leitura e escrita através de fluxos;
- usar Scanner e PrintStream.

## 15.1 - Conhecendo uma API

Vamos passar, agora, a conhecer APIs do Java. `java.io` e `java.util` possuem as classes que você mais comumente vai usar, não importando se seu aplicativo é desktop, web, ou mesmo para celulares.

Apesar de ser importante conhecer nomes e métodos das classes mais utilizadas, o interessante aqui é que você enxergue que todos os conceitos previamente estudados são aplicados a toda hora nas classes da biblioteca padrão.

Não se preocupe em decorar nomes. Atenha-se em entender como essas classes estão relacionadas e como elas estão tirando proveito do uso de interfaces, polimorfismo, classes abstratas e encapsulamento. Lembre-se de estar com a documentação (javadoc) aberta durante o contato com esses pacotes.

Veremos também threads e sockets em capítulos posteriores, que ajudarão a condensar nosso conhecimento, tendo em vista que no exercício de sockets utilizaremos todos conceitos aprendidos, juntamente com as várias APIs.

## 15.2 - Orientação a objeto

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecido como **io**) é orientada a objetos e usa os principais conceitos mostrados até agora: interfaces, classes abstratas e polimorfismo.

A idéia atrás do polimorfismo no pacote `java.io` é de utilizar fluxos de entrada (`InputStream`) e de saída (`OutputStream`) para toda e qualquer operação, seja ela relativa a um **arquivo**, a um campo **blob** do banco de dados, a uma conexão remota via **sockets**, ou até mesmo às **entrada** e **saída padrão** de um programa (normalmente o teclado e o console).

As classes abstratas `InputStream` e `OutputStream` definem, respectivamente, o comportamento padrão dos fluxos em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes.

A grande vantagem dessa abstração pode ser mostrada em um método qualquer que utiliza um `OutputStream` recebido como argumento para escrever em um fluxo de saída. Para onde o método está escrevendo? Não se sabe e não importa: quando o sistema precisar escrever em um arquivo ou em uma socket, basta chamar o mesmo método, já que ele aceitar qualquer filha de `OutputStream`!

### 15.3 - `InputStream`, `InputStreamReader` e `BufferedReader`

Vamos ler um byte de um arquivo:

```
1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         int b = is.read();
5     }
6 }
```

A classe `InputStream` é abstrata e `FileInputStream`, uma de suas filhas concretas. Essa última recebe uma `String` como argumento no construtor, que é o nome do arquivo a ser lido. Ela vai procurar o arquivo no diretório em que o java foi invocado (no caso do Eclipse, vai ser a partir de dentro do diretório do projeto). Alternativamente você pode usar um caminho absoluto.

Quando trabalhamos com `java.io`, diversos métodos lançam `IOException`, que é uma exception do tipo checked - o que nos obriga a tratá-la ou declará-la. Nos exemplos aqui, estamos declarando `IOException` através da cláusula `throws` do `main` apenas para facilitar o exemplo. Caso a exception ocorra, a JVM vai parar, mostrando a stacktrace. Esta não é uma boa prática em uma aplicação real: trate suas exceptions para sua aplicação poder abortar elegantemente.

`InputStream` tem diversas outras filhas, como `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, entre outras.

Para recuperar um caractere, precisamos traduzir os bytes com o encoding dado para o respectivo código unicode, isso pode usar um ou mais bytes. Escrever esse decodificador é muito complicado, quem faz isso por você é a classe `InputStreamReader`.

```
1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         int c = isr.read();
6     }
7 }
```

O construtor de `InputStreamReader` pode receber o encoding a ser utilizado como parâmetro, se desejado, tal como UTF-8 ou ISO-8859-1.

### Encodings

Devido a grande quantidade de aplicativos internacionalizados de hoje em dia, é imprescindível que um bom programador entenda bem o que são os character encodings e o Unicode. O blog da Caelum possui um bom artigo a respeito:

<http://blog.caelum.com.br/2006/10/22/entendendo-unicode-e-os-character-encodings/>

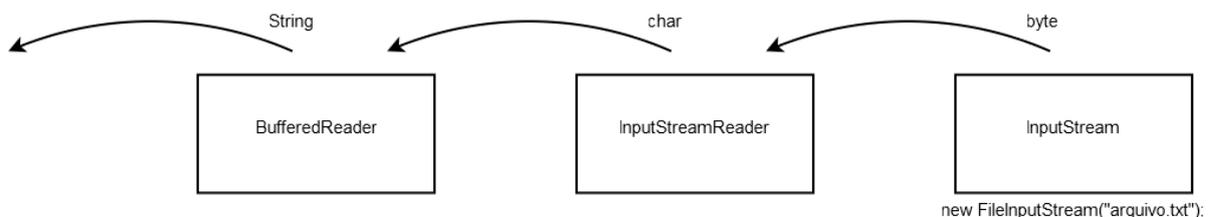
`InputStreamReader` é filha da classe abstrata `Reader`, que possui diversas outras filhas - são classes que manipulam chars.

Apesar da classe abstrata `Reader` já ajudar no trabalho de manipulação de caracteres, ainda seria difícil pegar uma `String`. A classe `BufferedReader` é um `Reader` que recebe outro `Reader` pelo construtor e concatena os diversos chars para formar uma `String` através do método `readLine`:

```
1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6         String s = br.readLine();
7     }
8 }
```

Como o próprio nome diz, essa classe lê do `Reader` por pedaços (usando o buffer) para evitar realizar muitas chamadas ao sistema operacional. Você pode até configurar o tamanho do buffer pelo construtor.

É essa a composição de classes que está acontecendo:



Esse padrão de composição é bastante utilizado e conhecido. É o **Decorator Pattern**.

Aqui, lemos apenas a primeira linha do arquivo. O método `readLine` devolve a linha que foi lida e muda o cursor para a próxima linha. Caso ele chegue ao fim do `Reader` (no nosso caso, fim do arquivo), ele vai devolver `null`. Então, com um simples laço, podemos ler o arquivo por inteiro:

```
1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6
7         String s = br.readLine(); // primeira linha
8
9         while(s != null) {
10            System.out.println(s);
11            s = br.readLine();
12        }
13    }
14 }
```

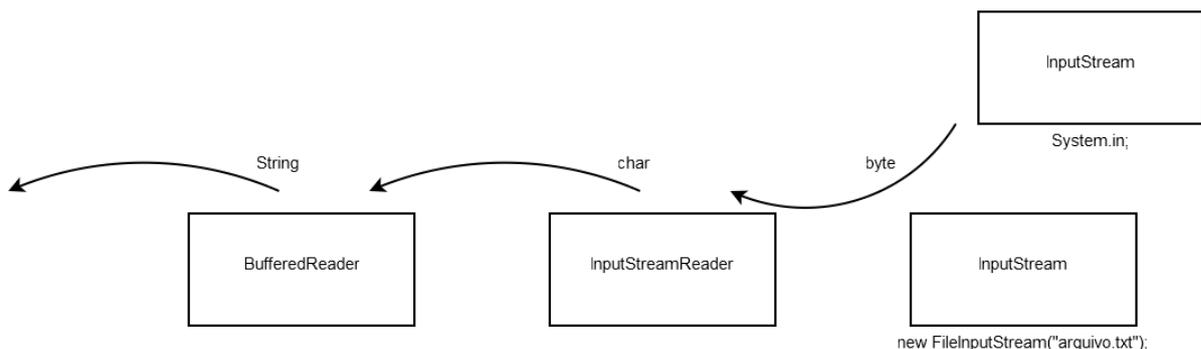
```
12     }
13   }
14 }
```

## 15.4 - Lendo Strings do teclado

Com um passe de magia, passamos a ler do teclado em vez de um arquivo, utilizando o `System.in`, que é uma referência a um `InputStream` o qual, por sua vez, lê da entrada padrão.

```
1 class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = System.in;
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6         String s = br.readLine();
7
8         while(s != null) {
9             System.out.println(s);
10            s = br.readLine();
11        }
12    }
13 }
```

Apenas modificamos a quem a variável `is` está se referindo. Podemos receber argumentos do tipo `InputStream` e ter esse tipo de abstração: não importa exatamente de onde estamos lendo esse punhado de bytes, desde que a gente receba a informação que estamos querendo. Como na figura:

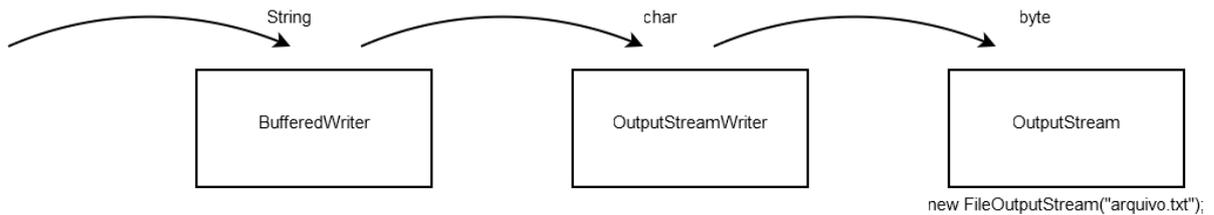


Repare que a ponta da direita poderia ser qualquer `InputStream`, seja `ObjectInputStream`, `AudioInputStream`, `ByteArrayInputStream`, ou a nossa `FileInputStream`. Polimorfismo! Ou você mesmo pode criar uma filha de `InputStream`, se desejar.

Por isso é muito comum métodos receberem e retornarem `InputStream`, em vez de suas filhas específicas. Com isso, elas desacoplam as informações e escondem a implementação, facilitando a mudança e manutenção do código. Repare que isso vai ao encontro de tudo o que aprendemos durante os capítulos que apresentaram classes abstratas, interfaces, polimorfismo e encapsulamento.

## 15.5 - A analogia na saída: OutputStream, OutputStreamWriter e BufferedWriter

Como você pode imaginar, escrever em um arquivo é o mesmo processo:



```

1 class TestaSaida {
2     public static void main(String[] args) throws IOException {
3         OutputStream os = new FileOutputStream("saida.txt");
4         OutputStreamWriter osw = new OutputStreamWriter(os);
5         BufferedWriter bw = new BufferedWriter(osw);
6
7         bw.write("caelum");
8
9         bw.close();
10    }
11 }

```

Lembre-se de dar *refresh* (clique da direita no nome do projeto, refresh) no seu projeto do Eclipse para que o arquivo criado apareça. O `FileOutputStream` pode receber um booleano como segundo parâmetro, para indicar se você quer reescrever o arquivo ou manter o que já estava escrito (`append`).

O método `write` do `BufferedWriter` não insere o(s) caractere(s) de quebra de linha. Para isso, você pode chamar o método `newLine`.

### Fechando o arquivo

É importante sempre fechar o arquivo. Você pode fazer isso chamando diretamente o método `close` do `FileInputStream/OutputStream`, ou ainda chamando o `close` do `BufferedReader/Writer`. Nesse último caso, o `close` será cascadeado para os objetos os quais o `BufferedReader/Writer` utiliza para realizar a leitura/escrita, além dele fazer o **flush** dos buffers no caso da escrita.

É comum que o `close` esteja dentro de um bloco `finally`. Se um arquivo for esquecido aberto e a referência para ele for perdida, ele será fechado pelo *garbage collector*, que veremos mais a frente, por causa do `finalize`. Mas não é bom você se prender a isso. Se você esquecer de fechar o arquivo, no caso de um programa minúsculo como esse, o programa vai terminar antes que o tal do *garbage collector* te ajude, resultando em um arquivo não escrito (os bytes ficaram no buffer do `BufferedWriter`).

Problemas similares podem acontecer com leitores que não forem fechados.

## 15.6 - Uma maneira mais fácil: Scanner e PrintStream

A partir do Java 5 temos a classe `java.util.Scanner`, que facilita bastante o trabalho de ler de um `InputStream`. Além disso, a classe `PrintStream` possui um construtor que já recebe o nome de um arquivo como argumento. Dessa forma, a leitura do teclado com saída para um arquivo ficou muito simples:

```
Scanner s = new Scanner(System.in);
PrintStream ps = new PrintStream("arquivo.txt");
while(s.hasNextLine()) {
    ps.println(s.nextLine());
}
```

Nenhum dos métodos lança `IOException`: `PrintStream` lança `FileNotFoundException` se você o construir passando uma `String`. Essa exceção é filha de `IOException` e indica que o arquivo não foi encontrado. O `Scanner` considerará que chegou ao fim se uma `IOException` for lançada, mas o `PrintStream` simplesmente engole exceptions desse tipo. Ambos possuem métodos para você verificar se algum problema ocorreu.

A classe `Scanner` é do pacote `java.util`. Ela possui métodos muito úteis para trabalhar com `Strings`, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares. Fica fácil parsear um arquivo com qualquer formato dado.

#### System.out

O atributo `out` da classe `System` é do tipo `PrintStream` (e, portanto, é um `OutputStream`).

## 15.7 - Um pouco mais...

- Existem duas classes chamadas `java.io.FileReader` e `java.io.FileWriter`. Elas são atalhos para a leitura e escrita de arquivos.
- O `do { .. } while(condicao);` é uma alternativa para se construir um laço. Pesquise-o e utilize-o no código para ler um arquivo, ele vai ficar mais sucinto (você não precisará ler a primeira linha fora do laço).

## 15.8 - Exercícios: Java I/O

1) Crie um programa (simplesmente uma classe com um `main`) que leia da entrada padrão. Para isso, você vai precisar de um `BufferedReader` que leia do `System.in` da mesma forma como fizemos:

```
1 public class TestaEntrada {
2     public static void main(String[] args) throws IOException {
3         InputStream is = System.in;
4         InputStreamReader isr = new InputStreamReader(is);
5         BufferedReader br = new BufferedReader(isr);
6
7         String linha = br.readLine(); // primeira linha
8
9         while(linha != null) {
10            System.out.println(linha);
11            linha = br.readLine();
12        }
13    }
14 }
```

O compilador vai reclamar que você não está tratando algumas exceções (como `java.io.IOException`). Utilize a cláusula `throws` para deixar “escapar” a exceção pelo seu `main`, ou use os devidos `try/catch`.

Utilize o quick fix do Eclipse para isso. Deixar todas as exceptions passarem despercebidas não é uma boa prática! Você pode usar aqui, pois estamos focando apenas no aprendizado da utilização do `java.io`.

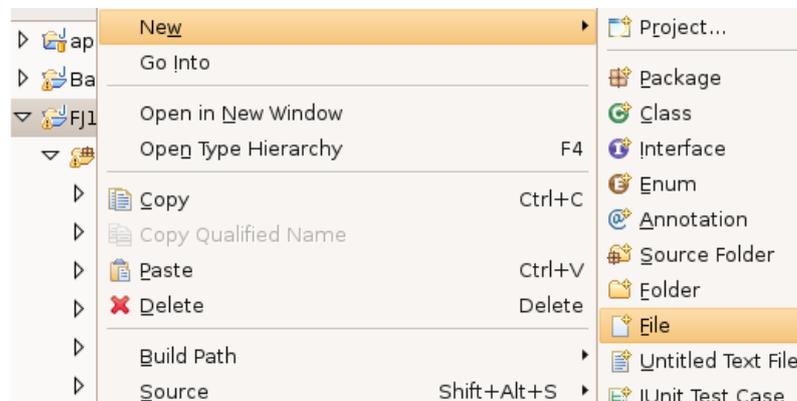
### EOF

Quando rodar sua aplicação, para encerrar a entrada de dados do teclado, é necessário enviarmos um sinal de fim de stream. É o famoso EOF, End Of File.

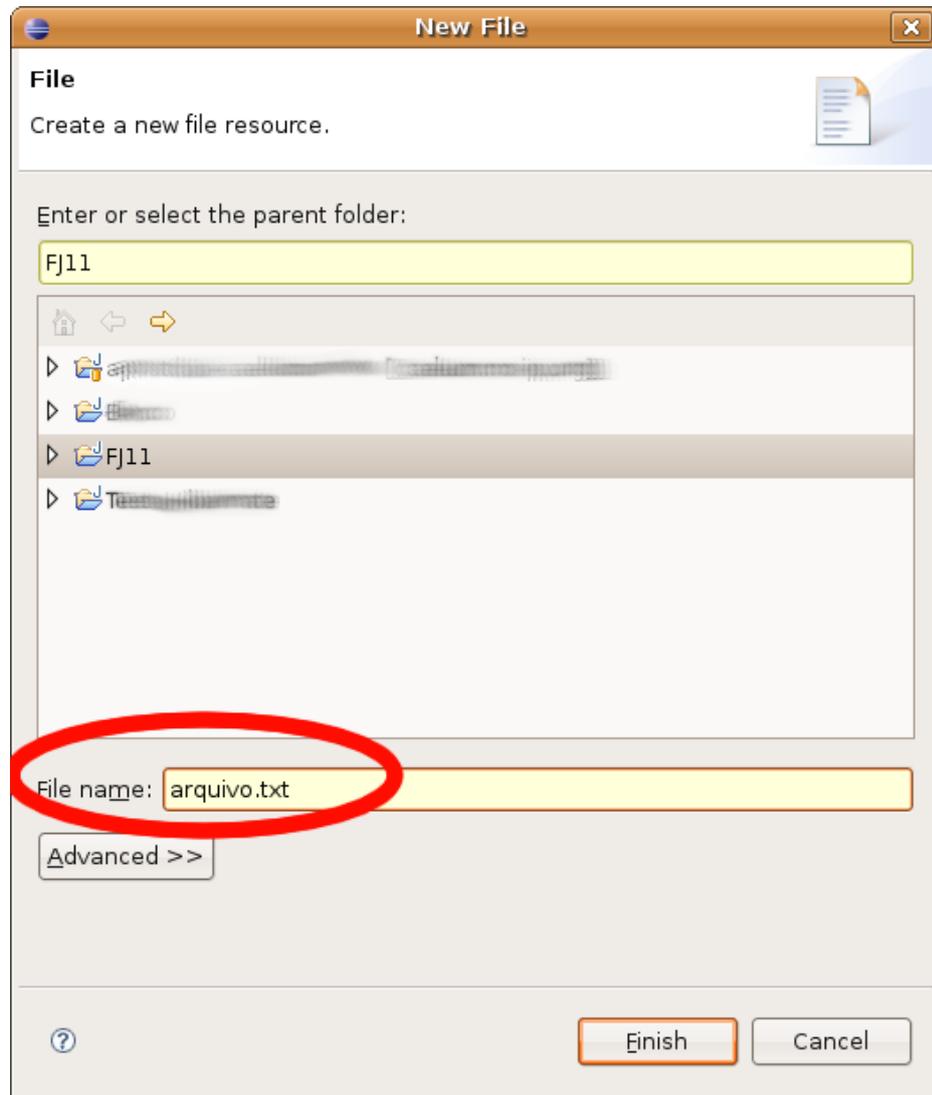
No Linux/Mac/Solaris/Unix você faz isso com o `CONTROL+D`. No Windows, use o `CONTROL+Z`.

2) Vamos ler de um arquivo, em vez do teclado. Antes, vamos criar o arquivo que será lido pelo programa:

a) Clique com o botão direito em cima do seu projeto e selecione New-> File:



b) Nesta tela, digite o nome do arquivo no lugar indicado:



c) Troque o `System.in` por um `FileInputStream`:

```
InputStream is = new FileInputStream("arquivo.txt");
```

3) (conceitual) Seu programa agora lê todas as linhas desse arquivo. Repare na utilização do polimorfismo. Como ambos são `InputStream`, isso faz com que eles se encaixem no `InputStreamReader`.

4) Utilize a classe `Scanner` para ler de um arquivo e colocar na tela. O código vai ficar incrivelmente pequeno.

```
1 class EntradaDeUmArquivo {
2     public static void main(String[] args) throws IOException {
3         InputStream is = new FileInputStream("arquivo.txt");
4
5         Scanner entrada = new Scanner(is);
6         while (entrada.hasNextLine()) {
7             System.out.println(entrada.nextLine());
8         }
9
10        is.close();
11    }
```

```
12 }
```

Depois troque a variável `is` para que ela se refira ao `System.in`. Agora você está lendo do teclado!

- 5) (opcional) Altere seu programa para que ele leia do arquivo e, em vez de jogar na tela, jogue em um arquivo. Você vai precisar, além do código anterior para ler de um arquivo, do código para escrever num arquivo. Para isso, você pode usar o `BufferedWriter`:

```
OutputStream os = new FileOutputStream("saida.txt");
OutputStreamWriter osw = new OutputStreamWriter(os);
BufferedWriter bw = new BufferedWriter(osw);
```

Agora, dentro do loop de leitura do teclado, você deve usar `bw.write(x)`, onde `x` é a linha que você leu. Use `bw.newLine()` para pular de linha. Não se esqueça de, no término do loop, dar um `bw.close()`. Você pode seguir o modelo:

```
String linha = br.readLine();

while (linha != null) {
    bw.write(linha);
    bw.newLine();
    // leia a proxima linha do buffered reader:
    linha = br.readLine();
}
bw.close();
```

Após rodar seu programa, dê um refresh no seu projeto (clique da direita no nome do projeto, refresh) e veja que ele criou um arquivo `saida.txt` no diretório.

- 6) (opcional) Altere novamente o programa para ele virar um pequeno editor: lê do teclado e escreve em arquivo. Repare que a mudança é mínima!
- 7) (opcional) Utilizando as classes `PrintStream` e `Scanner`, você pode reescrever o exercício de copiar arquivos de uma maneira muito sucinta, experimente!

## 15.9 - Discussão em aula: Design Patterns e o Template Method

Aplicar bem os conceitos de orientação a objetos é sempre uma grande dúvida. Sempre queremos encapsular direito, favorecer a flexibilidade, desacoplar classes, escrever código elegante e de fácil manutenção. E ouvimos falar que a Orientação a Objetos ajuda em tudo isso.

Mas, onde usar herança de forma saudável? Como usar interfaces? Onde o polimorfismo me ajuda? Como encapsular direito? Classes abstratas são usadas em que situações?

Muitos anos atrás, grandes nomes do mundo da orientação a objetos perceberam que criar bons designs orientados a objetos era um grande desafio para muitas pessoas. Perceberam que muitos problemas de OO apareciam recorrentemente em vários projetos; e que as pessoas já tinham certas soluções para esses problemas clássicos (nem sempre muito elegantes).

O que fizeram foi criar **soluções padrões para problemas comuns** na orientação a objetos, e chamaram isso de **Design Patterns**, ou Padrões de Projeto. O conceito vinha da arquitetura onde era muito comum ter esse tipo de solução. E, em 1994, ganhou grande popularidade na computação com o livro *Design Patterns*:

*Elements of Reusable Object-Oriented Software*, um catálogo com várias dessas soluções escrito por Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides (a Gangue dos Quatro, GoF).

*Design Patterns* tornou-se referência absoluta no bom uso da orientação a objetos. Outros padrões surgiram depois, em outras literaturas igualmente consagradas. O conhecimento dessas técnicas é imprescindível para o bom programador.

**Discuta com o instrutor como Design Patterns ajudam a resolver problemas de modelagem em sistemas orientados a objetos. Veja como Design Patterns são aplicados em muitos lugares do próprio Java.**

O instrutor comentará do Template Method e mostrará o código fonte do método `read()` da classe `java.io.InputStream`:

```
1 public int read(byte b[], int off, int len) throws IOException {
2     if (b == null) {
3         throw new NullPointerException();
4     } else if (off < 0 || len < 0 || len > b.length - off) {
5         throw new IndexOutOfBoundsException();
6     } else if (len == 0) {
7         return 0;
8     }
9
10    int c = read();
11    if (c == -1) {
12        return -1;
13    }
14
15    b[off] = (byte) c;
16
17    int i = 1;
18    try {
19        for (; i < len ; i++) {
20            c = read();
21            if (c == -1) {
22                break;
23            }
24            b[off + i] = (byte)c;
25        }
26    } catch (IOException ee) {
27    }
28    return i;
29 }
```

**Discuta em aula como esse método aplica conceitos importantes da orientação a objetos e promove flexibilidade e extensibilidade.**

# Collections framework

*“A amizade é um contrato segundo o qual nos comprometemos a prestar pequenos favores para que no-los retribuam com grandes.”*  
 – Baron de la Brede et de Montesquieu

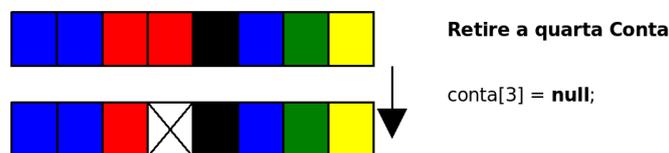
Ao término desse capítulo, você será capaz de:

- utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
- iterar e ordenar listas e coleções;
- usar mapas para inserção e busca de objetos.

## 16.1 - Arrays são trabalhosos, utilizar estrutura de dados

A utilização de arrays é complicada em muitos pontos:

- não podemos redimensionar um array em Java;
- é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
- não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares.



Na figura acima, você pode ver um array que antes estava sendo completamente utilizado e que, depois, teve um de seus elementos removidos.

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco?

Precisaremos procurar por um espaço vazio?

Guardaremos em alguma estrutura de dados externa, as posições vazias?

E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?

Há mais questões: como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

Além dessas dificuldades que os arrays apresentavam, faltava um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como listas ligadas e tabelas de espalhamento.

Com esses e outros objetivos em mente, a Sun criou um conjunto de classes e interfaces conhecido como **Collections Framework**, que reside no pacote `java.util` desde o Java2 1.2.

#### Collections

A **API do Collections** é robusta e possui diversas classes que representam estruturas de dados avançadas.

Por exemplo, não é necessário reinventar a roda e criar uma lista ligada, mas sim utilizar aquela que a Sun disponibilizou.

## 16.2 - Listas: `java.util.List`

Um primeiro recurso que a API de `Collections` traz são **listas**. Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.

Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos.

Ela resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho “infinito”,...). Esse código já está pronto!

A API de `Collections` traz a interface `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

A implementação mais utilizada da interface `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

#### ArrayList não é um array!

É comum confundirem uma `ArrayList` com um array, porém ela não é um array. O que ocorre é que, internamente, ela usa um array como estrutura para armazenar os dados, porém este atributo está propriamente encapsulado e você não tem como acessá-lo. Repare, também, que você não pode usar `[]` com uma `ArrayList`, nem acessar atributo `length`. Não há relação!

Para criar um `ArrayList`, basta chamar o construtor:

```
ArrayList lista = new ArrayList();
```

É sempre possível abstrair a lista a partir da interface `List`:

```
List lista = new ArrayList();
```

Para criar uma lista de nomes (`String`), podemos fazer:

```
List lista = new ArrayList();
lista.add("Guilherme");
lista.add("Paulo");
lista.add("Sérgio");
```

A interface `List` possui dois métodos `add`, um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição da mesma.

Note que, em momento algum, dizemos qual é o tamanho da lista; podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário.

Toda lista (na verdade, toda `Collection`) trabalha do modo mais genérico possível. Isto é, não há uma `ArrayList` específica para `Strings`, outra para `Números`, outra para `Datas` etc. **Todos os métodos trabalham com `Object`.**

Assim, é possível criar, por exemplo, uma lista de `Contas Correntes`:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(100);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(300);

List contas = new ArrayList();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

Para saber quantos elementos há na lista, podemos usar o método `size()`:

```
System.out.println(contas.size());
```

Há, ainda, um método `get(int)` que recebe como argumento o índice do elemento que se quer recuperar. Através dele, podemos fazer um `for` para iterar na lista de contas:

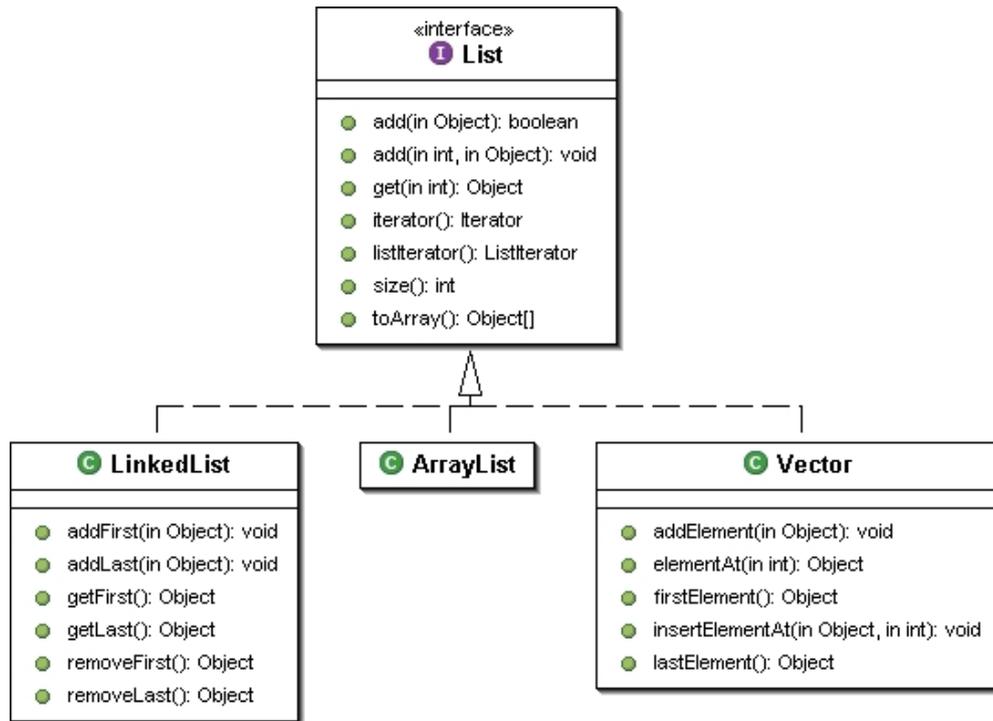
```
for(int i = 0; i < contas.size(); i++) {
    contas.get(i); // código não muito útil...
}
```

Mas como fazer para imprimir o saldo dessas contas? Podemos acessar o `getSaldo()` diretamente após fazer `contas.get(i)`? Não podemos; lembre-se que toda lista trabalha sempre com `Object`. Assim, a referência devolvida pelo `get(i)` é do tipo `Object`, sendo necessário o `cast` para `ContaCorrente` se quisermos acessar o `getSaldo()`:

```
for(int i = 0; i < contas.size(); i++) {
    ContaCorrente cc = (ContaCorrente) contas.get(i);
    System.out.println(cc.getSaldo());
}
// note que a ordem dos elementos não é alterada
```

Há ainda outros métodos como `remove()` que recebe um objeto que se deseja remover da lista; e `contains()`, que recebe um objeto como argumento e devolve `true` ou `false`, indicando se o elemento está ou não na lista.

A interface `List` e algumas classes que a implementam podem ser vistas no diagrama **UML** a seguir:



#### Acesso aleatório e percorrendo listas com `get`

Algumas listas, como a `ArrayList`, têm acesso aleatório aos seus elementos: a busca por um elemento em uma determinada posição é feita de maneira imediata, sem que a lista inteira seja percorrida (que chamamos de acesso sequencial).

Neste caso, o acesso através do método `get(int)` é muito rápido. Caso contrário, percorrer uma lista usando um `for` como esse que acabamos de ver, pode ser desastroso. Ao percorrermos uma lista, devemos usar **sempre** um `Iterator` ou `enhanced for`, como veremos.

Uma lista é uma excelente alternativa a um array comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

A outra implementação muito usada (`LinkedList`), fornece métodos adicionais para obter e remover o primeiro e último elemento da lista.

Vale ressaltar a importância do uso da interface `List`: quando desenvolvemos, procuramos sempre nos referir a ela, e não as implementações específicas. Por exemplo, se temos um método que vai buscar uma série de contas no banco de dados, poderíamos fazer assim:

```

class ContaDAO {
    public ArrayList<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<Conta>();

        // para cada conta do banco de dados, contas.add
    }
}
    
```

```
        return contas;
    }
}
```

Porém, pra que precisamos retornar a referência específica a uma `ArrayList`? Para que ser tão específico? Dessa maneira, o dia que optarmos por devolver uma `LinkedList` em vez de `ArrayList`, as pessoas que estão usando o método `buscaTodasContas` poderão ter problemas, pois estavam fazendo referência a uma `ArrayList`. O ideal é sempre trabalhar com a interface mais genérica possível:

```
class ContaDAO {

    // modificacao apenas no retorno:
    public List<Conta> buscaTodasContas() {
        ArrayList<Conta> contas = new ArrayList<Conta>();

        // para cada conta do banco de dados, contas.add

        return contas;
    }
}
```

Assim como no retorno, é boa prática trabalhar com a interface em todos os lugares possíveis: métodos recebem `List` em vez de uma implementação em específico. Também declaramos atributos como `List` em vez de nos comprometer como uma ou outra implementação. Dessa forma obtemos um **baixo acoplamento**: podemos trocar a implementação, já que estamos programando para a interface!

#### Vector

Outra implementação é a tradicional classe `Vector`, presente desde o Java 1.0, que foi adaptada para uso com o framework de `Collections`, com a inclusão de novos métodos. Ela deve ser tratada com cuidado pois lida de uma maneira diferente com processos correndo em paralelo e será mais lenta que uma `ArrayList` quando não houver acesso simultâneo aos dados.

## 16.3 - Listas no Java 5.0 com Generics

Em qualquer lista, é possível colocar qualquer `Object`. Com isso, é possível misturar objetos:

```
List lista = new ArrayList();
lista.add("Uma string");
lista.add(new ContaCorrente());
...
```

Mas e depois, na hora de recuperar esses objetos? Como o método `get` devolve um `Object`, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

Geralmente, não nos interessa uma lista com vários tipos de objetos misturados; no dia-a-dia, usamos listas como aquela de contas correntes. No Java 5.0, podemos usar o recurso de Generics para restringir as listas a um determinado tipo de objetos (e não qualquer `Object`):

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
contas.add(c1);
```

```
contas.add(c3);
contas.add(c2);
```

Repare no uso de um parâmetro ao lado de `List` e `ArrayList`: ele indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo `ContaCorrente`. Isso nos traz uma segurança em tempo de compilação:

```
contas.add("uma string"); // isso não compila mais!!
```

O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo `ContaCorrente`:

```
for(int i = 0; i < contas.size(); i++) {
    ContaCorrente cc = contas.get(i); // sem casting!
    System.out.println(cc.getSaldo());
}
```

## 16.4 - Ordenação: Collections.sort

Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.

A classe `Collections` traz um método estático `sort` que recebe um `List` como argumento e o ordena por ordem crescente. Por exemplo:

```
List lista = new ArrayList();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");
System.out.println(lista); //repare que o toString de ArrayList foi sobrescrito!
Collections.sort(lista);
System.out.println(lista);
```

Ao testar o exemplo acima, você observará que, primeiro, a lista é impressa na ordem de inserção e, depois de chamar o `sort`, ela é impressa em ordem alfabética.

Mas toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, `ContaCorrente`. E se quisermos ordenar uma lista de `ContaCorrente`? Em que ordem a classe `Collections` ordenará? Pelo saldo? Pelo nome do correntista?

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(500);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(150);

List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
contas.add(c1);
contas.add(c3);
```

```
contas.add(c2);  
  
Collections.sort(contas); // qual o critério de ordenação?
```

Sempre que falamos em ordenação, precisamos pensar em um **critério de ordenação**, uma forma de determinar qual elemento vem antes de qual. É necessário instruir o Java sobre como **comparar** nossas `ContaCorrente` a fim de determinar uma ordem na lista.

Vamos fazer com que os elementos da nossa coleção implementem a interface `java.lang.Comparable`, que define o método `int compareTo(Object)`. Este método deve retornar **zero**, se o objeto comparado for **igual** a este objeto, um número **negativo**, se este objeto for **menor** que o objeto dado, e um número **positivo**, se este objeto for **maior** que o objeto dado.

Para ordenar as `ContaCorrentes` por saldo, basta implementar o `Comparable`:

```
public class ContaCorrente extends Conta implements Comparable<ContaCorrente> {  
  
    // ... todo o codigo anterior fica aqui  
  
    public int compareTo(ContaCorrente outra) {  
  
        if(this.saldo < outra.saldo) {  
            return -1;  
        }  
  
        if(this.saldo > outra.saldo) {  
            return 1;  
        }  
  
        return 0;  
    }  
}
```

Com o código anterior, nossa classe tornou-se "**comparável**": dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método `compareTo` na classe.

Agora sim, quando chamarmos o método `sort` de `Collections`, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método `compareTo`.

Mas, e o exemplo anterior, com uma lista de `Strings`? Por que a ordenação funcionou, naquele caso, sem precisarmos fazer nada? Simples: quem escreveu a classe `String` (lembre que ela é uma classe normal do Java) implementou a interface `Comparable` e o método `compareTo` para `Strings`, fazendo comparação em ordem alfabética. (Consulte a documentação da classe `String` e veja o método `compareTo` lá).

### Definindo outros critérios de ordenação

É possível definir outros critérios de ordenação usando um objeto do tipo `Comparator`. Existe um método `sort` em `Collections` que recebe, além da `List`, um `Comparator` definindo um critério de ordenação específico. É possível ter vários `Comparators` com critérios diferentes para usar quando for necessário.

### Outros métodos da classe `Collections`

A classe `Collections` traz uma série de métodos estáticos úteis na manipulação de coleções (veremos outros tipos além de `List` mais adiante).

- `binarySearch(List, Object)`: Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
- `max(Collection)`: Retorna o maior elemento da coleção.
- `min(Collection)`: Retorna o menor elemento da coleção.
- `reverse(List)`: Inverte a lista.
- ...e muitos outros. Consulte a documentação para ver outros métodos.

Existe uma classe análoga, a `java.util.Arrays`, que faz operações similares com arrays.

## 16.5 - Exercícios: Ordenação

1) Abra sua classe `Conta` e veja se ela possui o atributo `numero`. Se não possuir, adicione-o:

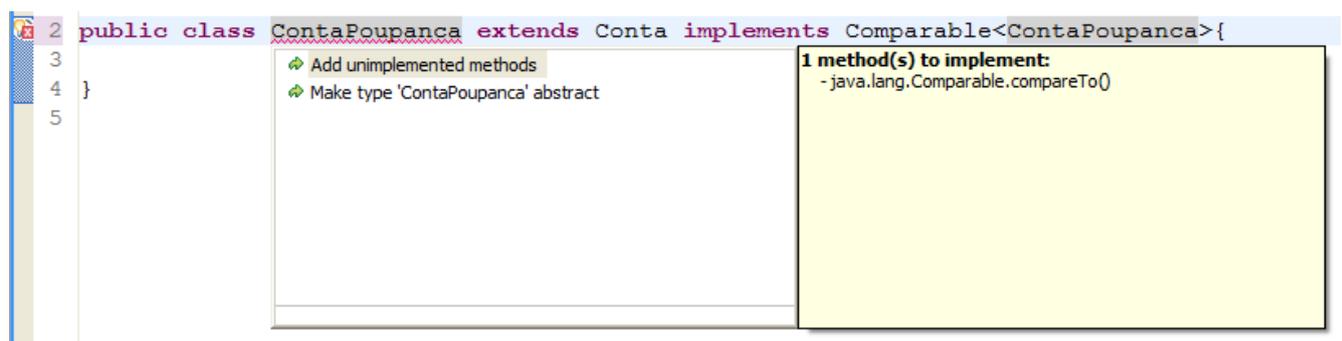
```
private int numero;
```

E gere o getter e o setter pelo Eclipse.

2) Faça sua classe `ContaPoupanca` implementar a interface `Comparable<ContaPoupanca>`. Utilize o critério de ordenar pelo número da conta ou pelo seu saldo (como visto no código deste capítulo).

```
public class ContaPoupanca extends Conta implements Comparable<ContaPoupanca>{
```

Repare que o Eclipse prontamente lhe oferecerá um quickfix, oferecendo a criação do esqueleto dos métodos definidos na interface `Comparable`:



```
public class ContaPoupanca extends Conta
    implements Comparable<ContaPoupanca> {

    // ... todo o codigo anterior fica aqui

    public int compareTo(ContaPoupanca outra) {
        if (this.getNumero() < outra.getNumero()) {
            return -1;
        }
        if (this.getNumero() > outra.getNumero()) {
            return 1;
        }
        return 0;
    }
}
```

#### Outra implementação...

O que acha da implementação abaixo?

```
public int compareTo(ContaPoupanca outra) {
    return this.getNumero() - outra.getNumero();
}
```

- 3) Crie uma classe `TestaOrdenacao`, onde você vai instanciar diversas contas e adicioná-las a uma `List<ContaPoupanca>`. Use o `Collections.sort()` nessa lista:

```
public class TestaOrdenacao {

    public static void main(String[] args) {

        List<ContaPoupanca> contas = new ArrayList<ContaPoupanca>();

        ContaPoupanca c1 = new ContaPoupanca();
        c1.setNumero(150);
        contas.add(c1);

        ContaPoupanca c2 = new ContaPoupanca();
        c2.setNumero(100);
        contas.add(c2);

        ContaPoupanca c3 = new ContaPoupanca();
        c3.setNumero(200);
        contas.add(c3);

        Collections.sort(contas);
    }
}
```

Faça um laço para imprimir todos os saldos das contas na lista já ordenada:

```
for(int i = 0; i < contas.size(); i++) {
    System.out.println("numero: " + contas.get(i).getNumero());
}
```

}

**Atenção especial:** repare que escrevemos um método `compareTo` em nossa classe e nosso código **nunca** o invoca!! Isto é muito comum. Reescrevemos (ou implementamos) um método e quem o invocará será um outro conjunto de classes (nesse caso, quem está chamando o `compareTo` é o `Collections.sort`, que o usa como base para o algoritmo de ordenação). Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe `Collections` nunca imaginou que ordenaria objetos do tipo `ContaPoupanca`, mas já que eles são `Comparable`, o seu método `sort` está satisfeito.

(opcional) Se preferir, insira novas contas através de um laço (`for`). Adivinhe o nome da classe para colocar saldos aleatórios? `Random`. Do pacote `java.util`. Consulte sua documentação para usá-la.

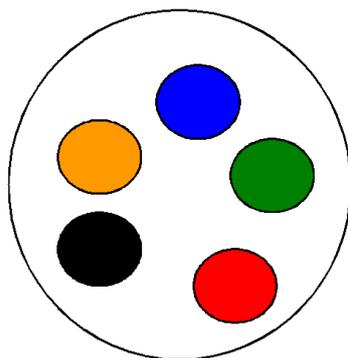
- 4) O que teria acontecido se a classe `ContaPoupanca` não implementasse `Comparable<ContaPoupanca>`?
- 5) (opcional) Imprima a referência para essa lista. Repare que o `toString` de uma `Array/LinkedList` é reescrito:

```
System.out.println(contas);
```

- 6) (opcional) Mude o critério de comparação da sua `ContaPoupanca`. Adicione um atributo `nomeDoCliente` na sua classe (caso ainda não exista algo semelhante) e tente mudar o `compareTo` para que uma lista de `ContaPoupanca` seja ordenada alfabeticamente pelo atributo `nomeDoCliente`.

## 16.6 - Conjunto: `java.util.Set`

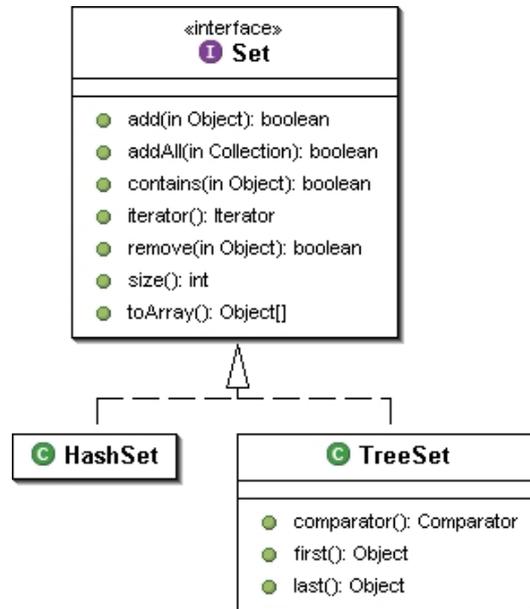
Um conjunto (`Set`) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.



### Possíveis ações em um conjunto:

- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- **Não existem elementos duplicados!**
- **Ao percorrer um conjunto, sua ordem não é conhecida!**



Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.

Tal ordem varia de implementação para implementação.

Um conjunto é representado pela interface `Set` e tem como suas principais implementações as classes `HashSet` e `TreeSet`.

O código a seguir cria um conjunto e adiciona três itens, apesar de tentar adicionar quatro:

```
Set conjunto = new HashSet();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");
conjunto.add("item 3");

// imprime a sequência na tela
System.out.println(conjunto);
```

O resultado são os elementos do conjunto, a ordem na qual eles aparecem pode, ou não, ser a ordem na qual eles foram inseridos e é incorreto supor que será sempre a mesma ordem!

#### Ordenando um set

Seria possível usar uma outra implementação de conjuntos, como um `TreeSet`, que insere os elementos de tal forma que, quando forem percorridos, eles apareçam em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface `java.lang.Comparable`. Ou, ainda, pode se passar um `Comparator` para seu construtor.

No Java 5.0, assim como as listas, conjuntos também podem ser parametrizados utilizando Generics, eliminando a necessidade de castings.

## 16.7 - Principais interfaces: java.util.Collection

As coleções têm como base a interface `Collection`, que define métodos para adicionar e remover um elemento, e verificar se ele está na coleção, entre outras operações, como mostra a tabela a seguir:

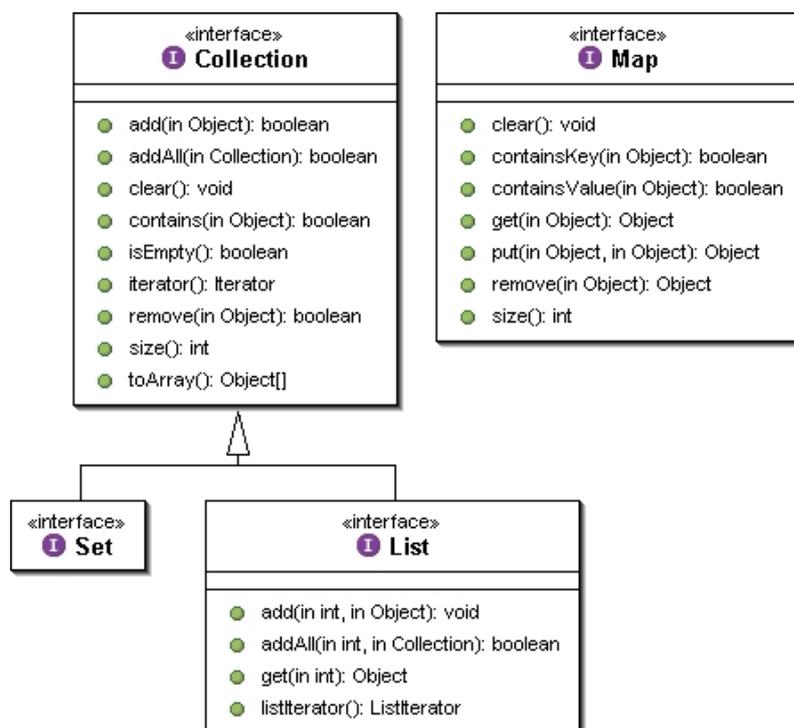
<code>boolean add(Object)</code>	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna <code>true</code> ou <code>false</code> indicando se a adição foi efetuada com sucesso.
<code>boolean remove(Object)</code>	Remove determinado elemento da coleção. Se ele não existia, retorna <code>false</code> .
<code>int size()</code>	Retorna a quantidade de elementos existentes na coleção.
<code>boolean contains(Object)</code>	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método <code>equals()</code> do objeto, e não através do operador <code>==</code> .
<code>Iterator iterator()</code>	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

Uma coleção pode implementar diretamente a interface `Collection`, porém normalmente se usa uma das duas subinterfaces mais famosas: justamente `Set` ou `List`.

A interface `Set`, como vimos, define um conjunto de elementos únicos enquanto a interface `List` permite a réplica de elementos.

A busca em um `Set` pode ser mais rápida do que em um objeto do tipo `List`, pois diversas implementações utilizam-se de tabelas de espalhamento (*hash tables*), trazendo a busca para tempo linear.

A interface `Map` faz parte do framework, mas não estende `Collection`. (veremos `Map` mais adiante).



No Java 5, temos outra interface filha de `Collection`: a `Queue`, que define métodos de entrada e de saída e cujo critério será definido pela sua implementação (por exemplo LIFO, FIFO ou ainda um heap onde cada elemento possui sua chave de prioridade).

## 16.8 - Percorrendo coleções no Java 5

Como percorrer os elementos de uma coleção? Se for uma lista, podemos sempre utilizar um laço `for`, chamando o método `get` para cada elemento. Mas e se a coleção não permitir indexação?

Por exemplo, um `Set` não possui uma função para pegar o primeiro, o segundo ou o quinto elemento do conjunto...

Podemos usar o **enhanced-for** (o `foreach`) do Java 5 para percorrer qualquer `Collection` sem se preocupar com essas coisas. Vimos antes o uso do `for` do Java 5 em listas, veja agora seu uso em `Set`:

```
1 Set conjunto = new HashSet();
2 conjunto.add("item 1");
3 conjunto.add("item 2");
4 conjunto.add("item 3");
5
6 // retorna o iterator
7 for (Object elemento : conjunto) {
8     String palavra = (String) elemento;
9     System.out.println(palavra);
10 }
```

O Java vai usar o `iterator` da `Collection` dada para percorrer a coleção. Se você já estiver usando uma coleção parametrizada, o `for` pode ser feito utilizando um tipo mais específico:

```
1 Set<String> conjunto = new HashSet<String>();
2 conjunto.add("item 1");
3 conjunto.add("item 2");
4 conjunto.add("item 3");
5
6 // retorna o iterator
7 for (String palavra : conjunto) {
8     System.out.println(palavra);
9 }
```

Em que ordem os elementos serão acessados?

Numa lista, os elementos aparecerão de acordo com o índice em que foram inseridos, isto é, de acordo com o que foi pré-determinado. Em um conjunto, a ordem depende da implementação da interface `Set`.

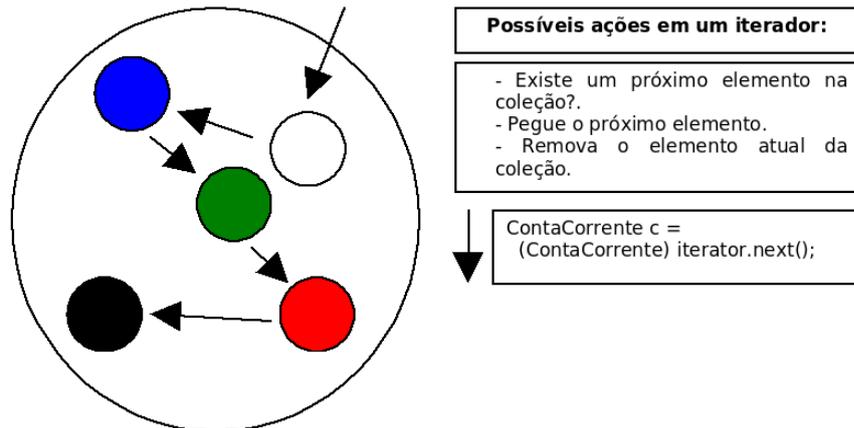
Por que o `Set` é, então, tão importante e usado?

Para perceber se um item já existe em uma lista, é muito mais rápido usar um `Set` do que um `List`, e os `TreeSets` já vêm ordenados de acordo com as características que desejarmos!

No eclipse, você pode escrever `foreach` e dar **ctrl+espaço**, que ele vai gerar o esqueleto desse `enhanced for`! Muito útil!

## 16.9 - Para saber mais: Iterando sobre coleções com `java.util.Iterator`

Antes do Java 5 introduzir o novo `enhanced-for`, iterações em coleções eram feitas com o `Iterator`. Toda coleção fornece acesso a um *iterator*, um objeto que implementa a interface `Iterator`, que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura abaixo mostra.



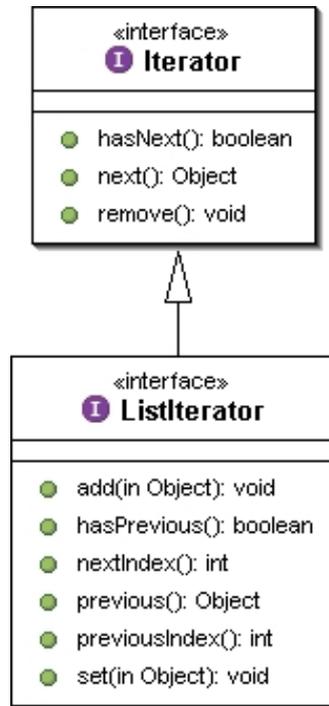
Ainda hoje (depois do Java 5) podemos usar o `Iterator` (que inclusive foi atualizado para usar `Generics`), mas o mais comum é usar o `enhanced-for`. E, na verdade, o `enhanced-for` é apenas um açúcar sintático que usa `iterator` por trás dos panos.

Primeiro criamos um `Iterator` que entra na coleção.

A cada chamada do método `next`, o `Iterator` retorna o próximo objeto do conjunto.

Um `iterator` pode ser obtido com o método `iterator()` de `Collection`, por exemplo:

```
Iterator i = lista.iterator();
```



A interface `Iterator` possui dois métodos principais: `hasNext()` (com retorno booleano), indica se ainda existe um elemento a ser percorrido; `next()`, retorna o próximo objeto.

Voltando ao exemplo do conjunto de strings, vamos percorrer o conjunto:

```
1 // Usando sintaxe de Java 1.4:
2
3 // popula o conjunto
4 Set conjunto = new HashSet();
5 conjunto.add("item 1");
6 conjunto.add("item 2");
7 conjunto.add("item 3");
8
9 // retorna o iterator
10 Iterator i = conjunto.iterator();
11 while (i.hasNext()) {
12
13     // recebe a palavra
14     Object elemento = i.next();
15     String palavra = (String) elemento;
16
17     // mostra a palavra
18     System.out.println(palavra);
19 }
```

O `while` anterior só termina quando todos os elementos do conjunto forem percorridos, isto é, quando o método `hasNext` mencionar que não existem mais itens.

### ListIterator

Uma lista fornece, além de acesso a um `Iterator`, um `ListIterator`, que oferece recursos adicionais, específicos para listas.

Usando o `ListIterator`, você pode, por exemplo, adicionar um elemento na lista ou voltar para o elemento que foi “iterado” anteriormente.

### Usar Iterator em Java 5

O `Iterator` pode sim ainda ser útil no Java 5. Além de iterar na coleção como faz o `enhanced-for`, o `iterator` consegue remover elementos da coleção durante a iteração, o que é bastante útil.

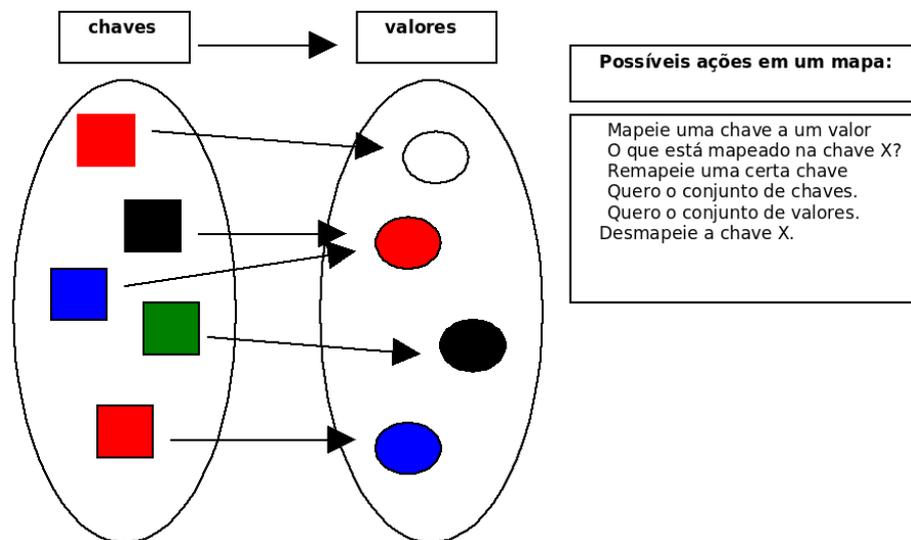
Além disso, no Java 5, o `Iterator` foi parametrizado e usa generics, facilitando castings e evitando erros.

## 16.10 - Mapas - `java.util.Map`

Um mapa é composto de uma associação de um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, usado em várias linguagens.

Algumas linguagens, como Perl ou PHP, possuem suporte nativo a mapas, onde são conhecidos como matrizes associativas.

`java.util.Map` é um mapa, pois é possível usá-lo para mapear uma chave, por exemplo: mapeie o valor “Caelum” à chave “escola”, ou mapeie “Rua Vergueiro” à chave “rua”.



O método `put(Object, Object)` da interface `Map` recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método `get(Object)`.

Observe o exemplo: criamos duas contas correntes e as colocamos em um mapa associando-as ao seu dono, respectivamente.

```
ContaCorrente c1 = new ContaCorrente();  
c1.deposita(10000);
```

```
ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);

// cria o mapa
Map mapaDeContas = new HashMap();

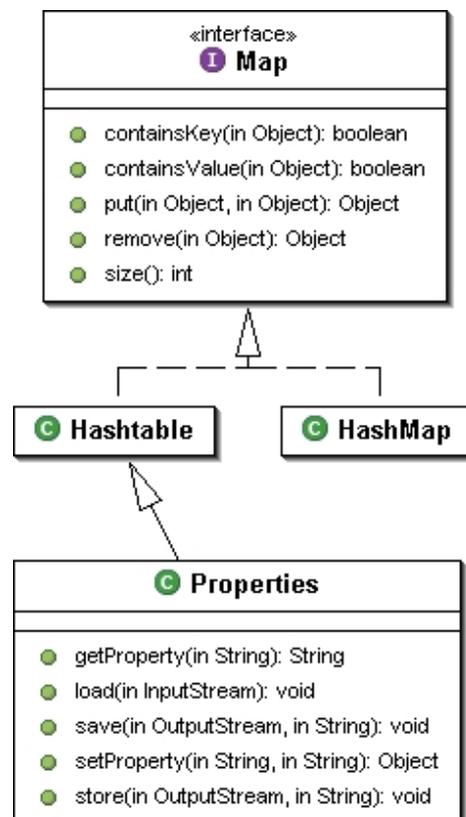
// adiciona duas chaves e seus valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);

// qual a conta do diretor?
Object elemento = mapaDeContas.get("diretor");
ContaCorrente contaDoDiretor = (ContaCorrente) elemento;
```

Um mapa, assim como as coleções, trabalha diretamente com `Objects` (tanto na chave quanto no valor), o que torna necessário o casting no momento que recuperar elementos.

Suas principais implementações são o `HashMap`, o `TreeMap` e o `Hashtable`.

Apesar do mapa fazer parte do framework, ele não implementa a interface `Collection`, por ter um comportamento bem diferente. Porém, as coleções internas de um mapa (a de chaves e a de valores, ver Figura 7) são acessíveis por métodos definidos na interface `Map`.



O método `keySet()` retorna um `Set` com as chaves daquele mapa e o método `values()` retorna a `Collection` com todos os valores que foram associados a alguma das chaves.

## 16.11 - Mapas no Java 5.0

Assim como as coleções, um mapa no Java 5.0 é parametrizado. O interessante é que ele recebe dois parâmetros: a chave e o valor:

```
1 ContaCorrente c1 = new ContaCorrente();
2 c1.deposita(10000);
3
4 ContaCorrente c2 = new ContaCorrente();
5 c2.deposita(3000);
6
7 // cria o mapa
8 Map<String,ContaCorrente> mapaDeContas = new HashMap<String,ContaCorrente>();
9
10 // adiciona duas chaves e seus valores
11 mapaDeContas.put("diretor", c1);
12 mapaDeContas.put("gerente", c2);
13
14 // qual a conta do diretor? (sem casting!)
15 ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
```

Aqui, como no caso da Lista, se você tentar colocar algo diferente de `String` na chave e `ContaCorrente` no valor, vai ter um erro de compilação.

## 16.12 - Para saber mais: Properties

Um mapa importante é a tradicional classe `Properties`, que mapeia strings e é muito utilizada para a configuração de aplicações.

A `Properties` possui, também, métodos para ler e gravar o mapeamento com base em um arquivo texto, facilitando muito a sua persistência.

```
Properties config = new Properties();

config.setProperty("database.login", "scott");
config.setProperty("database.password", "tiger");
config.setProperty("database.url","jdbc:mysql://localhost/teste");

// muitas linhas depois...

String login = config.getProperty("database.login");
String password = config.getProperty("database.password");
String url = config.getProperty("database.url");
DriverManager.getConnection(url, login, password);
```

Repare que não houve a necessidade do casting para `String` no momento de recuperar os objetos associados. Isto porque a classe `Properties` foi desenhada com o propósito de trabalhar com a associação entre Strings.

## 16.13 - Para saber mais: Equals e hashCode

Muitas das coleções do java guardam os objetos dentro de tabelas de hash. Essas tabelas são utilizadas para que a pesquisa de um objeto seja feita de maneira rápida.

Como funciona? Cada objeto é “classificado” pelo seu `hashCode` e, com isso, conseguimos espalhar cada objeto agrupando-os pelo `hashCode`. Quando buscamos determinado objeto, só vamos procurar entre os elementos que estão no grupo daquele `hashCode`. Dentro desse grupo, vamos testando o objeto procurado com o candidato usando `equals()`.

Para que isso funcione direito, o método `hashCode` de cada objeto deve retornar o mesmo valor para dois objetos, se eles são considerados `equals`. Em outras palavras:

a.`equals`(b) implica a.`hashCode`() == b.`hashCode`()

Implementar `hashCode` de tal maneira que ele retorne valores diferentes para dois objetos considerados `equals` quebra o contrato de `Object` e resultará em `collections` que usam espalhamento (como `HashSet`, `HashMap` e `Hashtable`), não achando objetos iguais dentro de uma mesma coleção.

#### No Eclipse...

O Eclipse é capaz de gerar uma implementação correta de `equals` e `hashCode` baseado nos atributos que você queira comparar. Basta ir no menu `Source` e depois em `Generate hashCode() and equals()`.

## 16.14 - Para saber mais: Boas práticas

As coleções do Java oferecem grande flexibilidade ao usuário. A perda de performance em relação à utilização de arrays é irrelevante, mas deve-se tomar algumas precauções:

- Grande parte das coleções usam, internamente, um array para armazenar os seus dados. Quando esse array não é mais suficiente, é criada um maior e o conteúdo da antiga é copiado. Este processo pode acontecer muitas vezes, no caso de você ter uma coleção que cresce muito. Você deve, então, criar uma coleção já com uma capacidade grande, para evitar o excesso de redimensionamento.
- Evite usar coleções que guardam os elementos pela sua ordem de comparação quando não há necessidade. Um `TreeSet` gasta computacionalmente  $O(\log(n))$  para inserir (ele utiliza uma árvore rubro-negra como implementação), enquanto o `HashSet` gasta apenas  $O(1)$ .
- Não itere sobre uma `List` utilizando um `for` de 0 até `list.size()` e usando `get(int)` para receber os objetos. Enquanto isso parece atraente, algumas implementações da `List` não são de acesso aleatório como a `LinkedList`, fazendo esse código ter uma péssima performance computacional. (use `Iterator`)

## 16.15 - Exercícios: Collections

1) Crie um código que insira 30 mil números numa `ArrayList` e pesquise-os. Vamos usar um método de `System` para cronometrar o tempo gasto:

```
1 public class TestaPerformance {
2
3     public static void main(String[] args) {
4         System.out.println("Iniciando...");
```

```
5     long inicio = System.currentTimeMillis();
6     Collection<Integer> teste = new ArrayList<Integer>();
7
8     for (int i = 0; i < 30000; i++) {
9         teste.add(i);
10    }
11
12    for (int i = 0; i < 30000; i++) {
13        teste.contains(i);
14    }
15
16    long fim = System.currentTimeMillis();
17    double tempo = (fim - inicio) / 1000.0;
18    System.out.println("Tempo gasto: " + tempo);
19 }
20 }
```

Troque a `ArrayList` por um `HashSet` e verifique o tempo que vai demorar:

```
Collection<Integer> teste = new HashSet<Integer>();
```

A diferença é mais que gritante. Se você passar de 30 mil para um número maior, como 50 ou 100 mil, verá que isso inviabiliza por completo o uso de uma `List`, no caso em que queremos utilizá-la essencialmente em pesquisas.

- 2) (conceitual, importante) Repare que, se você declarar a coleção e der `new` assim:

```
Collection<Integer> teste = new ArrayList<Integer>();
```

em vez de

```
ArrayList<Integer> teste = new ArrayList<Integer>();
```

É garantido que vai ter de alterar só essa linha para substituir a implementação por `HashSet`. Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código. Mais uma vez:  *programe voltado a interface, e não à implementação!* Esse é um **excelente** exemplo de bom uso de interfaces, afinal, de que importa como a coleção funciona? O que queremos é uma coleção qualquer, isso é suficiente para os nossos propósitos! Nosso código está com **baixo acoplamento** em relação a estrutura de dados utilizada: podemos trocá-la facilmente.

Esse é um código extremamente elegante e flexível. Com o tempo você vai reparar que as pessoas programar sempre se referindo a essas interfaces menos específicas: os métodos costumam receber e devolver `Collection`s, `List`s e `Set`s em vez de referenciar diretamente uma implementação. É o mesmo que ocorre com o uso de `InputStream` e `OutputStream`: eles são o suficiente, não há porque forçar o uso de algo mais específico.

Obviamente, algumas vezes não conseguimos trabalhar dessa forma e precisamos usar uma interface mais específica ou mesmo nos referir ao objeto pela sua implementação para poder chamar alguns métodos. Por exemplo, `TreeSet` tem mais métodos que os definidos em `Set`, assim como `LinkedList` em relação a `List`.

- 3) Faça testes com o `Map<String, ContaCorrente>`, como visto nesse capítulo.

```
1 public class TestaMapa {
2
```

```
3     public static void main(String[] args) {
4         Conta c1 = new ContaCorrente();
5         c1.deposita(10000);
6
7         Conta c2 = new ContaCorrente();
8         c2.deposita(3000);
9
10        // cria o mapa
11        Map mapaDeContas = new HashMap();
12
13        // adiciona duas chaves e seus valores
14        mapaDeContas.put("diretor", c1);
15        mapaDeContas.put("gerente", c2);
16
17        // qual a conta do diretor?
18        Object elemento = mapaDeContas.get("diretor");
19        Conta contaDoDiretor = (Conta) elemento;
20        System.out.println(contaDoDiretor.getSaldo());
21    }
22 }
```

Depois, altere o código para usar o generics, e não haver a necessidade do casting, além da garantia de que nosso mapa estará seguro em relação a tipagem usada.

Você pode utilizar o quickfix do Eclipse para ele consertar isso, na linha em que você está chamando o put. Depois de mais um quick fiz (descubra!) seu código deve ficar como segue:

```
// cria o mapa
Map<String, Conta> mapaDeContas = new HashMap<String, Conta>();
```

- 4) (opcional) Assim como no exercício 1, crie uma comparação entre `ArrayList` e `LinkedList`, para ver qual é a mais rápida para se adicionar elementos na primeira posição (`list.add(0, elemento)`) e qual é a mais rápida para se percorrer usando o `get(indice)` (sabemos que o correto seria utilizar o enhanced for ou o iterator).
- 5) (opcional) Crie uma classe `Banco` que possui lista de `Conta`. Repare que numa lista de `Conta`, você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca`. Crie um método `void adiciona(Conta c)`, um método `Conta pega(int x)` e outro `int pegaTotalDeContas()`, muito similar à relação anterior de `Empresa-Funcionário`.
- 6) (opcional) Crie um método na classe `Banco` que busca por uma determinada `Conta` e informa se ela existe.
- 7) (opcional, avançado) Crie o método `hashCode` para a sua conta, de forma que ele respeite o `equals` de que duas contas são `equals` quando tem o mesmo número. Verifique se sua classe funciona corretamente num `HashSet`. Remova o método `hashCode`. Continua funcionando?

Dominar o uso e o funcionamento do `hashCode` é fundamental para o bom programador.

## 16.16 - Desafios

- 1) Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `TreeSet`.
- 2) Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `ArrayList`.

# Programação Concorrente e Threads

“O único lugar onde o sucesso vem antes do trabalho é no dicionário.”  
– Albert Einstein

Ao término desse capítulo, você será capaz de:

- executar tarefas simultaneamente;
- colocar tarefas para aguardar até que um determinado evento ocorra;
- entender o funcionamento do Garbage Collector.

## 17.1 - Threads

### “Duas coisas ao mesmo tempo”

Em várias situações precisamos “rodar duas coisas ao mesmo tempo”. Imagine um programa que gera um relatório muito grande em PDF. É um processo demorado e, para dar alguma satisfação para o usuário, queremos mostrar uma barra de progresso. Queremos então gerar o PDF e *ao mesmo tempo* atualizar a barrinha.

Pensando um pouco mais amplamente, quando usamos o computador também fazemos várias coisas simultaneamente: queremos navegar na internet e *ao mesmo tempo* ouvir música.

A necessidade de se fazer várias coisas simultaneamente, ao mesmo tempo, **paralelamente** aparece frequentemente na computação. Para vários programas distintos, normalmente o próprio sistema operacional gerencia isso através de vários *processos* em paralelo.

Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de **Threads**.

### Threads em Java

Em Java, usamos a classe `Thread` do pacote `java.lang` para criarmos *linhas de execução* paralelas. A classe `Thread` recebe como argumento um objeto com o código que desejamos rodar. Por exemplo, no programa de PDF e barra de progresso:

```
public class GeraPDF {  
    public void rodar () {  
        // lógica para gerar o pdf...
```

```
    }  
}  
  
public class BarraDeProgresso {  
    public void rodar () {  
        // mostra barra de progresso e vai atualizando ela...  
    }  
}
```

E, no método main, criamos os objetos e passamos para a classe Thread. O método start é responsável por iniciar a execução da Thread:

```
public class MeuPrograma {  
    public static void main (String[] args) {  
  
        GeraPDF gerapdf = new GeraPDF();  
        Thread threadDoPdf = new Thread(gerapdf);  
        threadDoPdf.start();  
  
        BarraDeProgresso barraDeProgresso = new BarraDeProgresso();  
        Thread threadDaBarra = new Thread(barraDeProgresso);  
        threadDaBarra.start();  
  
    }  
}
```

O código acima, porém, não funcionará. Como a classe thread sabe que deve chamar o método `rodar`? Como ela sabe que nome de método daremos e que ela deve chamar esse método especial? Falta na verdade um **contrato** entre as nossas classes a serem executadas e a classe Thread.

Esse contrato existe e é feito pela *interface* `Runnable`: devemos dizer que nossa classe é “rodável” (executável) e que segue esse contrato. Na interface `Runnable` há apenas um método chamado `run`. Basta implementá-lo, “assinar” o contrato e a classe Thread já saberá executar nossa classe.

```
public class GeraPDF implements Runnable {  
    public void run () {  
        // lógica para gerar o pdf...  
    }  
}  
  
public class BarraDeProgresso implements Runnable {  
    public void run () {  
        // mostra barra de progresso e vai atualizando ela...  
    }  
}
```

A classe Thread recebe no construtor um objeto que **é um** `Runnable`, e seu método `start` chama o método `run` da nossa classe. Repare que a classe Thread não sabe qual é o tipo específico da nossa classe; para ela, basta saber que a classe segue o contrato estabelecido e possui o método `run`.

É o bom uso de interfaces, contratos e polimorfismo na prática!

### Estendendo a classe Thread

A classe Thread implementa Runnable. Então, você pode criar uma subclasse dela e reescrever o run, que, na classe Thread, não faz nada:

```
public class GeraPDF extends Thread {
    public void run () {
        // ...
    }
}
```

E, como nossa classe é uma Thread, podemos usar o start diretamente:

```
GeraPDF gera = new GeraPDF();
gera.start();
```

Apesar de ser um código mais simples, você está usando herança apenas por “preguiça” (herdamos um monte de métodos mas usamos apenas o run), e não por polimorfismo, que seria a grande vantagem. Prefira implementar Runnable a herdar de Thread.

### Dormindo

Para que a thread atual durma basta chamar o método a seguir, por exemplo, para dormir 3 segundos:

```
Thread.sleep(3 * 1000);
```

## 17.2 - Escalonador e trocas de contexto

Veja a classe a seguir:

```
1 public class Programa implements Runnable {
2
3     private int id;
4     // colocar getter e setter pro atributo id
5
6     public void run () {
7         for (int i = 0; i < 10000; i++) {
8             System.out.println("Programa " + id + " valor: " + i);
9         }
10    }
11 }
```

É uma classe que implementa Runnable e, no método run, apenas imprime dez mil números. Vamos usá-las duas vezes para criar duas threads e imprimir os números duas vezes simultaneamente:

```
1 public class Teste {
2     public static void main(String[] args) {
3
4         Programa p1 = new Programa();
5         p1.setId(1);
6
7         Thread t1 = new Thread(p1);
```

```
8         t1.start();
9
10        Programa p2 = new Programa();
11        p2.setId(2);
12
13        Thread t2 = new Thread(p2);
14        t2.start();
15
16    }
17 }
```

Se rodarmos esse programa, qual será a saída? De um a mil e depois de um a mil? Provavelmente não, senão seria sequencial. Ele imprimirá 0 de t1, 0 de t2, 1 de t1, 1 de t2, 2 de t1, 2 de t2 e etc? Exatamente intercalado?

Na verdade, não sabemos exatamente qual é a saída. Rode o programa várias vezes e observe: em cada execução a saída é um pouco diferente.

O problema é que no computador existe apenas um processador capaz de executar coisas. E quando queremos executar várias coisas ao mesmo tempo, e o processador só consegue fazer uma coisa de cada vez? Entra em cena o **escalonador de thread**.

O escalonador (scheduler), sabendo que apenas uma coisa pode ser executada de cada vez, pega todas as threads que precisam ser executadas e faz o processador ficar alternando a execução de cada uma delas. A idéia é executar um pouco de cada thread e fazer essa troca tão rapidamente que a impressão que fica é que as coisas estão sendo feitas ao mesmo tempo.

O escalonador é responsável por escolher qual a próxima thread a ser executada e fazer a **troca de contexto** (context switch). Ele primeiro salva o estado da execução da thread atual para depois poder retomar a execução da mesma. Aí ele restaura o estado da thread que vai ser executada e faz o processador continuar a execução desta. Depois de um certo tempo, esta thread é tirada do processador, seu estado (o contexto) é salvo e outra thread é colocada em execução. A *troca de contexto* é justamente as operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.

Quando fazer a troca de contexto, por quanto tempo a thread vai rodar e qual vai ser a próxima thread a ser executada, são escolhas do escalonador. Nós não controlamos essas escolhas (embora possamos dar “dicas” ao escalonador). Por isso que nunca sabemos ao certo a ordem em que programas paralelos são executados.

Você pode pensar que é ruim não saber a ordem. Mas perceba que se a ordem importa para você, se é importante que determinada coisa seja feita antes de outra, então não estamos falando de execuções paralelas, mas sim de um programa sequencial normal (onde uma coisa é feita depois da outra, em uma sequência).

Todo esse processo é feito automaticamente pelo escalonador do Java (e, mais amplamente, pelo escalonador do sistema operacional). Para nós, programadores das threads, é como se as coisas estivessem sendo executadas ao mesmo tempo.

### E em mais de um processador?

A VM do Java e a maioria dos SOs modernos consegue fazer proveito de sistemas com vários processadores ou multi-core. A diferença é que agora temos mais de um processador executando coisas e teremos, sim, execuções verdadeiramente paralelas.

Mas o número de processos no SO e o número de Threads paralelas costumam ser tão grandes que, mesmo com vários processadores, temos as trocas de contexto. A diferença é que o escalonador tem dois ou mais processadores para executar suas threads. Mas dificilmente terá uma máquina com mais processadores que threads paralelas executando.

## 17.3 - Garbage Collector

O **Garbage Collector** (coletor de lixo, lixeiro) é uma Thread responsável por jogar fora todos os objetos que não estão sendo referenciados por nenhuma outra Thread - seja de maneira direta ou indireta.

Considere o código:

```
Conta conta1 = new ContaCorrente();  
Conta conta2 = new ContaCorrente();
```

Até este momento, sabemos que temos 2 objetos em memória. Aqui, o Garbage Collector não pode eliminar nenhum dos objetos, pois ainda tem alguém se referindo a eles de alguma forma.

Podemos, então, executar uma linha que nos faça perder a referência para um dos dois objetos criados, como, por exemplo, o seguinte código:

```
conta2 = conta1;
```

Quantos objetos temos em memória?

Perdemos a referência para um dos objetos que foram criados. Esse objeto já não é mais acessível. Temos, então, apenas um objeto em memória?

Não podemos afirmar isso! Como o Garbage Collector é uma Thread, você não tem garantia de quando ele vai rodar. Você só sabe que, em algum momento no futuro, aquela memória vai ser liberada.

Algumas pessoas costumam atribuir `null` a uma variável, com o intuito de acelerar a passagem do Garbage Collector por aquele objeto:

```
for(int i = 0; i < 100; i++) {  
    List x = new ArrayList();  
    // faz algumas coisas com a arraylist  
    x = null;  
}
```

Isso não é necessário. O Garbage Collector age apenas sobre objetos, nunca sobre variáveis. Nesse caso, a variável `x` não existirá mais a cada iteração, deixando a `ArrayList` criada sem nenhuma referência para ela.

Em alguns casos particulares, é interessante atribuir `null`, por exemplo, quando você tem uma referência que é um atributo e já não vai mais usar aquele objeto.

### System.gc()

Você nunca consegue forçar o Garbage Collector, mas chamando o método estático `gc` da classe `System`, você está sugerindo que a Virtual Machine rode o Garbage Collector naquele momento. Se sua sugestão vai ser aceita ou não, isto depende de JVM para JVM, e você não tem garantias. Evite o uso deste método. Você não deve basear sua aplicação em quando o Garbage Collector vai rodar ou não.

### Finalizer

A classe `Object` define também um método `finalize`, que você pode reescrever. Esse método será chamado no instante antes do Garbage Collector coletar este objeto. Não é um destrutor, você não sabe em que momento ele será chamado. Algumas pessoas o utilizam para liberar recursos “caros” como conexões, threads e recursos nativos. Isso deve ser utilizado apenas por segurança: o ideal é liberar esses recursos o mais rápido possível, sem depender da passagem do Garbage Collector.

## 17.4 - Para saber mais: problemas com concorrência

O uso de Threads começa a ficar interessante e complicado quando precisamos compartilhar objetos entre várias Threads.

Imagine a seguinte situação: temos um Banco com milhões de Contas Bancárias. Clientes sacam e depositam dinheiro continuamente, 24 horas por dia. No primeiro dia de cada mês, o Banco precisa atualizar o saldo de todas as Contas de acordo com uma taxa específica. Para isso, ele utiliza o `AtualizadorDeContas` que vimos anteriormente.

O `AtualizadorDeContas`, basicamente, pega uma a uma cada uma das milhões de contas e chama seu método `atualiza`. A atualização de milhões de contas é um processo demorado, que dura horas; é inviável parar o banco por tanto tempo até que as atualizações tenham completado. É preciso executar as atualizações paralelamente às atividades, de depósitos e saques, normais do banco.

Ou seja, teremos várias threads rodando paralelamente. Em uma thread, pegamos todas as contas e vamos chamando o método `atualiza` de cada uma. Em outra, podemos estar sacando ou depositando dinheiro. Estamos compartilhando objetos entre múltiplas threads (as contas, no nosso caso).

Agora imagine a seguinte possibilidade (mesmo que muito remota): no exato instante em que o atualizador está atualizando uma Conta X, o cliente dono desta Conta resolve efetuar um saque. Como sabemos, ao trabalhar com Threads, o escalonador pode parar uma certa Thread a qualquer instante para executar outra, e você não tem controle sobre isso.

Veja essa classe `Conta`:

```
1 public class Conta {
2
3     private double saldo;
4
5     // outros metodos e atributos...
6
7     public void atualiza(double taxa) {
8         double saldoAtualizado = this.saldo * (1 + taxa);
9         this.saldo = saldoAtualizado;
10    }
11
```

```
12     public void deposita(double valor) {
13         double novoSaldo = this.saldo + valor;
14         this.saldo = novoSaldo;
15     }
16 }
```

Imagine uma Conta com saldo de 100 reais. Um cliente entra na agência e faz um depósito de 1000 reais. Isso dispara uma Thread no banco que chama o método `deposita()`; ele começa calculando o `novoSaldo` que passa a ser 1100 (linha 13). Só que por algum motivo que desconhecemos, o escalonador pára essa thread.

Neste exato instante ele começa a executar uma outra Thread que chama o método `atualiza` da mesma Conta, por exemplo, com taxa de 1%. Isso quer dizer que o `novoSaldo` passa a valer 101 reais (linha 8). E, nesse instante o escalonador troca de Threads novamente. Agora ele executa a linha 14 na Thread que fazia o depósito; o saldo passa a valer 1100. Acabando o `deposita`, o escalonador volta pra Thread do `atualiza` e executa a linha 9, fazendo o saldo valer 101 reais.

Resultado: o depósito de mil reais foi totalmente ignorado e seu Cliente ficará pouco feliz com isso. Perceba que não é possível detectar esse erro, já que todo o código foi executado perfeitamente, sem problemas. O problema, aqui, foi o acesso simultâneo de duas Threads ao mesmo objeto.

E o erro só ocorreu porque o escalonador parou nossas Threads naqueles exatos lugares. Pode ser que nosso código fique rodando 1 ano sem dar problema algum e em um belo dia o escalonador resolve alternar nossas Threads daquela forma. Não sabemos como o escalonador se comporta! Temos que proteger nosso código contra esse tipo de problema. Dizemos que essa classe não é *thread safe*, isso é, não está pronta para ter uma instância utilizada entre várias threads concorrentemente.

O que queríamos era que não fosse possível alguém atualizar a `Conta` enquanto outra pessoa está depositando um dinheiro. Queríamos que uma Thread não pudesse mexer em uma `Conta` enquanto outra Thread está mexendo nela. Não há como impedir o escalonador de fazer tal escolha. Então, o que fazer?

Uma idéia seria criar uma **trava** e, no momento em que uma Thread entrasse em um desses métodos, ela trancaria a entrada com uma chave. Dessa maneira, mesmo que sendo colocada de lado, nenhuma outra Thread poderia entrar nesses métodos, pois a chave estaria com a outra Thread.

Essa idéia é chamada de **região crítica**. É um pedaço de código que definimos como crítico e que não pode ser executado por duas threads ao mesmo tempo. Apenas uma thread por vez consegue entrar na região crítica.

Podemos fazer isso em Java. Podemos usar qualquer objeto como um **lock** (trava, chave), para poder **sincronizar** em cima desse objeto, isto é, se uma Thread entrar em um bloco que foi definido como sincronizado por esse lock, apenas uma Thread poderá estar lá dentro ao mesmo tempo, pois a chave estará com ela.

A palavra chave `synchronized` dá essa característica a um bloco de código e recebe qual é o objeto que será usado como chave. A chave só é devolvida no momento em que a Thread que tinha essa chave sair do bloco, seja por `return` ou disparo de uma exceção (ou ainda na utilização do método `wait()`).

Queremos, então, bloquear o acesso simultâneo a uma mesma `Conta`:

```
public class Conta {

    private double saldo;

    // outros metodos e atributos...
```

```
public void atualiza(double taxa) {
    synchronized (this) {
        double saldoAtualizado = this.saldo * (1 + taxa);
        this.saldo = saldoAtualizado;
    }
}

public void deposita(double valor) {
    synchronized (this) {
        double novoSaldo = this.saldo + valor;
        this.saldo = novoSaldo;
    }
}
}
```

Observe o uso dos blocos `synchronized` dentro dos dois métodos. Eles bloqueiam uma Thread utilizando o mesmo objeto `Conta`, o `this`.

Esses métodos agora são mutuamente exclusivos e só executam de maneira atômica. Threads que tentam pegar um lock que já está pego, ficarão em um conjunto especial esperando pela liberação do lock (não necessariamente numa fila).

#### Sincronizando o bloco inteiro

É comum sempre sincronizarmos um método inteiro, normalmente utilizando o `this`.

```
public void metodo() {
    synchronized (this) {
        // conteudo do metodo
    }
}
```

Para este mesmo efeito, existe uma sintaxe mais simples, onde o `synchronized` pode ser usado como modificador do método:

```
public synchronized void metodo() {
    // conteudo do metodo
}
```

#### Mais sobre locks e monitores e concorrência

Se o método for estático, será sincronizado usando o lock do objeto que representa a classe (`NomeDaClasse.class`).

Além disso, o pacote `java.util.concurrent`, conhecido como **JUC**, entrou no Java 5.0 para facilitar uma série de trabalhos comuns que costumam a aparecer em uma aplicação concorrente.

## 17.5 - Para saber mais: Vector e Hashtable

Duas collections muito famosas são `Vector` e `Hashtable`, a diferença delas com suas irmãs `ArrayList` e `HashMap` é que as primeiras são thread safe.

Você pode se perguntar porque não usamos sempre essas classes thread safe. Adquirir um lock tem um custo, e caso um objeto não va ser usado entre diferentes threads, não há porque usar essas classes que consomem mais recursos. Mas nem sempre é fácil enxergar se devemos sincronizar um bloco, ou se devemos utilizar blocos sincronizados.

Antigamente o custo de se usar locks era altíssimo, hoje em dia isso custa pouco para a JVM, mas não é motivo para você sincronizar tudo sem necessidade.

## 17.6 - Um pouco mais...

1-) Você pode mudar a prioridade de cada uma de suas Threads, mas isto também é apenas uma sugestão ao escalonador.

2-) Existe um método stop nas Threads, porque não é boa prática chamá-lo?

3-) Um tópico mais avançado é a utilização de wait, notify e notifyAll para que as Threads comuniquem-se de eventos ocorridos, indicando que podem ou não podem avançar de acordo com condições

## 17.7 - Exercícios: Threads

1) Teste o exemplo deste capítulo para imprimir números em paralelo.

Escreva a classe Programa:

```
1 public class Programa implements Runnable {
2
3     private int id;
4     // colocar getter e setter pro atributo id
5
6     public void run () {
7         for (int i = 0; i < 10000; i++) {
8             System.out.println("Programa " + id + " valor: " + i);
9         }
10    }
11 }
```

Escreva a classe de Teste:

```
1 public class Teste {
2     public static void main(String[] args) {
3
4         Programa p1 = new Programa();
5         p1.setId(1);
6
7         Thread t1 = new Thread(p1);
8         t1.start();
9
10        Programa p2 = new Programa();
11        p2.setId(2);
12
13        Thread t2 = new Thread(p2);
14        t2.start();
15    }
```

```
16     }  
17 }
```

Rode várias vezes a classe Teste e observe os diferentes resultados em cada execução.

No capítulo de Sockets usaremos threads para solucionar um problema real de execuções paralelas.

## 17.8 - Desafios

- 1) Gere todos os números entre 1 e 1000 e ordene em ordem decrescente utilizando um `TreeSet`.

## E agora?

*“A primeira coisa a entender é que você não entende.”*  
– Soren Aabye Kierkegaard

Onde continuar ao terminar o ‘Java e Orientação a Objetos’.

### 18.1 - Exercício prático

A melhor maneira para fixar tudo o que foi visto nos capítulos anteriores é planejar e montar pequenos sistemas. Pense na modelagem de suas classes, como e onde usar herança, polimorfismo, encapsulamento e outros conceitos. Pratique o uso das APIs mais úteis do Java integrando-as ao seus sistemas.

### 18.2 - Certificação

Entrar em detalhes nos assuntos contidos até agora, iriam no mínimo tornar cada capítulo quatro vezes maior do que já é.

Os tópicos abordados (com a adição e remoção de alguns) constituem boa parte do que é cobrado na certificação oficial para programadores da Sun.

Para maiores informações sobre certificações, consulte a própria Sun, o [javaranch.com](http://javaranch.com) ou o [guj.com.br](http://guj.com.br), que possui diversas informações sobre o assunto. A Caelum oferece um curso de preparação para a prova de certificação como programador em Java pela Sun, o FJ-19.

### 18.3 - Web

Um dos principais focos de Java, hoje em dia, é onde a maior parte das vagas existem: programando para a web. Entram aqui tecnologias como servlets, JSPs e ferramentas famosas do mercado, como o Struts.

A Caelum oferece o curso FJ-21, onde você pode estudar os tópicos necessários para começar a trabalhar com Java na web usando as melhores práticas, design patterns e tecnologias do mercado.

### 18.4 - Revistas

Diversas revistas no Brasil e no exterior estudam o mundo java como ninguém, e podem ajudar o iniciante a conhecer muito do que está acontecendo lá fora nas aplicações comerciais.

## 18.5 - Grupos de Usuários

Diversos programadores com o mínimo ou máximo de conhecimento se reúnem online para a troca de dúvidas, informações e idéias sobre projetos, bibliotecas e muito mais. São os grupos de usuários de java.

Um dos mais importantes e famosos no Brasil é o GUJ – <http://www.guj.com.br>

## 18.6 - Falando em Java - Próximos módulos

O 'Falando em Java' não pára por aqui. A Caelum oferece uma grande variedade de cursos que você pode seguir:

**FJ-19:** Preparatório para Certificação Java

**FJ-21:** Java para desenvolvimento Web

**FJ-26:** Laboratório de MVC com JSF e Hibernate para Web

**FJ-28:** Desenvolvimento ágil para Web 2.0 com VRaptor, Hibernate e AJAX

**FJ-31:** Enterprise JavaBeans (EJB)

**FJ-55:** Java para pequenos dispositivos (Java ME)

**FJ-91:** Arquitetura e Design de Projetos Java

**PM-51:** Programação Extrema (XP) com Java

**PM-81:** Gerenciamento de Projetos de Software com Scrum

**RR-11:** Desenvolvimento ágil para Web 2.0 com Ruby on Rails

Consulte mais informações no nosso site e entre em contato conosco.

# Apêndice A - Utilizando bibliotecas e frameworks: O JFreeChart

*“Duas coisas são infinitas: o universo e a estupidez humana; e não estou certo sobre o universo.”*  
– Albert Einstein

Ao término desse capítulo você será capaz de:

- entender o conceito de **classpath**;
- adicionar jars no path do seu projeto eclipse;
- utilizar o JFreeChart.

## 19.1 - Bibliotecas e o Java

Uma das grandes vantagens frequentemente levantadas em favor do Java é o imenso número de bibliotecas de qualidade para as mais variadas necessidades. O Java padrão já é poderoso o suficiente para se fazer milhares de coisas. Mas existem muitas outras ferramentas em cima disso que facilitam ainda mais certas atividades.

E o melhor de tudo isso, é que existe uma imensa comunidade software livre por trás desses produtos. Hoje, a maioria das mais importantes ferramentas e bibliotecas Java usadas no mercado são código aberto e gratuitas. A Sun inclusive reconhece isso e apóia muito esse movimento.

## 19.2 - Como usar uma biblioteca?

Uma biblioteca de terceiros nada mais é que um conjunto de classes escritas por eles e distribuídas para nosso uso.

E o formato usado para isso é o formato padrão do Java para distribuir aplicações, o **JAR**, Java Archive, que vimos no capítulo 13.

O acesso aos recursos da biblioteca é feito através do acesso a classes e métodos públicos disponibilizados por ela. É a **API**, *Application Programming Interface*, ou seja a interface de uso daquele programa, daquela biblioteca, para que possamos integrá-la ao nosso sistema.

## 19.3 - Classpath

Quando compilamos um programa java, como ele sabe onde procurar as suas classes? Na biblioteca padrão e em alguns diretório específicos em que a JVM foi instalada!

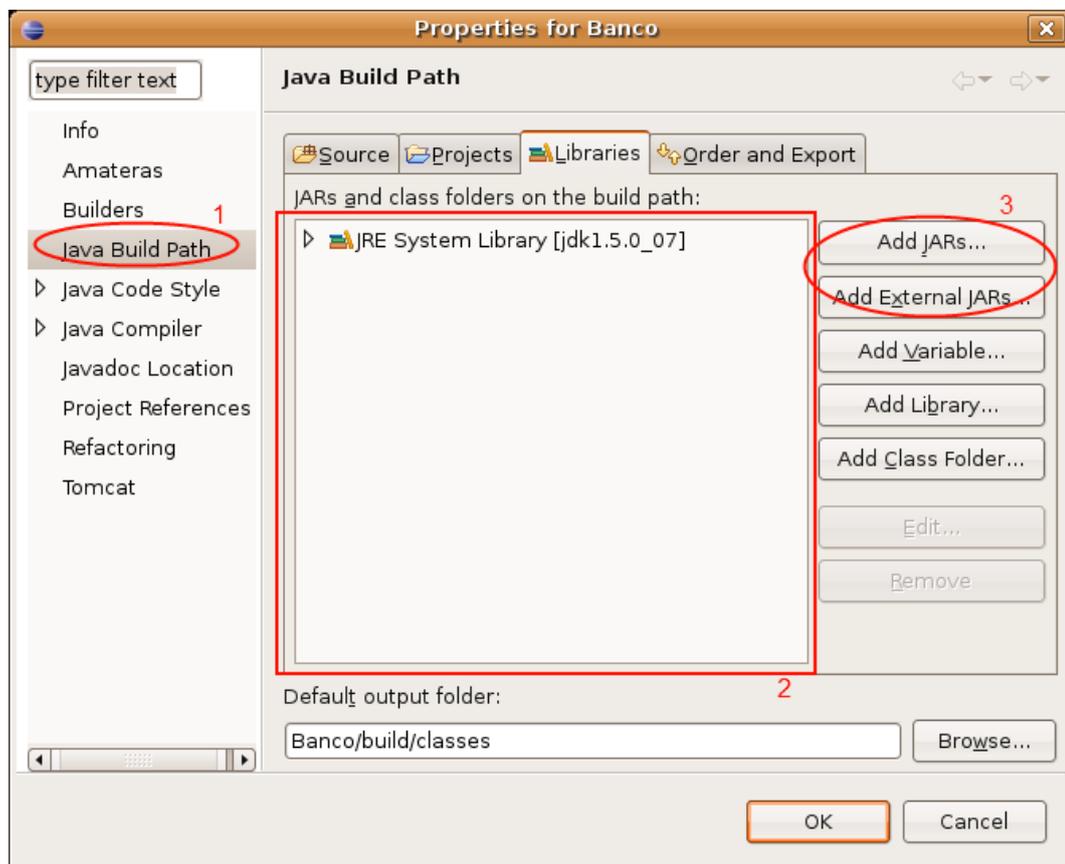
Mas, e nos projetos que usam classes diferentes das padrões? Por exemplo, as que usam jars de terceiros? É aqui que o Classpath entra história, é nele que definimos qual o caminho das nossas classes. Por isso o nome Classpath(class = classe, path = caminho).

Há algumas formas de configurarmos o classpath:

- Configurando uma variável de ambiente (**desaconselhado**);
- Passando como argumento em linha de comando (**trabalhoso**);
- Utilizando ferramentas como Ant e Maven;
- Deixando o eclipse configurar por você.

No eclipse, é muito simples:

- 1) Clique com o botão direito em cima do nome do seu projeto.
- 2) Escolha a opção Properties.
- 3) Na parte esquerda da tela abaixo, selecione a opção “Java Build Path”.



De acordo com a figura:

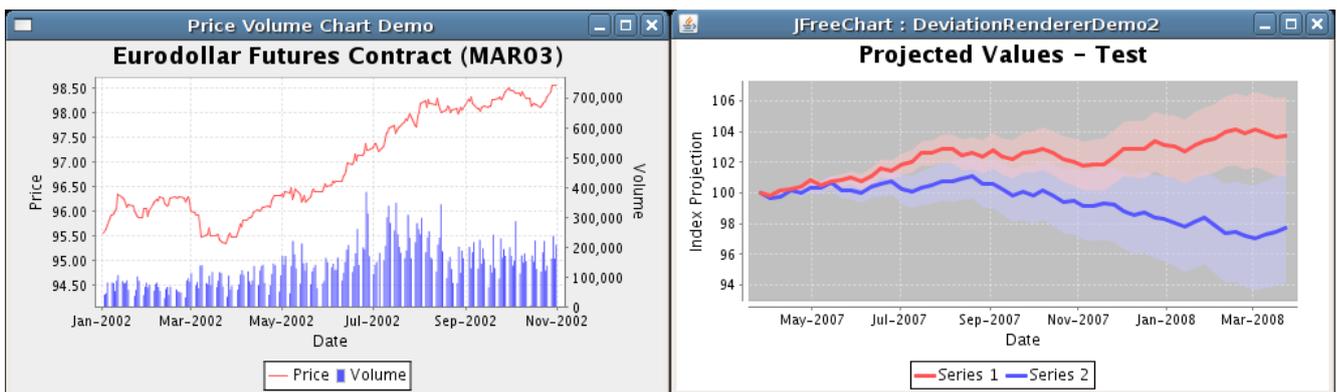
- 1) “Java Build Path” é onde você configura o classpath do seu projeto.
- 2) Lista de locais definidos que, por padrão, só vem com a máquina virtual configurada.
- 3) Opções para adicionar mais caminhos, “Add JARs...” adiciona Jar’s que estejam no seu projeto, “Add External JARs” adiciona Jar’s que estejam em qualquer outro lugar da máquina, porém guardará uma referência para aquele caminho (então seu projeto poderá não funcionar corretamente quando colocado em outro micro, mas existe como utilizar variáveis para isso).

## 19.4 - JFreeChart



O **JFreeChart** é hoje a biblioteca mais famosa para desenho de gráficos. É um projeto de software livre iniciado em 2000 e que tem ampla aceitação pelo mercado.

Além do fato de ser livre, possui a vantagem de ser bastante robusta e flexível. É possível usá-la para desenhar gráficos de pontos, de barra, de torta, de linha, gráficos financeiros, gantt charts, em 2D ou 3D e mais um monte de coisas. Consegue dar saída em JPG, PNG, SVG, EPS e exibir em componentes Swing.



Sua licença é LGPL o que permite ser usada em projetos de código fechado. O site oficial possui links para download, demos e documentação:

<http://www.jfree.org/jfreechart/>

Existe um livro oficial do JFreeChart escrito pelos desenvolvedores com exemplos e explicações detalhadas de vários gráficos diferentes. Ele é pago e pode ser obtido no Site oficial. Além disso, há muitos tutoriais gratuitos na Internet.

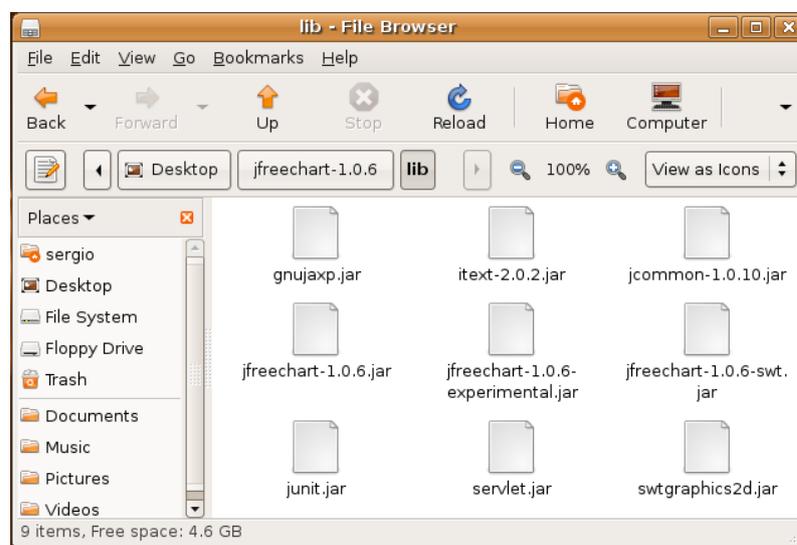
Nós vamos usar o JFreeChart para aprendermos a manipular bibliotecas externas.

## 19.5 - Preparando um projeto no Eclipse para utilizar o JFreeChart

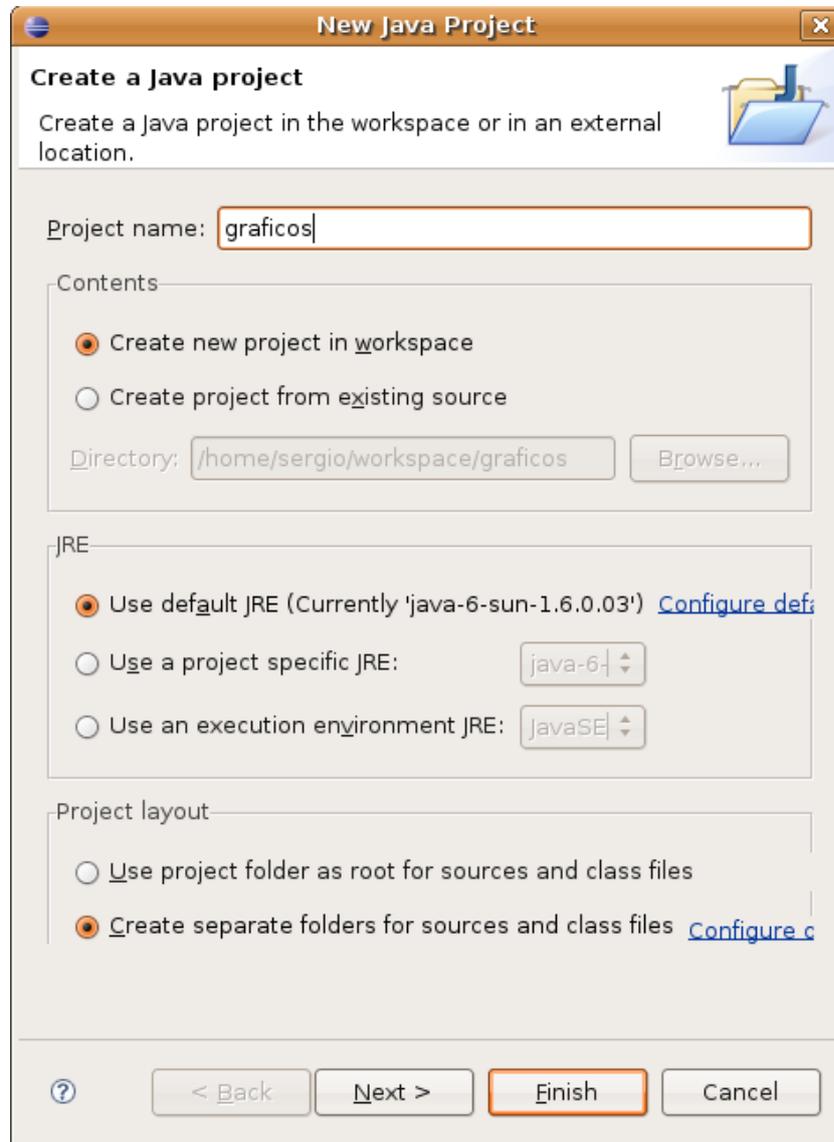
Baixe o JFreeChart em: <http://www.jfree.org/jfreechart/download.html>

Na Caelum, o arquivo encontra-se no atalho **CAELUM** no seu Desktop, dentro da pasta **11**. Basta abri-lo e mandar extrair no **Desktop**.

Abra a pasta extraída no seu Desktop. Repare que há várias coisas lá, como código fonte, scripts ant, testes, exemplos e os jars da biblioteca. No JFreeChart, os jars necessários para usarmos a biblioteca estão na pasta **lib**:



Volte no Eclipse e crie um novo projeto chamado **graficos**. Vá no menu **File, New Java Project**. Na próxima tela, coloque o nome como **graficos** e marque a opção **Create separate folders for sources and class files**. Clique em **Finish**.



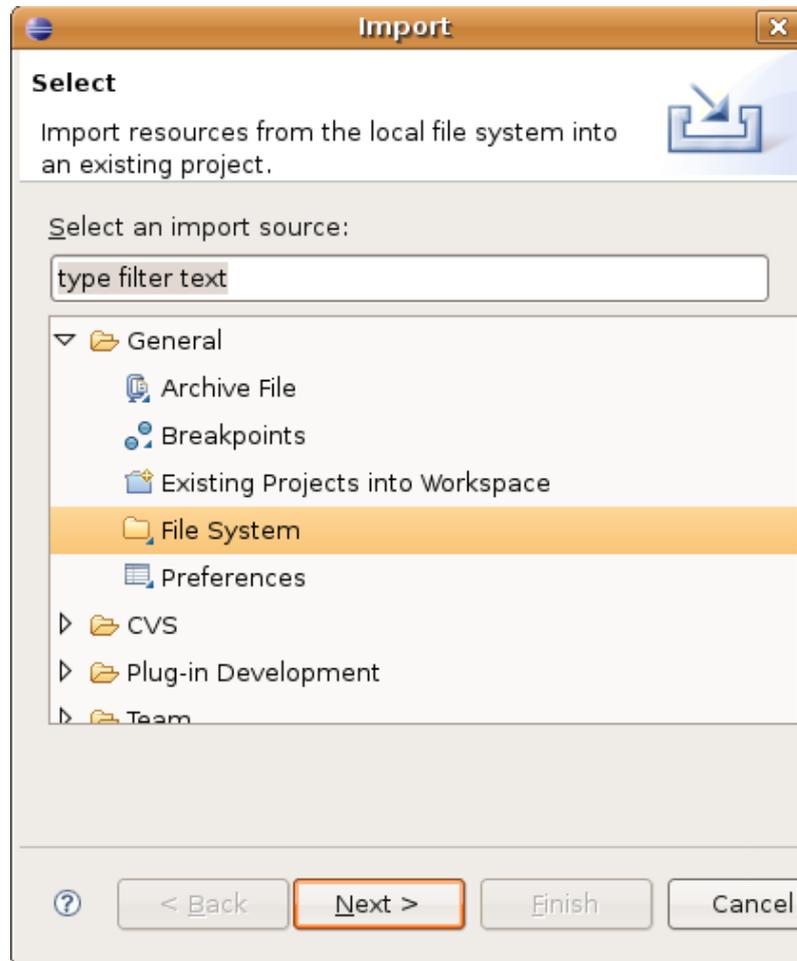
Precisamos copiar os JARs do JFreeChart para o projeto. Para organizar melhor, vamos criar uma pasta chamada **lib** para colocarmos os arquivos lá. Clique com o botão direito no projeto e vá em **New, Folder**. Dê o nome de lib e clique em ok.

#### Colocar os jars no projeto ou referenciá-los externamente?

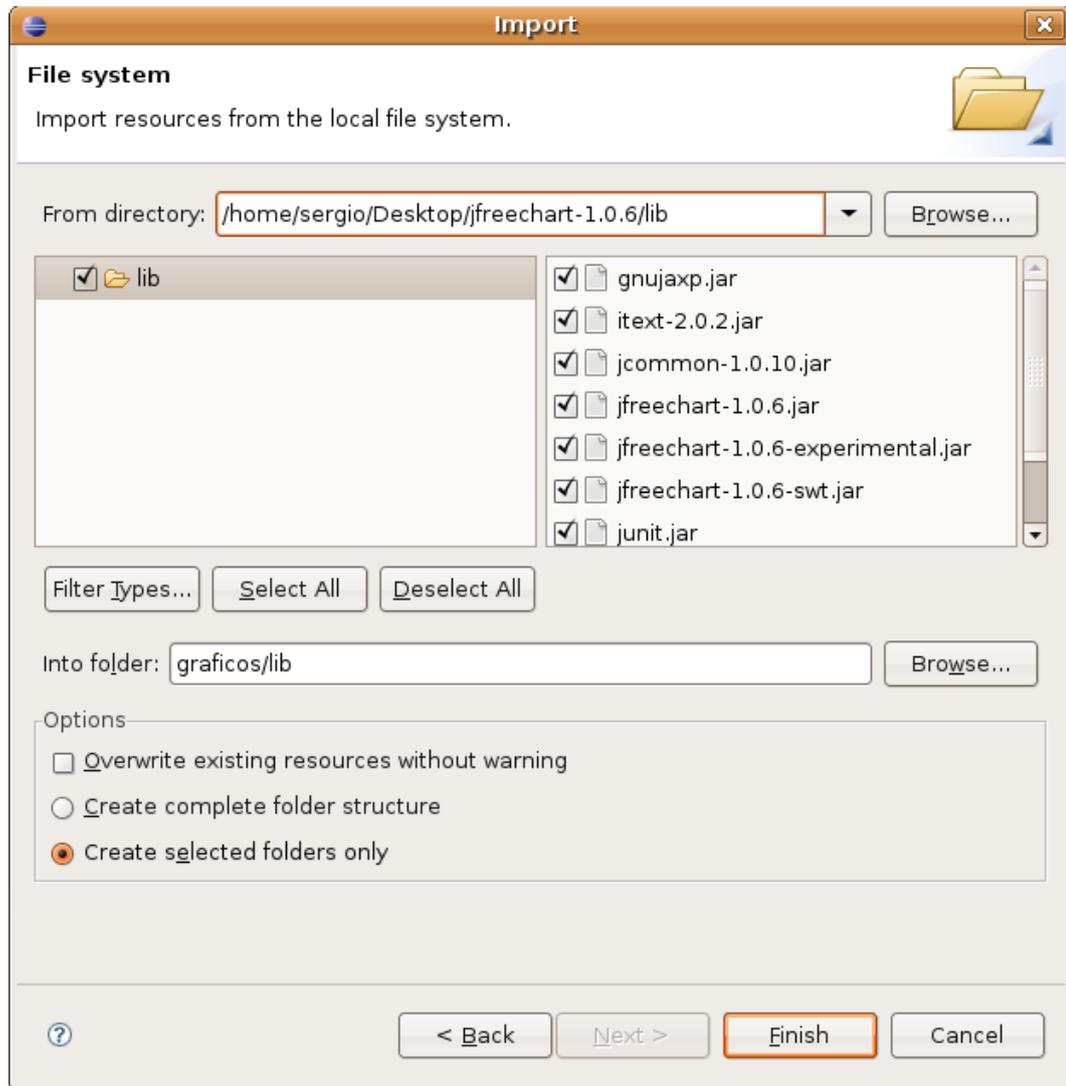
É uma boa prática sempre colocarmos os jars de dependência de nosso projeto no próprio projeto e não deixá-los em outro lugar e referenciá-los externamente.

Desta forma, cada projeto pode ter sua própria versão e evitamos problemas de compatibilidade. Além disso, nosso projeto é auto-contido e conseguimos transportá-lo facilmente para outras máquinas.

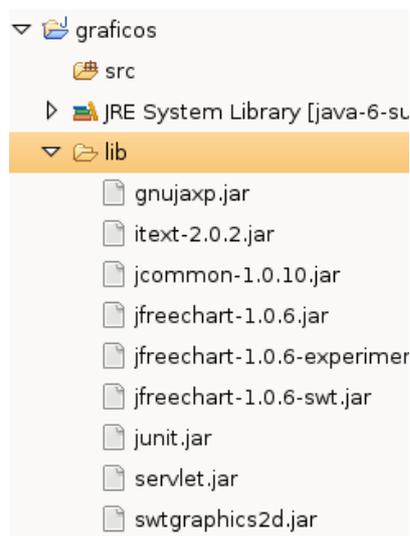
Agora precisamos copiar os jars do JFreeChart. Clique com o botão direito na pasta **lib** e vá em **Import**. Selecione **File System**:



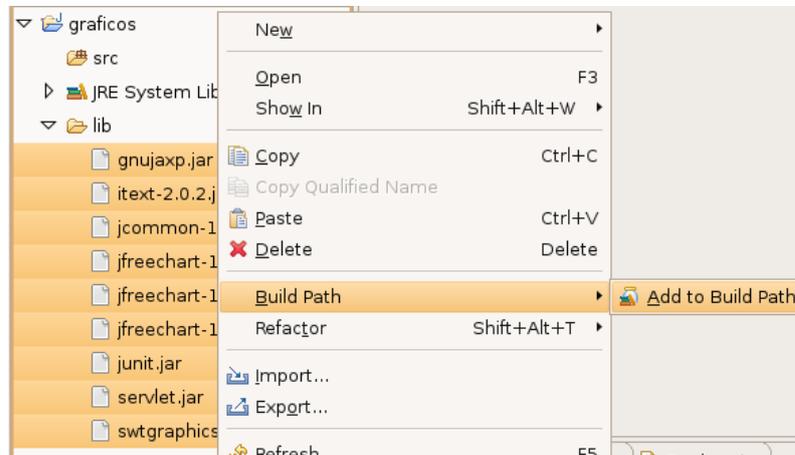
Clique em Next. Na próxima tela, indique a pasta **Desktop/jfreechart-xxx/lib** no **From directory**. Clique em **Select All** e dê Finish:



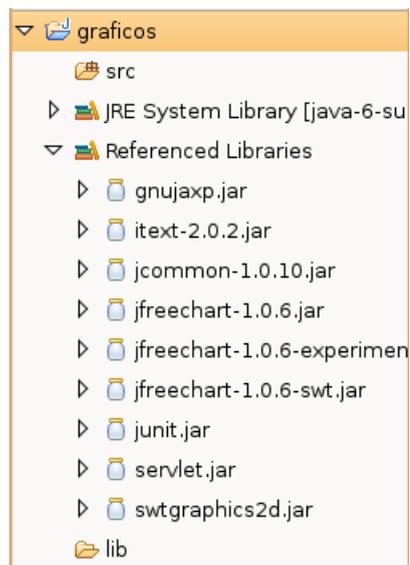
Seu projeto deve estar assim:



Selecione todos os jars dentro da pasta lib. Clique da direita, **Build Path, Add to Build Path**:



Veja que agora o eclipse mostra seus jars dentro de **Referenced Libraries** e não mais na pasta de origem (no sistema de arquivos, eles continuam no mesmo lugar, é apenas a forma do eclipse apresentar o projeto):



Estamos prontos para usar o JFreeChart agora!

## 19.6 - Utilizando o JFreeChart

Com nosso projeto todo configurado, vamos agora programar usando a API do JFreeChart. É fundamental quando programamos com alguma biblioteca, ter acesso ao Javadoc da mesma para saber quais classes e métodos usar, pra servem os parâmetros e etc.

O Javadoc do JFreeChart pode ser acessado aqui: <http://www.jfree.org/jfreechart/api/javadoc/>

A classe principal é a `org.jfree.chart.JFreeChart`, ela representa um gráfico e pode assumir vários formatos (torta, barra, pontos, linhas etc). Trabalhar diretamente com essa classe é um pouco trabalhoso; existe a `org.jfree.chart.ChartFactory` que possui uma série de métodos estáticos para facilitar nosso trabalho.

Queremos criar um gráfico de torta 3D. Consultando o Javadoc da `ChartFactory`, descobrimos o método `createPieChart3D` que devolve um objeto do tipo `JFreeChart` prontinho.

Este método recebe alguns argumentos:

- um título (`String`);
- os dados a serem exibidos (do tipo `PieDataset`);
- um `boolean` indicando se queremos legenda ou não;
- um `boolean` dizendo se queremos tooltips;
- um `boolean` para exibir URLs no gráfico ou não.

Repare que conseguimos descobrir tudo isso pelo Javadoc.

O principal aqui é o `PieDataset`, justamente o conjunto de dados que queremos exibir no gráfico. `PieDataset` é uma interface e há várias implementações: por exemplo, uma simples onde adicionamos os elementos manualmente, outra para trabalhar direto com banco de dados (JDBC) e outras.

Vamos usar a `DefaultPieDataset`, a implementação padrão e mais fácil de usar:

```
// cria o conjunto de dados
DefaultPieDataset ds = new DefaultPieDataset();
ds.setValue("legenda1", 2.3);
ds.setValue("legenda2", 1.4);
ds.setValue("legenda3", 3.5);

// cria o gráfico
JFreeChart grafico = ChartFactory.createPieChart3D("Meu Grafico", ds, true, false, false);
```

Repare que a classe `DefaultDataset` possui o método `setValue` que recebe a legenda que queremos mostrar e o valor correspondente (`double`).

Depois de criado o gráfico, queremos salvá-lo em um arquivo, enviar via rede, exibir na tela, mandar pra impressora ou qualquer coisa do gênero. A classe `org.jfree.chart.ChartUtilities` possui uma série de métodos para fazer coisas do gênero.

Talvez o método mais interessante e poderoso de `ChartUtilities` seja o `writeChartAsPNG`. Veja o que diz o javadoc dele:

Writes a chart to an output stream in PNG format.

E a lista de parâmetros ainda segundo o javadoc:

*Parameters:*

- *out* - the output stream (null not permitted).
- *chart* - the chart (null not permitted).
- *width* - the image width.
- *height* - the image height.

Ou seja: passamos o gráfico, o tamanho (altura e largura) e onde queremos que seja feita a saída dos dados do gráfico através de um `java.io.OutputStream`.

Assim o JFreeChart consegue escrever nosso gráfico em qualquer fluxo de saída de dados: seja um arquivo (`FileOutputStream`), seja enviando via rede (pela `Socket`), seja usando dinamicamente numa página web (usando `Servlets`) ou em qualquer outro lugar que suporte o padrão `OutputStream` genérico de envio de dados.

É o uso da API do `java.io` novamente que, através do polimorfismo, me garante esse poder todo sem que o JFreeChart precise saber onde exatamente o gráfico será salvo.

Vamos usar esse método para salvar em um arquivo PNG:

```
FileOutputStream fos = new FileOutputStream("grafico.png");
ChartUtilities.writeChartAsPNG(fos, grafico, 550, 400);
fos.close();
```

Agora, se juntarmos tudo, teremos um programa que gera os gráficos de torta em um arquivo.

## 19.7 - Aplicando as boas práticas de OO

O que acontecerá se precisarmos criar dois gráficos de torta diferentes? Vamos copiar e colar todo aquele código e modificar apenas as partes que mudam? E se precisarmos deixar o JFreeChart de lado e usar outra forma de gerar gráficos? Essas mudanças são fáceis se temos o código todo *espalhado* pelo nosso programa?

Os princípios de orientação a objetos e as boas práticas de programação podem nos ajudar nesses casos. Vamos **encapsular** a forma como o gráfico é criado dentro de uma classe, a `ChartBuilder`.

Essa classe deve ser capaz de gerar um gráfico de torta com os dados que quisermos e salvar o mesmo na saída que quisermos. Mas se ela vai receber os dados para gerar o gráfico, como virão esses dados? Meu programa vai passar um `PieDataset`? E no dia que precisar trocar o JFreeChart e todo o meu programa usar o `PieDataset`?

Vamos encapsular a idéia de se passar os dados para o gráfico: vamos passar um `Map<String,Double>` com a legenda que queremos mostrar e o respectivo valor. E o construtor de nossa classe `ChartBuilder` se encarrega de *traduzir* esse mapa em um `PieDataset`:

```
1 public class ChartBuilder {
2
3     private DefaultPieDataset dataSet;
4
5     /**
6      * Construtor que recebe os dados a serem exibidos
7      */
8     public ChartBuilder(Map<String,Double> mapa) {
9         // transforma o Map em PieDataset
10
11         this.dataSet = new DefaultPieDataset();
12
13         for (String chave : mapa.keySet()) {
14             this.dataSet.setValue(chave, mapa.get(chave));
15         }
16     }
17 }
```

E quem for usar essa classe, fará:

```
Map<String,Double> mapa = new HashMap<String, Double>();
mapa.put("legenda1", 2.3);
mapa.put("legenda2", 1.4);
mapa.put("legenda3", 3.5);

ChartBuilder g = new ChartBuilder(mapa);
```

Repare como esse código de teste não possui nada que o ligue ao JFreeChart especificamente. O dia que precisarmos mudar de biblioteca, precisaremos mudar apenas a classe ChartBuilder. **Encapsulamento!**

Vamos encapsular também a criação do gráfico e o salvamento em um fluxo de saída:

```
1 package br.com.caelum.fj11;
2 // muitos imports...
3
4 public class ChartBuilder {
5
6     private DefaultPieDataset dataSet;
7     private JFreeChart grafico;
8
9     public ChartBuilder(Map<String,Double> mapa) {
10         this.dataSet = new DefaultPieDataset();
11
12         for (String chave : mapa.keySet()) {
13             this.dataSet.setValue(chave, mapa.get(chave));
14         }
15     }
16
17     public void criaGrafico(String titulo) {
18         this.grafico = ChartFactory.createPieChart3D(titulo, this.dataSet, true, false, false);
19         this.grafico.getPlot().setForegroundAlpha(0.5f);
20     }
21
22     public void salvar (OutputStream out) throws IOException {
23         ChartUtilities.writeChartAsPNG(out, this.grafico, 550, 400);
24     }
25 }
```

Vamos analisar esse código em detalhes. O método `criaGrafico` recebe apenas uma String com o título do gráfico e cria o objeto JFreeChart, salvando-o no atributo `grafico`. Note na segunda linha deste método a chamada ao `setForegroundAlpha` para deixar o gráfico com uma transparência de 50%.

#### Mais métodos do JFreeChart..

A chamada ao `setForegroundAlpha` foi para mostrar como personalizar o seu gráfico. Existem centenas de outros métodos e opções que você pode usar (mudar cores, mudar fontes, ângulo e muitos outros).

Você pode testar dar um Ctrl+Espaço no Eclipse e ver os métodos disponíveis. Consulte também o javadoc.

Por fim, o método `salvar` salva o gráfico em um `OutputStream` que recebe como argumento. É boa prática

deixar nossa classe a mais genérica possível para funcionar com qualquer fluxo de saída. Quem decide onde realmente salvar (arquivo, rede etc) é quem chama a classe.

Veja como fica o programa de teste que usa essa classe:

```
Map<String,Double> mapa = new HashMap<String, Double>();
mapa.put("legenda1", 2.3);
mapa.put("legenda2", 1.4);
mapa.put("legenda3", 3.5);

ChartBuilder g = new ChartBuilder(mapa);
g.criaGrafico("Meu Grafico");
g.salvar(new FileOutputStream("grafico.png"));
```

Não usa nada específico do JFreeChart. É um código **encapsulado, flexível, pouco acoplado e elegante**: usa as boas práticas de OO.

## 19.8 - Para saber mais: Design Patterns Factory Method e Builder

Dois famosos design patterns do GoF são o **Factory Method** e o **Builder** e estamos usando ambos no nosso sistema.

Ambos são ditos **padrões de criação (creational patterns)** pois nos ajudam a criar objetos complicados.

A factory é usada pelo JFreeChart na classe `ChartFactory`. A idéia é que criar um objeto `JFreeChart` diretamente é complicado. Então criaram um *método de fábrica* que encapsula essas complicações e já me devolve o objeto prontinho para uso.

O padrão Builder estamos usando na classe `ChartBuilder` que escrevemos. Queremos encapsular a criação complicada de um gráfico e que pode mudar depois com o tempo (podemos querer usar outra API de geração de gráficos). Entra aí o *objeto construtor* da nossa classe Builder: seu único objetivo é descrever os passos para criação do nosso objeto final (o gráfico) e encapsular a complexidade disso.

Leia mais sobre esses e outros Design Patterns no livro do GoF.

## 19.9 - Exercícios: JFreeChart

Esteja certo de ter feito os passos anteriores de criação do projeto no eclipse e configuração do jfreechart antes de continuar.

1) Crie a classe `ChartBuilder` no pacote `br.com.caelum.fj11` como vimos antes.

Use os recursos do Eclipse para escrever esse código! Abuse do `Ctrl+Espaço`, do `Ctrl+1` e do `Ctrl+Shift+0`.

```
1 package br.com.caelum.fj11;
2 // imports...
3
4 public class ChartBuilder {
5
6     private DefaultPieDataset dataSet;
7     private JFreeChart grafico;
8
9     public ChartBuilder(Map<String,Double> mapa) {
```

```
10     this.dataSet = new DefaultPieDataset();
11
12     for (String chave : mapa.keySet()) {
13         this.dataSet.setValue(chave, mapa.get(chave));
14     }
15 }
16
17 public void criaGrafico(String titulo) {
18     this.grafico = ChartFactory.createPieChart3D(titulo, this.dataSet, true, false, false);
19     this.grafico.getPlot().setForegroundAlpha(0.5f);
20 }
21
22 public void salvar (OutputStream out) throws IOException {
23     ChartUtilities.writeChartAsPNG(out, this.grafico, 550, 400);
24 }
25 }
```

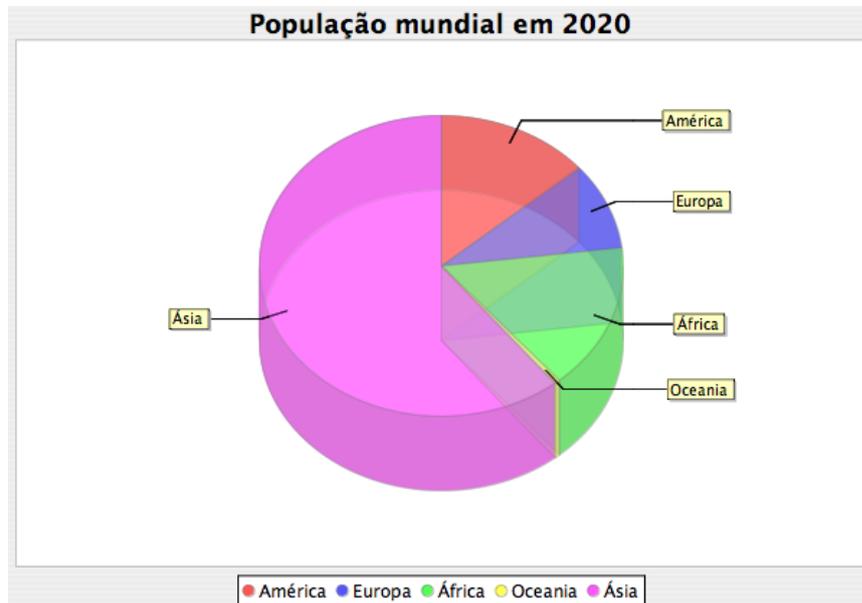
- 2) Escreva uma classe para gerar um gráfico com a projeção da população mundial para 2020 separa por continentes em bilhões de habitantes.

Faça um método main que use a ChartBuilder:

```
1 package br.com.caelum.fj11;
2 // imports...
3
4 public class GraficoPopulacao {
5     public static void main(String[] args) throws IOException, IOException {
6
7         Map<String,Double> mapa = new HashMap<String, Double>();
8         mapa.put("Ásia", 4.57);
9         mapa.put("África", 1.18);
10        mapa.put("Europa", 0.705);
11        mapa.put("América", 1.03);
12        mapa.put("Oceania", 0.038);
13
14        ChartBuilder g = new ChartBuilder(mapa);
15        g.criaGrafico("População mundial em 2020");
16        g.salvar(new FileOutputStream("grafico.png"));
17
18    }
19 }
```

(dados da wikipedia: [http://en.wikipedia.org/wiki/World\\_population](http://en.wikipedia.org/wiki/World_population))

- 3) Rode a classe GraficoPopulacao no Eclipse. Dê um F5 no nome do projeto e veja que o arquivo **grafico.png** apareceu.



## 19.10 - Exercícios adicionais

- 1) O método salvar sempre usa a dimensão 550x400 pixels. Faça um modo de se trocar o tamanho. Faça com que fique opcional, sem que, por exemplo, o usuário tenha que passar como parâmetro no método salva. Se ele não quiser, o default é 550x400.
- 2) Como fazer para mudar as cores do gráfico? Tente descobrir um jeito pela documentação do JFreeChart:  
<http://www.jfree.org/jfreechart/api/javadoc/index.html>

## 19.11 - Desafio: Fluent Interface

Dois grandes nomes da orientação a objetos, Eric Evans e Martin Fowler, criaram o termo **Fluent Interfaces** para descrever interfaces de uso de classes mais limpas e intuitivas.

Eles argumentam que frequentemente precisamos usar muitas variáveis locais quando precisamos fazer operações uma após a outra que estejam interligadas. E sugerem um modo mais elegante de se fazer isso. Veja com o exemplo.

Hoje usamos a ChartBuilder dessa forma:

```
ChartBuilder g = new ChartBuilder(mapa);  
g.criaGrafico("Meu Grafico");  
g.setTamanho(560, 410);  
g.salvar(new FileOutputStream("grafico.png"));
```

Em Fluent Interface ficaria:

```
new ChartBuilder(mapa)  
    .criaGrafico("Meu Grafico")  
    .setTamanho(600, 420)  
    .salvar(new FileOutputStream("grafico.png"));
```

Repare que chamamos os métodos de forma encadeada, sem variáveis locais envolvidas.

O desafio é implementar a classe ChartBuilder de forma a usar Fluent Interface e permitir chamadas como as feitas no código acima.

#### Ótimo artigo sobre Fluent Interface

Martin Fowler fala bastante sobre fluent interfaces nesse ótimo artigo: <http://martinfowler.com/bliki/FluentInterface.html>

---

#### Usos famosos e DSLs

Fluent interfaces são muito usadas no Hibernate, por exemplo. O jQuery, uma famosa biblioteca de efeitos javascript, popularizou-se por causa de sua fluent interfaces.

São muito usadas (e recomendadas) na construção de DSLs (Domain Specific Languages). Leia o artigo citado antes e pesquise sobre isso.

---

## Apêndice B - Sockets

*“Olho por olho, e o mundo acabará cego.”*  
– Mohandas Gandhi

Conectando-se a máquinas remotas.

### 20.1 - Motivação: uma API que usa os conceitos aprendidos

Neste capítulo, você vai conhecer a API de **Sockets** do java pelo pacote `java.net`.

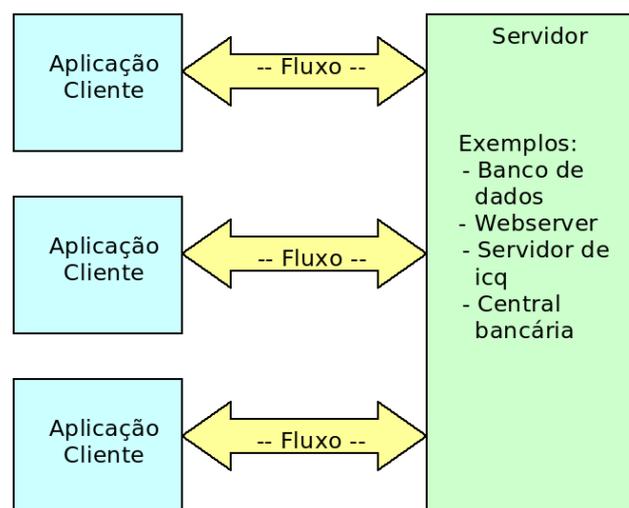
Mais útil que conhecer a API, é você perceber que estamos usando, aqui, todos os conceitos e bibliotecas aprendidas durante os outros capítulos. Repare, também, que é relativamente simples aprender a utilizar uma API, agora que temos todos os conceitos necessários para tal.

Lembre-se de fazer esse apêndice com o javadoc aberto ao seu lado.

### 20.2 - Protocolo

Da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação: o protocolo que iremos usar aqui é o **TCP** (Transmission Control Protocol).

Através do **TCP**, é possível criar um fluxo entre dois computadores - como é mostrado no diagrama abaixo:



É possível conectar mais de um cliente ao mesmo servidor, como é o caso de diversos banco de dados, webservers, etc.

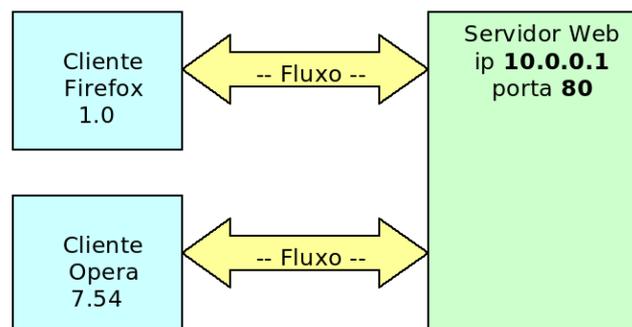
Ao escrever um programa em Java que se comunique com outra aplicação, não é necessário se preocupar com um nível tão baixo quanto o protocolo. As classes que trabalham com eles já foram disponibilizadas para serem usadas por nós no pacote `java.net`.

A vantagem de se usar TCP, em vez de criar nosso próprio protocolo de bytes, é que o TCP vai garantir a entrega dos pacotes que transferirmos e criar um protocolo base para isto é algo bem complicado.

## 20.3 - Porta

Acabamos de mencionar que diversos computadores podem se conectar a um só, mas, na realidade, é muito comum encontrar máquinas clientes com uma só conexão física. Então, como é possível se conectar a dois pontos? Como é possível ser conectado por diversos pontos?

Todas as aplicações que estão enviando e recebendo dados fazem isso através da mesma conexão física, mas o computador consegue discernir, durante a chegada de novos dados, quais informações pertencem a qual aplicação. Mas como?



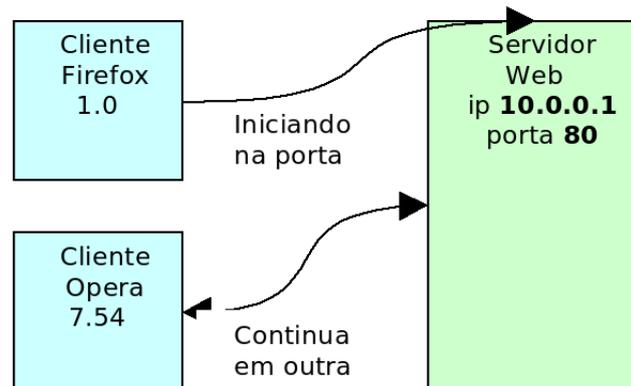
Assim como existe o **IP** para indentificar uma máquina, a **porta** é a solução para indentificar diversas aplicações em uma máquina. Esta porta é um número de 2 bytes, **varia de 0 a 65535**. Se todas as portas de uma máquina estiverem ocupadas, não é possível se conectar a ela enquanto nenhuma for liberada.

Ao configurar um servidor para rodar na porta 80 (padrão http), é possível se conectar a esse servidor através dessa porta que, junto com o ip, vai formar o endereço da aplicação. Por exemplo, o servidor web da caelum.com.br pode ser representado por: `caelum.com.br:80`

## 20.4 - Socket

Mas se um cliente se conecta a um programa rodando na porta 80 de um servidor, enquanto ele não se desconectar dessa porta, será impossível que outra pessoa se conecte?

Acontece que, ao efetuar e aceitar a conexão, o servidor redireciona o cliente de uma porta para outra, liberando novamente sua porta inicial e permitindo que outros clientes se conectem novamente.



Em Java, isso deve ser feito através de threads e o processo de aceitar a conexão deve ser rodado o mais rápido possível.

## 20.5 - Servidor

Iniciando agora um modelo de servidor de chat, o serviço do computador que funciona como base deve, primeiro, abrir uma porta e ficar ouvindo até alguém tentar se conectar.

```
1 import java.net.*;
2
3 public class Servidor {
4     public static void main(String args[]) throws IOException {
5
6         ServerSocket servidor = new ServerSocket(12345);
7         System.out.println("Porta 10001 aberta!");
8         // a continuação do servidor deve ser escrita aqui
9
10    }
11 }
```

Se o objeto for realmente criado, significa que a porta 12345 estava fechada e foi aberta. Se outro programa possui o controle desta porta neste instante, é normal que o nosso exemplo não funcione, pois ele não consegue utilizar uma porta que já está em uso.

Após abrir a porta, precisamos esperar por um cliente através do método `accept` da `ServerSocket`. Assim que um cliente se conectar, o programa continuará, por isso dizemos que esse método é *blocante*, segura a thread até que algo o notifique.

```
Socket cliente = servidor.accept();
System.out.println("Nova conexão com o cliente " +
    cliente.getInetAddress().getHostAddress()
); // imprime o ip do cliente
```

Por fim, basta ler todas as informações que o cliente nos enviar:

```
Scanner scanner = new Scanner(cliente.getInputStream());

while (scanner.hasNextLine()) {
    System.out.println(scanner.nextLine());
}
```

```
}
```

Agora fechamos as conexões, começando pelo fluxo:

```
in.close();
cliente.close();
servidor.close();
```

O resultado é a classe a seguir:

```
1 public class Servidor {
2     public static void main(String[] args) throws IOException {
3         ServerSocket servidor = new ServerSocket(12345);
4         System.out.println("Porta 12345 aberta!");
5
6         Socket cliente = servidor.accept();
7         System.out.println("Nova conexão com o cliente " +
8             cliente.getInetAddress().getHostAddress()
9         );
10
11        Scanner s = new Scanner(cliente.getInputStream());
12        while (s.hasNextLine()) {
13            System.out.println(s.nextLine());
14        }
15
16        s.close();
17        servidor.close();
18        cliente.close();
19    }
20 }
```

## 20.6 - Cliente

Agora, a nossa tarefa é criar um programa cliente que envie mensagens para o servidor... o cliente é ainda mais simples do que o servidor.

O código a seguir é a parte principal e tenta se conectar a um servidor no ip 127.0.0.1 (máquina local) e porta 12345:

```
Socket cliente = new Socket("127.0.0.1",10001);
System.out.println("O cliente se conectou ao servidor!");
```

Queremos agora ler os dados do cliente, da entrada padrão (teclado):

```
Scanner teclado = new Scanner(System.in);
while (teclado.hasNextLine()) {
    // lê a linha e faz alguma coisa com ela
}
```

Agora, basta ler as linhas que o usuário digitar através do buffer de entrada (`in`), e jogá-las no buffer de saída:

```
PrintStream saida = new PrintStream(cliente.getOutputStream());
Scanner teclado = new Scanner(System.in);
while (teclado.hasNextLine()) {
    saida.println(teclado.nextLine());
}
saida.close();
teclado.close();
```

Repare que usamos os conceito de java.io aqui novamente, para leitura do teclado e envio de mensagens para o servidor. Para as classes Scanner e PrintWriter, tanto faz de onde que se lê ou escreve os dados: o importante é que esse stream seja um InputStream / OutputStream. É o poder das interfaces, do polimorfismo, aparecendo novamente.

Nosso programa final:

```
1 public class Cliente {
2     public static void main(String[] args) throws UnknownHostException, IOException {
3         Socket cliente = new Socket("127.0.0.1", 12345);
4         System.out.println("O cliente se conectou ao servidor!");
5
6         Scanner teclado = new Scanner(System.in);
7         PrintStream saida = new PrintStream(cliente.getOutputStream());
8
9         while (teclado.hasNextLine()) {
10            saida.println(teclado.nextLine());
11        }
12
13        saida.close();
14        teclado.close();
15        cliente.close();
16    }
17 }
```

#### Cliente de Chat

- Conecta
- Lê e escreve
- Fecha a conexão

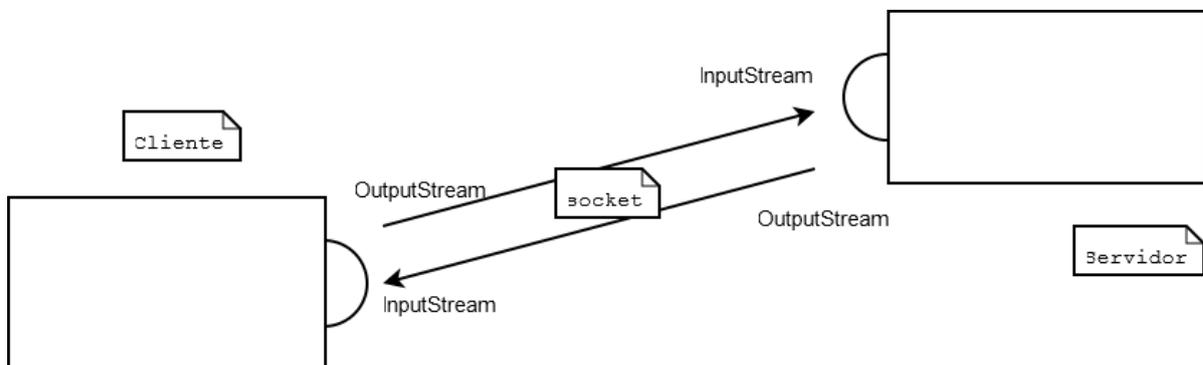
Para testar o sistema, precisamos rodar primeiro o servidor e, logo depois, o cliente. Tudo o que for digitado no cliente será enviado para o servidor.

### Multithreading

Para que o servidor seja capaz de trabalhar com dois clientes ao mesmo tempo é necessário criar uma thread logo após executar o método `accept`. A thread criada será responsável pelo tratamento dessa conexão, enquanto o loop do servidor disponibilizará a porta para uma nova conexão:

```
while (true) {  
  
    Socket cliente = servidor.accept();  
  
    // cria um objeto que irá tratar a conexão  
    TratamentoClass tratamento = new TratamentoClass(cliente);  
  
    // cria a thread em cima deste objeto  
    Thread t = new Thread(tratamento);  
  
    // inicia a thread  
    t.start();  
  
}
```

## 20.7 - Imagem geral



A socket do cliente tem um `InputStream`, que recebe do `OutputStream` do servidor, e tem um `OutputStream`, que transfere tudo para o `InputStream` do servidor. Muito parecido com um telefone!

Repare que cliente e servidor são rótulos que indicam um estado. Um micro (ou melhor, uma JVM) pode ser servidor num caso, mas pode ser cliente em outro caso.

## 20.8 - Exercícios: Sockets

1) Crie a classe `Servidor` como vimos nesse capítulo:

```
1 package br.com.caelum.chat;  
2  
3 import java.io.IOException;
```

```
4 import java.net.ServerSocket;
5 import java.net.Socket;
6 import java.util.Scanner;
7
8 public class Servidor {
9     public static void main(String[] args) throws IOException {
10         ServerSocket servidor = new ServerSocket(12345);
11         System.out.println("Porta 12345 aberta!");
12
13         Socket cliente = servidor.accept();
14         System.out.println("Nova conexão com o cliente " +
15             cliente.getInetAddress().getHostAddress()
16         );
17
18         Scanner s = new Scanner(cliente.getInputStream());
19         while (s.hasNextLine()) {
20             System.out.println(s.nextLine());
21         }
22
23         s.close();
24         servidor.close();
25         cliente.close();
26     }
27 }
```

## 2) Crie a classe Cliente como vista anteriormente:

```
1 package br.com.caelum.chat;
2
3 import java.io.IOException;
4 import java.io.PrintStream;
5 import java.net.Socket;
6 import java.net.UnknownHostException;
7 import java.util.Scanner;
8
9 public class Cliente {
10     public static void main(String[] args) throws UnknownHostException, IOException {
11         Socket cliente = new Socket("127.0.0.1", 12345);
12         System.out.println("O cliente se conectou ao servidor!");
13
14         Scanner teclado = new Scanner(System.in);
15         PrintStream saida = new PrintStream(cliente.getOutputStream());
16
17         while (teclado.hasNextLine()) {
18             saida.println(teclado.nextLine());
19         }
20
21         saida.close();
22         teclado.close();
23         cliente.close();
24     }
25 }
```

## 3) Rode a classe Servidor: repare no console do Eclipse que o programa fica esperando. Rode a classe

Cliente: a conexão deve ser feita e o Eclipse deve mostrar os dois consoles para você.

Digite mensagens no cliente e veja se elas aparecem corretamente no servidor.

- 4) Teste seu programa com um colega do curso, usando comunicação remota entre as duas máquinas. Combinem entre si quem vai rodar o cliente e quem vai rodar o servidor. Quem for rodar o cliente deve editar o ip na classe para indicar o endereço da outra máquina (verifique também se estão acessando a mesma porta).

#### Descobrimo o ip da máquina

No Windows, abra o console e digite *ipconfig* para saber qual é o seu ip. No Linux (ou no BSD, Mac, Solaris), vá no console e digite *ifconfig*.

## 20.9 - Desafio: Múltiplos Clientes

Quando o servidor aceita um cliente com a chamada ao `accept`, ele poderia chamar novamente este método para aceitar um novo cliente. E, se queremos aceitar vários clientes, simultâneos, basta chamar o `accept` várias vezes e tratar cada cliente em sua própria Thread.

Um esboço de solução para a classe Servidor:

```
ServerSocket servidor = new ServerSocket(12345);

// servidor fica eternamente aceitando clientes...
while (true) {
    Socket cliente = servidor.accept();
    // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

## 20.10 - Desafio: broadcast das mensagens

Agora que vários clientes podem mandar mensagens, gostaríamos que os clientes recebessem as mensagens enviadas pelas outras pessoas. Ao invés do servidor simplesmente escrever as mensagens no console, ele deve mandar cada mensagem para todos os clientes conectados.

Precisamos manter uma lista de clientes conectados e, quando chegar uma mensagem (de qualquer cliente), percorremos essa lista e mandamos para todos.

Use um `List` para guardar os `PrintStreams` dos clientes. Logo depois que o servidor aceitar um cliente novo, crie um `PrintStream` usando o `OutputStream` dele e adicione na lista. E, quando receber uma mensagem nova, envia para todos na lista.

Um esboço:

Adicionando na lista:

```
while (true) {
    Socket cliente = servidor.accept();
    this.lista.add(new PrintStream(cliente.getOutputStream()));

    // dispara uma Thread que trata esse cliente e já espera o próximo
}
```

```
}
```

Método que distribui as mensagens:

```
void distribuiMensagem(String msg) {  
    for (PrintStream cliente : lista) {  
        cliente.println(msg);  
    }  
}
```

Mas agora nosso cliente também recebe mensagens. Então precisamos fazer com que o Cliente, além de ler mensagens do teclado e enviar para o servidor, simultaneamente também possa receber mensagens de outros clientes enviadas pelo servidor.

Ou seja, precisamos de uma segunda Thread na classe Cliente que fica recebendo mensagens do InputStream do servidor e imprimindo no console.

Um esboço:

```
Scanner servidor = new Scanner(cliente.getInputStream());  
while (servidor.hasNextLine()) {  
    System.out.println(servidor.nextLine());  
}
```

Lembre que você precisará de no mínimo 2 threads para o cliente e 2 para o servidor. Então provavelmente você irá ter que escrever 4 classes.

Melhorias possíveis:

- Faça com o a primeira linha enviada pelo cliente seja sempre o nick dele. E quando o servidor enviar a mensagem, faça ele enviar o nick de cada cliente antes da mensagem.
- E quando um cliente desconectar? Como retirá-lo da lista?
- É difícil fazer o envio de arquivos pelo nosso sistema de chats? Sabendo que a leitura de um arquivo é feita pelo FileInputStream, seria difícil mandar esse InputStream pelo OutputStream da conexão de rede?

## 20.11 - Solução do sistema de chat

Uma solução para o sistema de chat cliente-servidor com múltiplos clientes proposto nos desafios acima. Repare que a solução não está nem um pouco elegante: o `main` já faz tudo, além de não tratarmos as exceptions. O código visa apenas mostrar o uso de uma API. É uma péssima prática colocar toda a funcionalidade do seu programa no `main` e também de jogar exceções para trás.

Nesta listagem, faltam os devidos **imports**.

Primeiro, as duas classes para o cliente. Repare que a única mudança grande é a classe nova, Recebedor:

```
1 public class Cliente {  
2     public static void main(String[] args) throws UnknownHostException, IOException {  
3         // dispara cliente  
4         new Cliente("127.0.0.1", 12345).executa();  
5     }  
}
```

```
6
7     private String host;
8     private int porta;
9
10    public Cliente (String host, int porta) {
11        this.host = host;
12        this.porta = porta;
13    }
14
15    public void executa() throws UnknownHostException, IOException {
16        Socket cliente = new Socket(this.host, this.porta);
17        System.out.println("O cliente se conectou ao servidor!");
18
19        // thread para receber mensagens do servidor
20        Recebedor r = new Recebedor(cliente.getInputStream());
21        new Thread(r).start();
22
23        // le msgs do teclado e manda pro servidor
24        Scanner teclado = new Scanner(System.in);
25        PrintStream saida = new PrintStream(cliente.getOutputStream());
26        while (teclado.hasNextLine()) {
27            saida.println(teclado.nextLine());
28        }
29
30        saida.close();
31        teclado.close();
32        cliente.close();
33    }
34 }
```

```
1 public class Recebedor implements Runnable {
2
3     private InputStream servidor;
4
5     public Recebedor(InputStream servidor) {
6         this.servidor = servidor;
7     }
8
9     public void run() {
10        // recebe msgs do servidor e imprime na tela
11        Scanner s = new Scanner(this.servidor);
12        while (s.hasNextLine()) {
13            System.out.println(s.nextLine());
14        }
15    }
16 }
```

Já o Servidor sofreu bastante modificações. A classe TrataCliente é a responsável por cuidar de cada cliente conectado no sistema:

```
1 public class Servidor {
2
3     public static void main(String[] args) throws IOException {
4         // inicia o servidor
```

```
5     new Servidor(12345).executa();
6 }
7
8 private int porta;
9 private boolean encerrar;
10 private List<PrintStream> clientes;
11
12 public Servidor (int porta) {
13     this.porta = porta;
14     this.clientes = new ArrayList<PrintStream>();
15 }
16
17 public void executa () throws IOException {
18     ServerSocket servidor = new ServerSocket(this.porta);
19     System.out.println("Porta 12345 aberta!");
20
21     while (!encerrar) {
22         // aceita um cliente
23         Socket cliente = servidor.accept();
24         System.out.println("Nova conexão com o cliente " +
25             cliente.getInetAddress().getHostAddress()
26         );
27
28         // adiciona saída do cliente à lista
29         PrintStream ps = new PrintStream(cliente.getOutputStream());
30         this.clientes.add(ps);
31
32         // cria tratador de cliente numa nova thread
33         TrataCliente tc = new TrataCliente(cliente.getInputStream(), this);
34         new Thread(tc).start();
35     }
36
37 }
38
39 public void distribuiMensagem(String msg) {
40     // envia msg para todo mundo
41     for (PrintStream cliente : this.clientes) {
42         cliente.println(msg);
43     }
44 }
45 }
```

```
1 public class TrataCliente implements Runnable {
2
3     private InputStream cliente;
4     private Servidor servidor;
5
6     public TrataCliente(InputStream cliente, Servidor servidor) {
7         this.cliente = cliente;
8         this.servidor = servidor;
9     }
10
11     public void run() {
12         // quando chegar uma msg, distribui pra todos
13         Scanner s = new Scanner(this.cliente);
```

```
14     while (s.hasNextLine()) {
15         servidor.distribuiMensagem(s.nextLine());
16     }
17     s.close();
18 }
19 }
```

## Apêndice C - Swing básico

*“Se eu enxerguei longe, foi por ter subido nos ombros de gigantes.”*

– Isaac Newton

Utilizando a API do Swing para fazer interfaces gráficas em Java.

### 21.1 - Interfaces gráficas em Java

Atualmente, o Java suporta, oficialmente, dois tipos de bibliotecas gráficas: AWT e Swing. A AWT foi a primeira API para interfaces gráficas a surgir no Java e foi, mais tarde, superada pelo Swing (a partir do Java 1.2), que possui diversos benefícios em relação a seu antecessor.

As bibliotecas gráficas são bastante simples no que diz respeito a conceitos necessários para usá-las: este curso de Java e Orientação a Objetos já capacita o aluno totalmente a estudar essas bibliotecas (e outras).

A complexidade no aprendizado de interfaces gráficas em Java reside no tamanho das bibliotecas e no enorme mundo de possibilidades; isso pode assustar, num primeiro momento.

AWT e Swing são bibliotecas gráficas oficiais inclusas em qualquer JRE ou JDK. Além destas, existem algumas outras bibliotecas de terceiros, sendo a mais famosa, o SWT - desenvolvida pela IBM e utilizada no Eclipse e em vários outros produtos.

### 21.2 - Portabilidade

As APIs de interface gráfica do Java favorecem, ao máximo, o lema de portabilidade da plataforma Java. O **look-and-feel** do Swing é único em todas as plataformas onde roda, seja ela Windows, Linux, ou qualquer outra. Isso implica que a aplicação terá exatamente a mesma interface (cores, tamanhos, etc) em qualquer sistema operacional.

Grande parte da complexidade das classes e métodos do Swing está no fato da API ter sido desenvolvida tendo em mente o máximo de portabilidade possível. Favorece-se, por exemplo, o posicionamento relativo de componentes, em detrimento do uso de posicionamento fixo, que poderia prejudicar usuários com resoluções de tela diferentes da prevista.

Com Swing, não importa qual sistema operacional, qual resolução de tela, ou qual profundidade de cores: sua aplicação se comportará da mesma forma em todos os ambientes.

### 21.3 - Começando com Swing

A biblioteca do Swing está no pacote `javax.swing` (inteira, exceto a parte de acessibilidade, que está em `javax.accessibility`).

Um primeiro exemplo com Swing pode ser a exibição de uma janela de mensagem contendo algum texto. Vamos usar para isso a classe `JOptionPane`, que possui um método estático chamado `showMessageDialog`.

```
package br.com.caelum.fj11.swing;
import javax.swing.JOptionPane;

public class OlaSwing {

    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Ola Swing!");
    }
}
```

Isso exibirá a seguinte tela:



O primeiro argumento recebido pelo método `showMessageDialog` indica qual é o componente pai (no nosso caso, `null`, porque não temos mais componentes); o segundo indica a mensagem a ser exibida. Há outro método sobrecarregado onde se pode configurar o título da janela, o tipo de mensagem ou até o ícone a ser exibido.

Nesta classe `JOptionPane`, há ainda outros métodos para exibir janelas de confirmação (Ok/Cancelar) e janelas para entrada de dados pelo usuário (input).

## 21.4 - Nosso primeiro formulário

A maioria das aplicações gráficas do mundo têm, em algum momento, algum formulário para entrada de dados. Vamos criar, então, um formulário bem simples utilizando Swing.

A API do Swing traz uma série de componentes visuais prontos para uso. São campos de texto, botões, checkboxes, labels, tabelas, árvores e muitos outros. Para começar nosso formulário, usaremos dois componentes: um campo de texto (`JTextField`) e uma etiqueta/label (`JLabel`):

```
// O JLabel recebe o texto a ser exibido
JLabel label = new JLabel("Seu nome:");

// O JTextField recebe o tamanho do campo
JTextField textField = new JTextField(20);
```

Todo componente Swing deve ser adicionado a um contêiner (Container) que administrará o agrupamento e exibição dos mesmos. Usaremos o container mais comum, um `JPanel` (algo como um painel de componentes). Através de seu método `add`, conseguimos adicionar nossos componentes:

```
// Cria um JPanel (container)
JPanel panel = new JPanel();
```

```
// adiciona os componentes
panel.add(label);
panel.add(textField);
```

Por último, para exibirmos nosso formulário simples, precisamos colocar nosso `Jpanel` em uma janela. Usaremos a classe `JFrame`, que representa uma janela simples.

```
// Criamos um JFrame passando o título da janela
JFrame frame = new JFrame("Meu primeiro formulário");

// Adicionamos nosso JPanel
frame.add(panel);

// Preparamos o JFrame para exibição
frame.pack();
frame.setVisible(true);
```

O método `pack()` de `JFrame`, chamado acima, serve para redimensionar nosso frame para um tamanho adequado baseado nos componentes que ele tem. E o `setVisible` recebe um `boolean` indicando se queremos que a janela seja visível ou não.

Vamos apenas adicionar um último comando, que indica ao nosso frame que a aplicação deve ser terminada quando o usuário fechar a janela.

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

No exemplo completo abaixo, colocamos todas as variáveis como atributos e criamos um método chamado `montaFormulario` com todo o código explicado antes. No método `main`, criamos um novo `Formulario` e chamamos o método `montaFormulario()`:

```
package br.com.caelum.fj11.swing;
import javax.swing.*;

public class Formulario {
    private JLabel label;
    private JTextField textField;

    private JPanel panel;
    private JFrame frame;

    private void montaFormulario() {
        label = new JLabel("Seu nome:");
        textField = new JTextField(20);

        panel = new JPanel();
        panel.add(label);
        panel.add(textField);

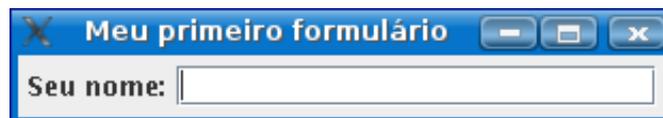
        frame = new JFrame("Meu primeiro formulário");
        frame.add(panel);

        frame.pack();
        frame.setVisible(true);
    }
}
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        new Formulario().montaFormulario();
    }
}
```

Ao rodar este programa, teremos a seguinte tela para digitação do nome:



## 21.5 - Adicionando eventos

O formulário anterior é até que interessante para começar a aprender Swing, mas ele é totalmente inútil, não faz nada. Queremos ser capazes de recuperar o valor digitado pelo usuário para efetuar alguma operação (salvar no banco de dados, exibir, enviar via rede, mandar um email, etc).

Para fazermos efetivamente alguma coisa, trabalhamos com eventos que são disparados pelo usuário. O Swing possui uma forma muito elegante de trabalhar com eventos - através de interfaces. O usuário pode disparar eventos ao digitar, ao clicar, ao passar o mouse, e muitas outras situações.

No nosso formulário, usaremos um botão (componente) que, quando clicado pelo usuário (evento), disparará um método (tratador/*handler*). Neste método, vamos recuperar o texto digitado pelo usuário e efetuar alguma ação.

O componente Swing que representa um botão é o `JButton`. Precisamos criar um botão e colocá-lo no nosso container (o `JPanel`):

```
// cria o JButton passando o texto do botao
JButton button = new JButton("Exibir");

// adiciona o botao ao JPanel
panel.add(button);
```

Isso acrescentará o botão ao formulário. Mas como disparar um método quando o botão for clicado? O Swing nos traz o conceito de **Listeners (ouvintes)**, que são interfaces que implementamos com métodos para serem disparados por eventos.

No nosso caso, para fazer um método disparar ao clique do botão, usamos a interface `ActionListener`. Essa interface nos dá um método `actionPerformed`:

```
public void actionPerformed(ActionEvent e) {
    // implementação aqui...
}
```

Vamos fazer, então, nossa própria classe formulário implementar essa interface e esse método. Queremos, quando o botão for clicado, pegar o texto digitado pelo usuário e exibir na tela (vamos usar o `JOptionPane` para isso). Nosso método, `actionPerformed` fica desta forma:

```
public void actionPerformed(ActionEvent e) {
    // pega o texto do JTextField
    String texto = textField.getText();

    // exibe usando o JOptionPane
    JOptionPane.showMessageDialog(frame, texto);
    // aqui usamos o frame como componente pai do messageDialog
}
```

O último detalhe que falta, é indicar que essa ação (esse `ActionListener`) deve ser disparado quando o botão for clicado. Fazemos isso através do método `addActionListener`, chamado no botão. Ele recebe como argumento um objeto que implementa `ActionListener` (no nosso caso, o próprio `Formulario`, o `this`):

```
button.addActionListener(this);
```

O código final de nosso formulário deve estar assim:

```
public class Formulario implements ActionListener {
    private JLabel label;
    private JTextField textField;
    private JButton button;

    private JPanel panel;
    private JFrame frame;

    private void montaFormulario() {
        label = new JLabel("Seu nome:");
        textField = new JTextField(20);
        button = new JButton("Exibir");

        button.addActionListener(this);

        panel = new JPanel();
        panel.add(label);
        panel.add(textField);
        panel.add(button);

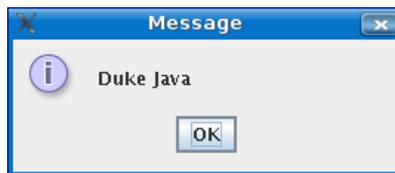
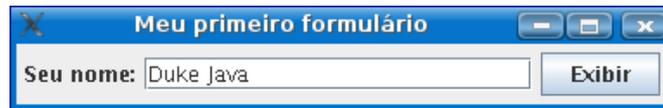
        frame = new JFrame("Meu primeiro formulario");
        frame.add(panel);

        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void actionPerformed(ActionEvent e) {
        String texto = textField.getText();
        JOptionPane.showMessageDialog(frame, texto);
    }

    public static void main(String[] args) {
        new Formulario().montaFormulario();
    }
}
```

Ao rodar esse programa, você verá nosso formulário com um label, um campo de texto e um botão. Depois de digitar algo e clicar no botão, você verá uma mensagem com o texto do campo preenchido:



## 21.6 - Gerenciadores de Layout

Quando adicionamos novos componentes, como o Java sabe onde posicioná-los? Por que sempre são adicionados do lado direito? Se redimensionamos a tela (teste) os elementos “fluem” para a linha de baixo. Por quê?

Essas e outras perguntas são respondidas pelo **Layout Manager**, o gerenciador de layout do Swing. O Java vem com uma série de **Layouts** diferentes, que determinam como os elementos serão dispostos na tela, seus tamanhos preferenciais, como eles se comportarão quando a janela for redimensionada e muitos outros aspectos.

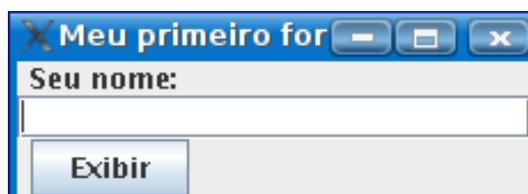
Ao escrever uma aplicação Swing você deve indicar qual Layout Manager você deseja utilizar. Por padrão, é utilizado o **FlowLayout** que especifica justamente que os elementos devem ser justapostos, que eles devem “fluir” um para baixo do outro quando a tela for redimensionada e etc.

Poderíamos usar um outro Layout Manager como o **GridLayout**, por exemplo. Nossa aplicação ficaria da seguinte forma:



Note como os elementos parecem estar dispostos em uma grade (um grid). Ao redimensionar essa tela, por exemplo, os elementos não fluem como antes; eles são redimensionados para se adaptarem ao novo tamanho do grid.

Ou ainda, usando o **BoxLayout** pelo eixo y:



Há uma série de Layout Managers disponíveis no Java, cada um com seu comportamento específico. Há inclusive Layout Managers de terceiros (não-oficiais do Java) que você pode baixar; o projeto `Jgoodies`, por exemplo, tem um excelente Layout Manager otimizado para trabalhar com formulários, o `FormLayout`.

Para saber mais sobre Layout Managers, quais são e como usar cada um deles, consulte a documentação do Swing.

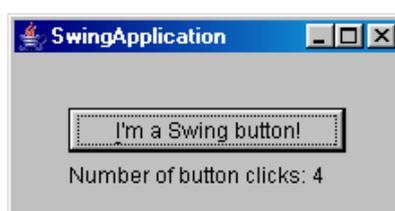
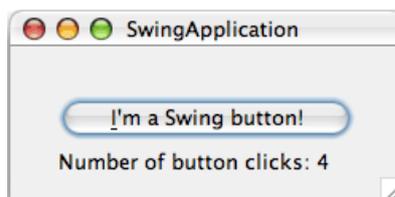
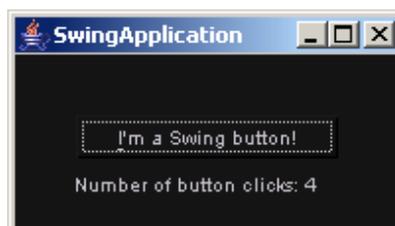
## 21.7 - Look And Feel

**Look-and-Feel** (ou LaF) é o nome que se dá à “cara” da aplicação (suas cores, formatos e etc). Por padrão, o Java vem com um look-and-feel próprio, que se comporta exatamente da mesma forma em todas as plataformas suportadas.

Mas às vezes esse não é o resultado desejado. Quando rodamos nossa aplicação no Windows, por exemplo, é bastante gritante a diferença em relação ao visual das aplicações nativas. Por isso é possível alterar qual o look-and-feel a ser usado em nossa aplicação.

Além do padrão do Java, o JRE 5 da Sun ainda traz LaF nativos para Windows e Mac OS, além do Motif e GTK. E, fora esses, você ainda pode baixar diversos LaF na Internet ou até desenvolver o seu próprio.

Veja esses screenshots da documentação do Swing mostrando a mesma aplicação rodando com 4 LaF diferentes:



## 21.8 - Para saber mais

- 1) Consultar o javadoc do Swing pode não ser muito simples. Por isso, a Sun disponibiliza um ótimo tutorial online sobre Swing em seu Site:

<http://java.sun.com/docs/books/tutorial/uiswing/>

- 2) Existem alguns bons editores visuais (Drag-and-Drop) para se trabalhar com Swing, entre produtos comerciais e livres. Destaque para:
  - Matisse, que vem embutido no Netbeans e é considerado, hoje, o melhor editor
  - VEP (Visual Editor Plugin), um plugin que pode ser instalado no Eclipse
- 3) Aplicações grandes com Swing podem ganhar uma complexidade enorme e ficar difíceis de manter. Alguns projetos tentam minimizar esses problemas; há, por exemplo, o famoso projeto Thinlet, onde você pode utilizar Swing escrevendo suas interfaces gráficas em XML.

# Apêndice D - Outros conceitos e APIs importantes

*“Primeiro aprenda ciência da computação e toda a teoria. Depois desenvolva um estilo de programação. E aí esqueça tudo e apenas ‘hackeie’.”*  
– George Carrette

Veremos, agora, diversos pequenos detalhes que não vimos no decorrer do curso, mas que são importantes no dia a dia.

## 22.1 - Import Estático

Algumas vezes, escrevemos classes que contêm muitos métodos e atributos estáticos (finais, como constantes). Essas classes são classes utilitárias e precisamos sempre nos referir a elas antes de chamar um método ou utilizar um atributo:

```
import pacote.ClasseComMetodosEstaticos;
class UsandoMetodosEstaticos {
    void metodo() {
        ClasseComMetodosEstaticos.metodo1();
        ClasseComMetodosEstaticos.metodo2();
    }
}
```

Começa a ficar muito chato escrever, toda hora, o nome da classe. Para resolver esse problema, no Java 5.0 foi introduzido o `static import`, que importa métodos e atributos estáticos de qualquer classe. Usando essa nova técnica, você pode importar os métodos do exemplo anterior e usá-los diretamente:

```
import static pacote.ClasseComMetodosEstaticos.*;
class UsandoMetodosEstaticos {
    void metodo() {
        metodo1();
        metodo2();
    }
}
```

Apesar de você ter importado todos os métodos e atributos estáticos da classe `ClasseComMetodosEstaticos`, a classe em si não foi importada e, se você tentasse dar `new`, por exemplo, ele não conseguiria encontrá-la, precisando de um `import` normal à parte.

Um bom exemplo de uso são os métodos e atributos estáticos da classe de matemática do Java.

```
import static java.lang.Math.*;

class TesteMatematico {
    double areaDaCircunferencia (double raio) {
        return PI * raio * raio; // usamos PI ao invés de Math.PI !!
    }
}
```

## 22.2 - final

A palavra chave `final` tem várias utilidades. Numa classe, define que a classe nunca poderá ter uma filha, isso é, não pode ser estendida. A classe `String`, por exemplo, é `final`.

Como modificador de método, `final` indica que aquele método não pode ser reescrito.

Ao usarmos como modificador de declaração de variável, indica que o valor daquela variável nunca poderá ser alterado, uma vez atribuído. Se a variável for um atributo, você tem que inicializar seu valor durante a construção do objeto - caso contrário, ocorre um erro de compilação, pois atributos `final` não são inicializados com valores default).

Imagine que, quando criamos um objeto `Cliente`, não queremos que seu `rg` seja modificado:

```
class Cliente {

    private final int rg;

    public Cliente(int rg) {
        this.rg = rg;
    }
}
```

Uma variável `static final` tem uma cara de constante daquela classe e, se for `public static final`, aí parece uma constante global! Por exemplo, na classe `Collections` do `java.util` você tem uma variável `public static final` chamada `EMPTY_LIST`. É convenção essas variáveis terem letras maiúsculas e separadas por *underscore* em vez de subir e descer.

Isso é muito utilizado, mas hoje no java 5 para criarmos constantes, é muito mais interessante utilizarmos o recurso de enumerações que, além de tipadas, já possuem diversos métodos auxiliares.

## 22.3 - Calendar

A classe abstrata `Calendar` encapsula um momento no tempo representado em milissegundos. Também provê métodos para manipulação desse momento.

A subclasse concreta de `Calendar` mais usada é a `GregorianCalendar`, que representa o calendário usado pela maior parte dos países. (outras implementações existem, como a do calendário budista `BuddhistCalendar`, mas estas são internas e devolvidas de acordo com seu `Locale`)

Para obter um `Calendar` que encapsula o instante atual (data e hora), usamos o método estático `getInstance()` de `Calendar` (veja o próximo exemplo).

A partir de um `Calendar`, podemos saber o valor de seus campos, como ano, mês, dia, hora, minuto... Para isso, usamos o método `get` que recebe um inteiro representando o campo; os valores possíveis estão em constantes na classe `Calendar`.

No exemplo abaixo, imprimimos o dia de hoje e o dia da semana correspondente. Note que o dia da semana devolvido é um inteiro que representa o dia da semana (`Calendar.MONDAY` etc):

```
Calendar c = Calendar.getInstance();
System.out.println("Dia do Mês: " + c.get(Calendar.DAY_OF_MONTH));
System.out.println("Dia da Semana: " + c.get(Calendar.DAY_OF_WEEK));
```

Um possível resultado é:

```
Dia do Mês:4
Dia da Semana: 5
```

No exemplo acima, o dia da semana 5 representa a quinta-feira.

Da mesma forma que podemos pegar os valores dos campos, podemos atribuir novos valores a esses campos por meio dos métodos `set`.

Há diversos métodos `set` em `Calendar`. O mais geral é o que recebe dois argumentos: o primeiro indica qual é o campo (usando aquelas constantes de `Calendar`) e, o segundo, o novo valor. Além desse método, outros métodos `set` recebem valores de determinados campos; o `set` de três argumentos, por exemplo, recebe ano, mês e dia. Vejamos um exemplo de como alterar a data de hoje:

```
Calendar c = Calendar.getInstance();
c.set(Calendar.HOUR, 10); // fazemos hora valer 10
c.set(Calendar.MINUTE, 30); // fazemos minuto valer 30
c.set(2005, 11, 25); // mudamos a data para o Natal, mês começa do 0
```

Outro método bastante usado é `add`, que adiciona uma certa quantidade a qualquer campo do `Calendar`. Por exemplo, para adicionar um ano à data de hoje:

```
Calendar c = Calendar.getInstance();
c.add(Calendar.YEAR, 1); // adiciona 1 ao ano
```

Note que, embora o método se chame `add`, você pode usá-lo para subtrair valores também; basta colocar uma quantidade negativa no segundo argumento!

Os métodos `after` e `before` são usados para comparar o objeto `Calendar` em questão a outro `Calendar`. O método `after` devolverá `true` quando o `Calendar` em questão estiver num momento no tempo maior que o do `Calendar` passado como argumento. Por exemplo, `after` devolverá `false` se compararmos o dia das crianças com o Natal, pois o dia das crianças não vem depois do Natal:

```
Calendar c1 = new GregorianCalendar(2005, Calendar.OCTOBER, 12);
Calendar c2 = new GregorianCalendar(2005, Calendar.DECEMBER, 25);
System.out.println(c1.after(c2));
```

Analogamente, o método `before` verifica se o momento em questão vem antes do momento do `Calendar` que foi passado como argumento. No exemplo acima, `c1.before(c2)` devolverá `true`, pois o dia das crianças vem antes do Natal.

Note que `Calendar` implementa `Comparable`. Isso quer dizer que você pode usar o método `compareTo` para comparar dois calendários. No fundo, `after` e `before` usam o `compareTo` para dar suas respostas.

Por último, um dos problemas mais comuns quando lidamos com datas é verificar o intervalo entre duas datas. O método abaixo devolve o número de dias entre dois objetos `Calendar`. O cálculo é feito pegando a diferença entre as datas em milissegundos e dividindo esse valor pelo número de milissegundos em um dia:

```
public int diferencaEmDias(Calendar c1, Calendar c2) {  
    long m1 = c1.getTimeInMillis();  
    long m2 = c2.getTimeInMillis();  
    return (int) ((m2 - m1) / (24*60*60*1000));  
}
```

## 22.4 - Date

A classe `Date` não é recomendada porque a maior parte de seus métodos estão marcados como `deprecated`, porém ela tem amplo uso legado nas bibliotecas do java. Ela foi substituída no java 1.1 pelo `Calendar`, para haver suporte correto a internacionalização do sistema de datas.

Você pode pegar um `Date` de um `Calendar` e vice-versa através dos getters e setters de `time`:

```
Calendar c = new GregorianCalendar(2005, Calendar.OCTOBER, 12);  
Date d = c.getTime();  
c.setTime(d);
```

Isso faz com que você possa operar com datas da maneira nova, mesmo que as APIs ainda usem objetos do tipo `Date` (como é o caso de `java.sql`).

## 22.5 - DateFormat

A classe `java.text.DateFormat` permite converter Strings de inputs do usuário para Dates, seguindo um determinado formato. Serve também para o caminho contrário: pegar uma data e gerar uma string de saída em um certo formato.

A `SimpleDateFormat` é a forma mais fácil de fazer tudo isso:

```
// usuário digita uma String  
String s = "21/02/1999";  
  
// date format  
DateFormat df = new SimpleDateFormat("dd/MM/yyyy");  
  
// converte para Date  
Date data = df.parse(s);  
  
// formata para String de novo  
String formatada = df.format(data);
```

## 22.6 - Outras classes muito úteis

Na edição 13 da revista MundoJava, vimos diversas classes utilitárias - algumas são muito importantes e você deve conhecê-las:

- `Random` – para gerar números pseudo-aleatórios
- `Formatter` – um formatador que recebe argumentos parecidos com o `printf` do C, tanto que agora a `PrintStream` (por exemplo, o `System.out`) possui um método com este nome.
- `Scanner` – já vimos um pouco dele aqui, mas ele é muito mais poderoso, possibilitando a utilização de expressões regulares, facilitando muito a leitura.
- `ResourceBundle` – para internacionalização.
- `java.io.File` – manipula diretórios, nomes de arquivos, verifica tamanhos e propriedades deles, deleta, move, etc.

## 22.7 - Anotações

Anotação é a maneira de se escrever metadados no java 5.0. Algumas anotações podem ser retidas (`retained`) no `.class`, fazendo com que, por *reflections*, nós possamos descobrir essas informações.

É utilizada, por exemplo, para indicar que determinada classe deve ser processada por um framework de uma certa maneira, facilitando, assim, as clássicas configurações através de centenas de linhas de XML.

Apesar dessa propriedade interessante, algumas anotações servem apenas para indicar algo ao compilador. `@Override` é o exemplo disso. Caso você use essa anotação em um método que não foi reescrito, vai haver um erro de compilação! A vantagem de usá-la é apenas para facilitar a legibilidade.

`@Deprecated` indica que um método não deve ser mais utilizado por algum motivo e decidiram não retirá-lo da API para não quebrar programas que já funcionavam anteriormente.

`@SuppressWarnings` indica para o compilador que ele não deve dar warnings a respeito de determinado problema, indicando que o programador sabe o que está fazendo. Um exemplo é o warning que o compilador do Eclipse dá quando você não usa determinada variável. Você vai ver que um dos quick fixes é a sugestão de usar o `@SuppressWarnings`.

Anotações podem receber parâmetros. Existem muitas delas na API do java 5, mas realmente é ainda mais utilizada em alguns frameworks, como o hibernate 3, o ejb 3 e o Junit4.

# Apêndice E - Instalação do Java

*“Quem pouco pensa, engana-se muito.”*  
– Leonardo da Vinci

Instalação do Java Development Kit em ambiente Windows e Linux.

## 23.1 - Escolhendo a VM

Como vimos antes, a VM é apenas uma especificação e devemos baixar uma implementação. Há muitas empresas que implementam uma VM, como a própria Sun, a IBM, a BEA e outras.

A da Sun é a mais usada e possui versões para Windows, Linux e Solaris. Você pode baixar o SDK deles acessando:

<http://java.sun.com/javase/downloads/index.jsp>

Nesta página, você tem a opção de baixar o JDK, JRE, JDK+Java EE, Documentação, Código Fonte e outras coisas.

## 23.2 - Instalando no Ubuntu e outros Linux

Cada distribuição Linux tem sua própria forma de instalação. Algumas já trazem o Java junto, outras possibilitam que você instale pelos repositórios oficiais e em alguns casos você precisa baixar direto da Sun e configurar tudo manualmente.

No Ubuntu, a distribuição usada na Caelum e a mais usada no mundo, a instalação é bastante simples. Basta ir no terminal e digitar:

```
sudo apt-get install sun-java6-jdk
```

Há ainda outros pacotes disponíveis para o plugin no navegador (sun-java6-plugin), a documentação (sun-java6-doc), o código fonte (sun-java6-source) ou apenas o jre (sun-java6-jre).

Se você usa outra distribuição que não o Ubuntu, consulte a documentação da mesma para saber como instalar o Java.

## 23.3 - No Mac OS X e Solaris

O Mac OS X já traz o Java instalado junto com o sistema operacional. É uma VM desenvolvida pela própria Apple e baseada no código da Sun. Atualizações para a VM devem ser obtidas no site da Apple em:

<http://developer.apple.com/java/>

O Solaris, por ser da própria Sun, também traz o Java instalado já de fábrica. É a própria VM da empresa, que pode ser baixada também no site oficial:

<http://java.sun.com/javase/downloads/>

## 23.4 - Instalação do JDK em ambiente Windows

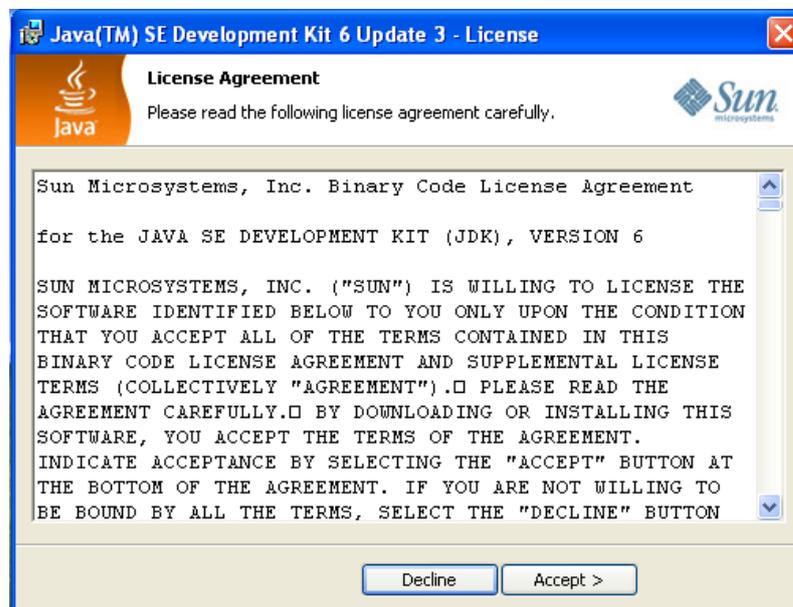
Para instalar o JDK no Windows, primeiro baixe-o no site da Sun. É um simples arquivo executável que contém o Wizard de instalação.

### Instalação

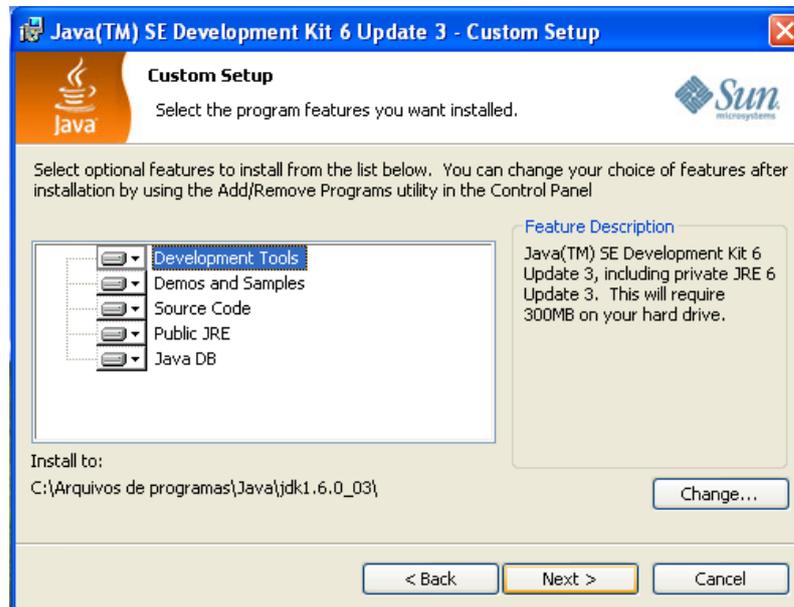
- 1) Dê um clique duplo no arquivo “jdk-<versão>-windows-i586-p.exe” e espere até ele entrar no wizard de instalação.



- 2) Nesta tela, aceite o contrato da Sun, clicando em *Accept*:

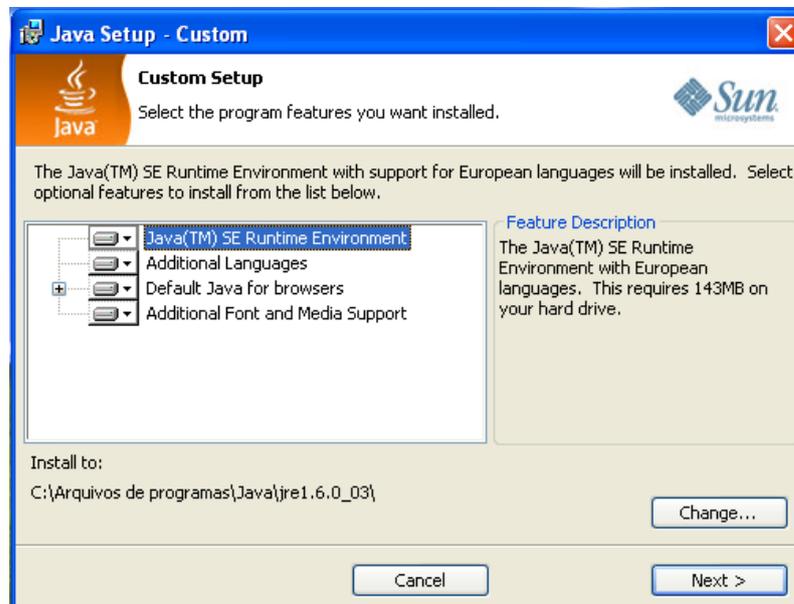


- 3) Agora, devemos selecionar quais recursos instalaremos junto com o java (ferramentas de desenvolvimento, demonstrações, o código fonte e o próprio java) e onde ele será instalado (guarde esse caminho porque usaremos ele mais pra frente). Deixe como está e clique em “Next”.



4) Pronto, agora ele instalará o JDK, apenas aguarde.

5) Agora ele começará a instalar o JRE (Java Runtime Environment), se você já não tiver instalado. Assim como o JDK, ele também tem algumas opções. Deixe como está e clique em “Next”.



6) Aguarde até o fim da instalação e clique em Finish.

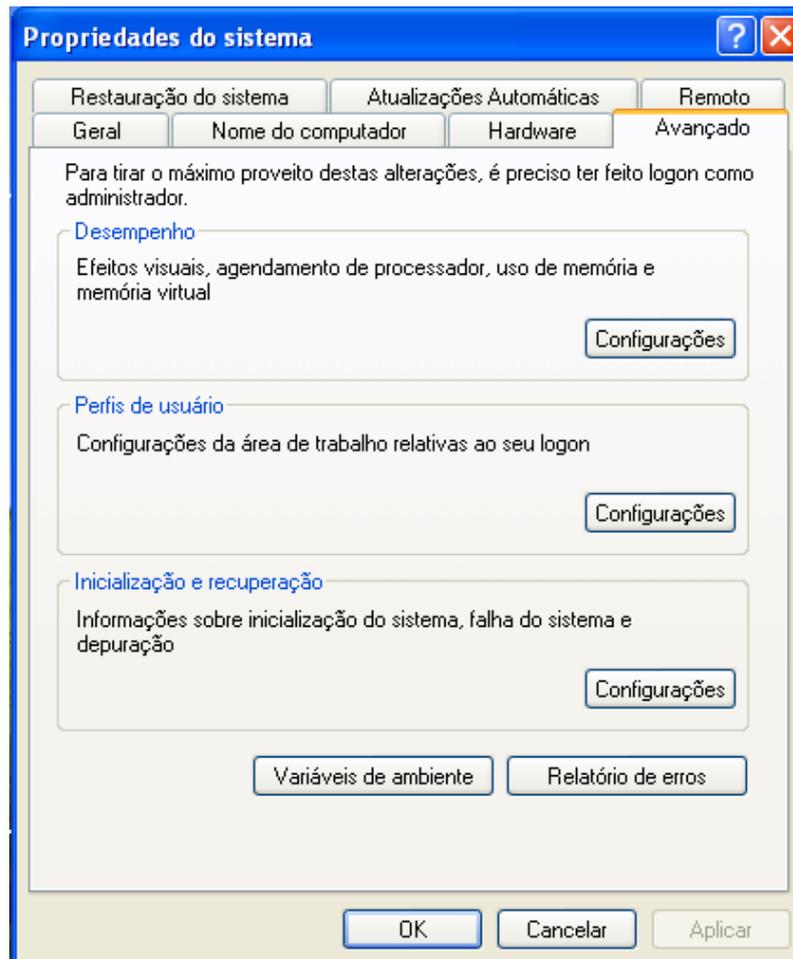
## Configurando

Precisamos configurar algumas variáveis de ambiente após a instalação.

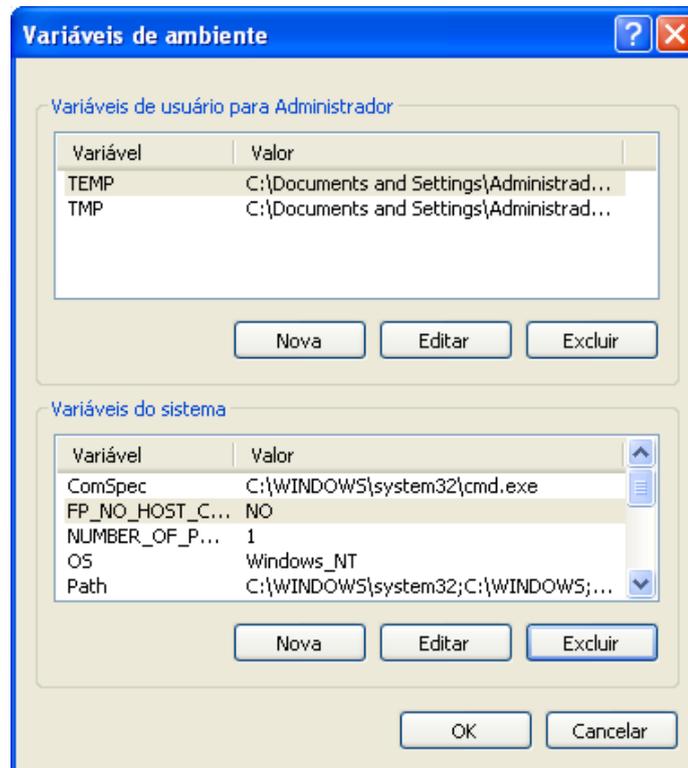
1) Clique com o botão direito em cima do ícone “Meu Computador” e selecione a opção “Propriedades”.



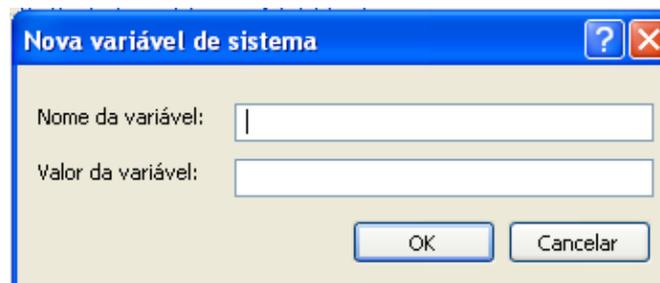
2) Agora escolha a aba “Avançado” e depois clique no botão “Variáveis de Ambiente”



3) Nesta tela, você verá, na parte de cima, as variáveis de ambiente do usuário corrente e, embaixo, as variáveis de ambiente do computador (servem para todos os usuários). Clique no botão “Nova” da parte de baixo

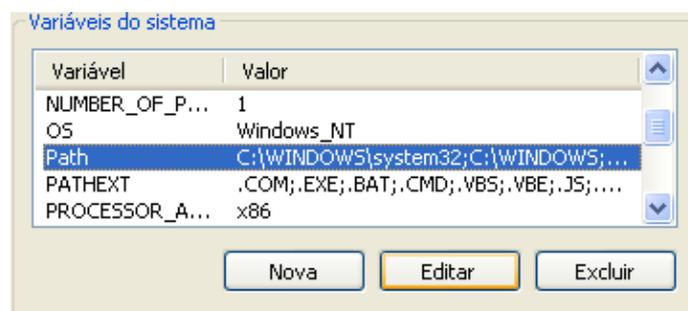


- 4) Agora em “Nome da Variável” digite JAVA\_HOME e, em valor da variável, digite o caminho que você anotou na instalação do Java. Provavelmente, será algo como: C:\Arquivos de programas\Java\jdk1.6.0\_03

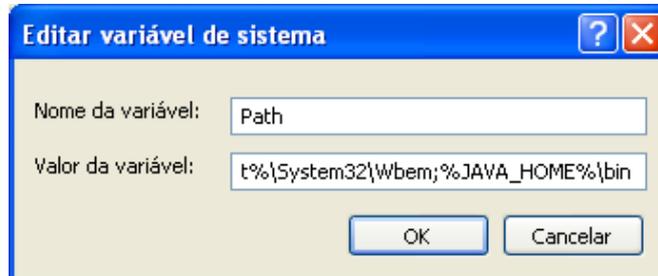


E, depois, clique em OK.

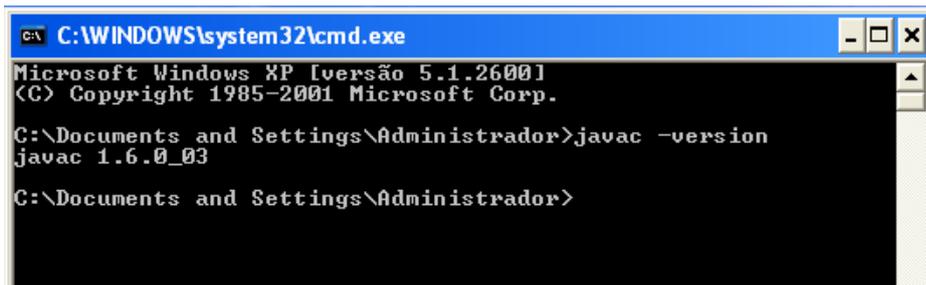
- 5) Crie uma nova variável de ambiente repetindo o passo 11, porém, agora, defina o nome da variável como CLASSPATH e o valor com . (só um ponto).
- 6) Agora, não vamos criar outra variável, mas sim *alterar*. Para isso, procure a variável PATH, ou Path (dá no mesmo), e clique no botão de baixo “Editar”.



- 7) Não altere o nome da variável! Deixe como está e adicione no final do valor ;%JAVA\_HOME%\bin, não esqueça do ponto-e-vírgula - assim, você está adicionando mais um caminho à sua variável Path.



- 8) Agora abra o prompt. Vá em Iniciar, Executar e digite `cmd`.
- 9) No console, digite `javac -version`. O comando deve mostrar a versão do Java Compiler e algumas opções. Caso isso não aconteça, reveja os passos e confira se não esqueceu ou pulou nenhum deles.



---

# Índice Remissivo

ABSTRACT, 97  
ANT, 142  
ARGUMENTO, 32  
ARQUIVOS, 164  
ARRAY, 49  
atomicidade, 202  
ATRIBUIÇÃO, 16  
ATRIBUTO, 31  
AUTOBOXING, 157  
  
boolean, 16  
BREAK, 23  
BYTECODE, 6  
  
CASTING, 18  
CASTING DE REFERÊNCIAS, 154  
CHAR, 16  
CLASSE, 30  
CLASSE ABSTRATA, 97  
Classpath, 208  
COLLECTIONS, 175  
COMPARABLE, 180  
COMPARE TO, 159  
COMPOSIÇÃO, 77  
CONDIÇÃO BOOLEANA, 20  
CONSTRUTOR, 62  
CONTINUE, 23  
CONTRATO, 107  
CTRL+ESPAÇO, 140  
  
Decorator Pattern, 166  
DOUBLE, 16  
  
Eclipse, 83  
ELSE, 21  
ENCAPSULAR, 59  
ENTRADA E SAÍDA, 164  
EQUALS, 156  
  
ESCOPO, 24  
EXCEPTION, 119  
  
FILEINPUTSTREAM, 165  
finalizer, 200  
FINALLY, 132  
FOR, 22  
FULL QUALIFIED NAME, 137  
  
Garbage Collector, 199  
GETTERS, 61  
  
HASHMAP, 190  
HASHTABLE, 190  
HERANÇA, EXTENDS, 71  
  
IF, 20  
IMPLEMENTS, 107  
IMPORT, 137  
INPUTSTREAM, 165  
INPUTSTREAMREADER, 165  
INT, 14  
INTERFACE, 107  
INVOCAÇÃO DE MÉTODO, 33  
ITERATOR, 187  
  
JAR, 207  
JAVA.IO, 164  
JAVA.LANG, 152  
java.lang.Object.finalize, 200  
java.lang.System.gc, 200  
java.util.concurrent, 202  
JAVADOC, 147  
juc, 202  
  
LAÇO, 22  
  
MÁQUINA VIRTUAL, 5  
MÉTODO, 32

MÉTODO ABSTRATO, 99  
MAIN, 10  
MAP, 185  
MATH, 160  
MATRIZ, 49  
MODIFICADOR DE ACESSO, 58  
  
NEW, 31  
NULL, 40  
  
OBJECT, 153  
OPERADOR DE NEGAÇÃO, 21  
OPERADORES ARITMÉTICOS, 15  
OPERADORES LÓGICOS, 21  
ORIENTAÇÃO A OBJETOS, 28  
OutputStream, 216  
  
PACKAGE, 136  
PACOTES, 135  
PARÂMETRO, 32  
PLATAFORMA JAVA, 3  
POLIMORFISMO, 75  
PORTA, 223  
PRIVATE, 57  
PROTECTED, 72  
PUBLIC, 58  
  
REESCRITA, 73  
REESCRITA DE MÉTODO, 73  
REFERÊNCIA, 35  
região crítica, 201  
RETURN, 34  
  
SERVERSOCKET, 224  
SET, 183  
SETTERS, 61  
SOBRECARGA, 105  
SOCKETS, 164  
SPLIT, 159  
STACKTRACE, 120  
STATIC, 65  
STATIC IMPORT, 242  
SUN, 3  
Super e sub classes, 72  
SYSTEM.EXIT, 152  
  
TCP, 222  
THIS, 32  
THROWS, 125  
TOSTRING, 155  
TREEMAP, 190  
  
VALORES DEFAULT, 39  
VARIÁVEIS, 14  
VOID, 32  
  
WHILE, 22  
WRAPPING, 157