

# PÓS-Graduação em Distância



**Java JSE: Programação  
Básica**

**Prof. Luiz Carlos Querino Filho  
Profa. Elisamara de Oliveira**



# Sumário

<b>Apresentação</b>	<b>4</b>
<b>Módulo 1: A Plataforma Java</b>	<b>5</b>
Aula 1: Histórico e evolução da linguagem Java	5
História da linguagem de programação Java	5
Principais conceitos da plataforma Java	7
Aula 2: IDE Eclipse	8
Ambiente de desenvolvimento Java	8
Obtendo e instalando o Eclipse	9
Conhecendo o Eclipse	11
Criando o primeiro projeto no Eclipse	14
Exercício: dado virtual	24
Aula 3: estrutura básica de programação em Java SE	24
Instruções de entrada e saída em janelas	24
Tipos primitivos do Java, <b>strings</b> e conversão de valores	26
Conceitos básicos de memória, comparação de <b>strings</b> e <b>strings</b> mutáveis	28
Uso do operador ternário	30
Projeto calculadora	30
Exercícios do Módulo 1	32
<b>Módulo 2 – Classes e objetos em Java</b>	<b>35</b>
Aula 4 – Conceitos básicos de classes e objetos	35
Classes, objetos, métodos e instância	35
Implementando a classe no sistema bancário	38
Métodos construtores	42
Métodos estáticos	44
Exercício prático - complementando o sistema bancário	46
Aula 5 - Encapsulamento	47
Definição e exemplo de encapsulamento	47
Modificadores de visibilidade: private, public e protected	47
Usando getters e setters	48
Exercício adicional	52
Encapsulamento de campos na prática: exercício prático guiado	53
Exercício do Módulo 2	57
<b>Módulo 3 – Herança e polimorfismo</b>	<b>59</b>
Aula 6 – Herança em Java	59
Conceito de herança	59
Superclasses e subclasses	60
Exercício prático guiado: herança de cliente no sistema bancário	60
Aula 7 – Polimorfismo	67
Conceito de polimorfismo	67
Sobrecarga de métodos	69
Polimorfismo na prática em Java: exercício prático guiado	70
Exercício do Módulo 3	75
<b>Módulo 4 – Arrays de objetos e coleções</b>	<b>76</b>
Aula 8 - Arrays e coleções em Java	76
Arrays de objetos e coleções	76



---

Uso de ArrayList e métodos para manipulação	77
Uso de coleções no sistema bancário: exercício prático guiado	79
Exercício do Módulo 4	83
<b>Respostas Comentadas dos Exercícios</b>	<b>84</b>
Módulo 1	84
Dado Virtual	84
Potência	85
Conversor Celsius para Fahrenheit	86
Histograma	89
Módulo 2	91
Forca	91
Módulo 3	95
Complementando o sistema bancário	95
Funcionarios	97
Módulo 4	101
Contatos	101
<b>Considerações Finais</b>	<b>105</b>
<b>Referências Bibliográficas</b>	<b>106</b>

# Apresentação

---

Caro aluno(a), seja bem-vindo(a) à nossa disciplina, na qual vamos trabalhar juntos nos conceitos e práticas essenciais da linguagem Java.

Neste momento, você já deve possuir uma boa noção dos conceitos básicos de engenharia de *software*, modelagem orientada a objetos, UML e lógica de programação. Todos esses temas são essenciais para esta disciplina, pois a programação em Java permite que você coloque em prática os princípios básicos de engenharia de *software*, implemente os sistemas orientados a objetos modelados com a UML e use os algoritmos e estruturas básicas que fundamentam a lógica de programação.

Nossa disciplina trata da introdução formal ao mundo da linguagem da Java por meio da sua “fundação básica”, o Java SE (*Standard Edition*). A linguagem Java, dada sua história e imensa popularidade, abrange uma série de áreas de utilização. O Java SE concentra-se em abranger os elementos básicos dela, que são comuns para todas as aplicações, sejam programas do tipo *desktop*, aplicativos *web* ou para dispositivos móveis. Ou seja, é o básico da linguagem, que você utilizará em praticamente qualquer programa feito em Java.

Quando você estiver mais avançado dentro da tecnologia Java, conhecerá os outros campos de aplicação da linguagem, como nos softwares empresariais e destinados à *web* que são criados com o Java EE (*Enterprise Edition*), mas perceberá que o Java SE estará sempre lá com você.

Nesta disciplina, começaremos a utilizar um IDE - *integrated development environment*, ou ambiente de desenvolvimento integrado. Um IDE é um aplicativo que facilita bastante o desenvolvimento de *software*, pois agrupa recursos de edição, compilação, teste e depuração de um programa em um mesmo local, aumentando a produtividade do desenvolvedor. Você utilizará, além do IDE, o compilador e interpretador Java incluso no JDK.

Então vamos lá! Prepare-se para mergulhar de cabeça na linguagem de programação orientada a objetos mais usada no mundo.

**Prof. Luiz Carlos Querino Filho**

# Módulo 1: A Plataforma Java

---

Este primeiro módulo trata de uma rápida introdução formal a Java, que pode ser definida não apenas como uma linguagem de programação, mas também uma completa plataforma de desenvolvimento de software. Veremos um pouco de sua história e seu modo de funcionamento e faremos a instalação do ambiente de desenvolvimento integrado no qual criaremos nossos aplicativos.

## Aula 1: Histórico e evolução da linguagem Java

### História da linguagem de programação Java

Para entender o início da linguagem Java, temos que compreender a distinção básica entre os paradigmas de programação orientado a objetos e estruturado.

No paradigma estruturado, que surgiu primeiro, os programas são definidos como uma coleção de *funções* e estruturas de programação básicas, que por sua vez utilizam *dados* armazenados em *variáveis*. Não há uma relação óbvia e direta entre funções e dados; é responsabilidade do programador estabelecê-la, passando as variáveis como parâmetros das funções.

Já no paradigma orientado a objetos, tem-se uma relação mais direta entre dados e ações (as funções da programação estruturada). Os dados possuem funções, que atuam diretamente sobre eles (ou sobre outros dados). Isso ocorre colocando-os em unidades denominadas *objetos*, os quais passam a ter também ações: passam a “saber” como usar ou processar seus dados (ou os existentes em outros objetos).

Usando a orientação a objetos (OO), podemos tornar os programas mais próximos do mundo real, pois nosso dia a dia é uma constante interação entre “objetos” que contêm informações.

As vantagens da programação OO em relação à estruturada são muitas: maior produtividade, aliada a uma grande capacidade de *reutilização*. Assim, várias linguagens foram desenvolvidas implementando esse paradigma. Dentre as que ganharam rápida popularidade e aceitação, nos primórdios, estão SmallTalk e C++. Esta última, principalmente por usar como base a linguagem C, largamente conhecida pelos programadores, tornou-se em pouco tempo a principal linguagem OO em uso. Porém tinha (e ainda tem) alguns pontos negativos, como uma sintaxe complexa para declaração e uso de classes e um modelo de gerenciamento de memória manual, que obriga o programador a ter controle preciso sobre a memória alocada para objetos.



Fonte: <http://www.vectorsland.com/vector/java-eps-logo-99090.html>

De acordo com Deitel e Deitel (2010), em junho de 1991, James Gosling, Mike Sheridan e Patrick Naughton, programadores da extinta Sun Microsystems (que foi adquirida pela Oracle em 2009), começaram a desenvolver uma linguagem de programação OO que fosse mais fácil de ser aprendida e utilizada do que C++. Além disso, deveria possuir recursos avançados de gerenciamento de memória que livrassem o programador da tarefa de alocar e liberar recursos para os objetos e, sobretudo, poder ser executada em computadores de diferentes arquiteturas. Como eram ávidos consumidores de café, batizaram a linguagem com o nome da ilha de onde vinha a sua marca favorita da bebida: Java.

A linguagem Java foi lançada oficialmente em 1995 e, graças a seus recursos poderosos, sua simplicidade e a facilidade no entendimento de sua sintaxe, além da capacidade de execução em múltiplas plataformas, tornou-se rapidamente um sucesso. Outra característica importante que contribuiu para o seu sucesso foi a existência de recursos para a sua utilização na internet. Na época, o acesso à rede começava a ser disponibilizado comercialmente, vindo a alcançar milhões de pessoas e a se tornar parte da vida de todo o mundo.

James Gosling, considerado o “pai do Java”, delineou no documento *The Java language environment* (que se tornou a “definição formal” da linguagem) os cinco princípios básicos dela (GOSLING; MCGILTON, 1997):

1. Java deve ser uma linguagem simples, orientada a objeto e com conceitos familiares aos programadores.
2. Java deve ser robusta e segura.
3. Java deve ser neutra em relação à arquitetura de execução, ou seja, deve ser portável.
4. Java deve executar com alto desempenho.
5. Java deve ser interpretada, utilizar *threads* e ser dinâmica.

À medida que se conhece a linguagem e seus recursos, percebe-se que, duas décadas depois, esses princípios básicos foram plenamente alcançados e implementados.

Em virtude de a Sun (e, hoje, a Oracle) sempre ter disponibilizado as ferramentas para criação de programas Java pela internet (o *JDK*), a linguagem ganhou enorme aceitação na comunidade de desenvolvedores, principalmente entre os entusiastas do chamado *software livre* ou *open source*. Por essas e outras razões, em 2006, a Sun passou a liberar todo o código-fonte principal dela, usando licenças livres como a *GPL*.

Hoje Java é, além de uma linguagem, uma plataforma completa de desenvolvimento e conta com uma rica comunidade de suporte. A filosofia do *software livre* está tão presente nela que, apesar de a marca ainda pertencer a uma empresa, todo o processo de atualização, melhoria e evolução da linguagem é controlado por um grupo que envolve várias pessoas da comunidade, conhecido como *Java Community Process (JCP)*. Até mesmo a mascote oficial da linguagem, o *Duke* (figura 1), tem seu desenho liberado nos termos do *software livre* e pode ser usado sem restrições.

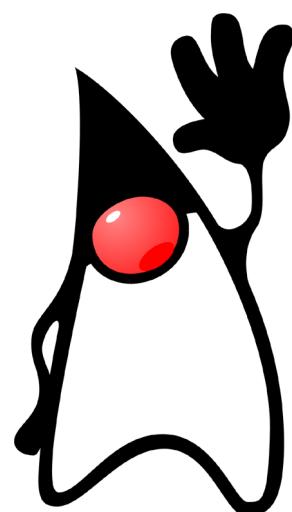


Figura 1: Duke, o mascote oficial da Java. Fonte: [http://commons.wikimedia.org/wiki/File:Duke\\_Wave.png](http://commons.wikimedia.org/wiki/File:Duke_Wave.png).

## Principais conceitos da plataforma Java

Como dissemos, Java não é somente uma linguagem de programação, mas uma plataforma de desenvolvimento de *software*. Mas o que isso significa? Que, junto da linguagem, existem bibliotecas, ferramentas, *frameworks* e recursos para que o programador crie aplicativos nos mais diferentes tipos de equipamentos, de computadores *desktop* a servidores e de *smartphones* a *set-top-boxes*.

A plataforma Java (ou seja, a base em que os programas Java executam) foi criada desde o início com o intuito de ser utilizada nos mais variados tipos de aparelhos, com os mais diversos tipos de processadores, arquiteturas e sistemas operacionais.

Mas como isso é possível?

Se você já programava antes de realizar este curso, com certeza deve conhecer o processo de compilação, em que o programa compilador converte o código-fonte escrito em uma determinada linguagem de programação para linguagem de máquina, que o computador consegue processar. Porém, com esse processo, o programa passa a ficar preso a um tipo específico de computador, que entenda a linguagem de máquina gerada.

Para conseguir sua meta de fazer um mesmo programa ser executado em diferentes plataformas, o Java teve de seguir uma abordagem diferente.

Um programa Java (com código escrito em arquivos com a extensão `.java`) também passa por um processo de *compilação*. Mas o arquivo gerado pelo compilador Java (o `javac`) não conterá código em linguagem de máquina, pois este é sempre dependente da plataforma na qual o programa será executado. Ao invés disso, é gerado em *bytecode*, que são instruções que somente podem ser executadas pelo *Java Virtual Machine (JVM)*, a máquina virtual Java. Esse processo é ilustrado na figura 2.

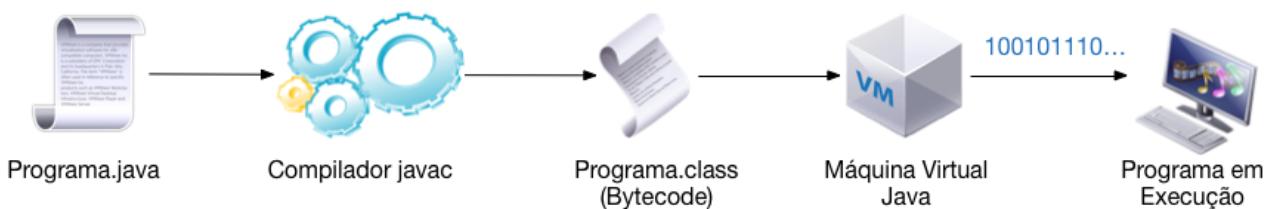


Figura 2: Ciclo de compilação e execução de um programa escrito em Java. Fonte: Adaptado de Oracle (2014).

Observe a sequência do processo de compilação e execução de um programa Java na figura 2. O compilador `javac` analisa o código-fonte escrito no arquivo *Programa.java* e gera o arquivo *Programa.class*, que contém o *bytecode* do programa. Esse *bytecode* apenas poderá ser executado por um programa especial, a máquina virtual Java, que deverá estar sempre presente no computador no qual queremos executar nosso programa. É tarefa da JVM ler o *bytecode* e convertê-lo para a linguagem de máquina específica do computador e do sistema operacional no qual será executado.

O compilador `javac` faz parte do JDK. Mas e a máquina virtual?

O JDK contém, além do javac e das bibliotecas básicas da linguagem, uma cópia da máquina virtual, que também é instalada. Isso porque, sem a JVM, não seria possível executar os programas usando o comando `java`, que é o interpretador de *bytecode*, responsável por enviar o programa para ser executado.

Quem não é programador não precisa, obviamente, instalar o JDK para obter a JVM. Por isso, a Oracle disponibiliza também o *Java Runtime Environment (JRE)*, que contém a máquina virtual para que os usuários comuns possam executar os aplicativos escritos em Java no seu formato compilado, ou seja, em *bytecode*.

Isso leva à seguinte conclusão: a máquina virtual Java funciona como uma espécie de tradutor entre o *bytecode* e a linguagem de máquina do computador no qual o programa será executado. Então, consequentemente, a máquina virtual deve ser feita “sob medida” para a plataforma (o tipo de computador e sistema operacional) no qual o programa será executado.

Por essa razão, o computador (ou dispositivo) em que um programa escrito em Java será executado possui uma máquina virtual construída especificamente para ele. Isso significa que aquela usada em sistemas Windows não funciona em sistemas Linux. Você deverá ter instalada a máquina virtual específica para seu computador e sistema operacional.

## Aula 2: IDE Eclipse

### Ambiente de desenvolvimento Java

Como você viu na figura 2, o processo de compilação e execução de um programa em Java envolve o uso de dois utilitários instalados pelo JDK no seu computador: o compilador `javac` e o interpretador `java`.

Contudo, à medida que os programas Java crescem em tamanho e complexidade, trabalhar com linha de comando torna-se bastante improdutivo. Começa-se a gastar um tempo valioso no ciclo de desenvolvimento do *software*. Para otimizar o processo de criação de um aplicativo, existem os *IDEs* (*integrated development environments*, ou ambientes de desenvolvimento integrado). Esses programas integram um conjunto de recursos que auxiliam o desenvolvedor no processo de criação de um *software*, como:

- Um editor de código, com colorização de sintaxe e possibilidade de autocompletar nomes de comandos e identificadores.
- Um facilitador de acesso ao compilador, em que basta o usuário clicar em um botão para que o processo de compilação seja feito para todos os arquivos de código-fonte existentes no programa.
- Execução automática do programa compilado com sucesso ou, no caso de erros de compilação, a visualização destes e a localização dos pontos no código que necessitam de correção.
- Recursos de depuração visual, para que o programador possa executar o código linha por linha, analisando os valores obtidos.
- Acesso à documentação da linguagem.

Dada a imensa popularidade da linguagem Java, existem diversos IDEs disponíveis para utilização, cada uma com suas vantagens e desvantagens. Os mais populares entre os desenvolvedores são o *Eclipse* (livre e mantido por uma comunidade mundial de usuários) e o *NetBeans* (também livre, mas mantido pela Oracle).

Ambos possuem recursos avançados de edição, compilação e teste de aplicativos Java, podendo ser considerados até mesmo equivalentes em quantidade de funcionalidades.

Nesta disciplina, utilizaremos o Eclipse por uma razão a mais: ele é o IDE “oficial” (por enquanto) para o desenvolvimento de aplicativos Android. Por isso, conhecendo o Eclipse agora, você já começa também a se preparar para a programação em Java para a plataforma de computação móvel mais usada no mundo.

### IDE não é a mesma coisa que compilador!

Essa é uma confusão muito comum.

Não é o IDE que compila os programas. Na maioria dos casos (como no Eclipse), ele funciona como um intermediário entre o usuário e o compilador/interpretador da linguagem. É responsável por passar o código digitado pelo usuário ao compilador e repassar o resultado (como erros e avisos). Existem casos em que um IDE traz junto um compilador, mas, em outros, é necessário que o usuário já o tenha instalado previamente.



## Obtendo e instalando o Eclipse

O Eclipse é um *software* livre, com código-fonte aberto. Isso significa, entre outras coisas, que você não precisa se preocupar com questões de licenciamento e pode usá-lo (e copiá-lo) livremente.

Para fazer o *download* do Eclipse, acesse o endereço <http://www.eclipse.org/downloads> (caso esse *link* não mais funcione, basta procurar pelo endereço atualizado).

Após entrar no site, obtenha a versão *Standard* (ou padrão). Escolha também a opção adequada ao seu sistema operacional e à quantidade de *bits* usados (siga a mesma regra de quando obteve o JDK: se optou pela versão de 64 bits, escolha o Eclipse também em 64 bits) e clique para iniciar o processo de *download*. A figura 3 ilustra esse processo.

The screenshot shows the Eclipse Downloads page. At the top, there's a banner for EclipseCon 2014 in San Francisco. Below the banner, the main navigation menu includes Home, Downloads, Users, Members, Committers, Resources, Projects, and About Us. To the right of the menu is a search bar with a 'Search' button. The main content area has a purple header 'Eclipse Downloads'. Underneath, there are two tabs: 'Packages' (which is selected) and 'Developer Builds'. A red arrow points to the 'Packages' tab. Another red arrow points to the 'Windows' dropdown menu, which is set to 'Eclipse Kepler (4.3.2) SR2 Packages for Windows'. Below this, there's a listing for 'Eclipse Standard 4.3.2, 200 MB'. It shows download statistics: 'Downloaded 546,845 Times' and 'Other Downloads'. To the right of the listing are download links for 'Windows 32 Bit' and 'Windows 64 Bit', each accompanied by a green download icon. To the right of these links is an advertisement for 'ORACLE Enterprise Pack for Eclipse'. At the bottom of the screenshot, there's a caption: 'Figura 3 - Obtendo o Eclipse versão Standard. Fonte: <http://www.eclipse.org/downloads>'.

## Eclipse: IDE Java escrita em... Java!

Para executar o Eclipse, você precisa ter o Java instalado. Isso parece óbvio, já que o usaremos para criar programas nessa linguagem. Mas tem outra coisa...

O Eclipse também foi escrito em Java!

Por isso, você precisa do Java não somente para compilar e executar os programas, mas também para poder executar a ferramenta usada no desenvolvimento dos seus aplicativos.



Após clicar na versão desejada (32 ou 64 bits), você será redirecionado para outra página, na qual poderá iniciar o *download*. Com base na sua localização, o site do Eclipse escolherá o servidor mais rápido dentre os disponíveis. A figura 4 ilustra esse processo.

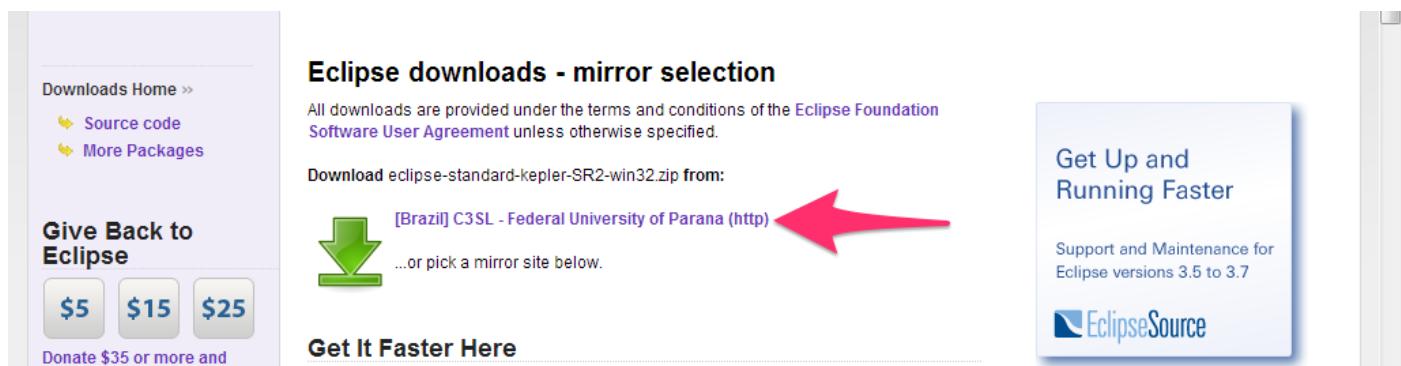


Figura 4 - Link para download do Eclipse. Fonte: <http://www.eclipse.org/downloads>.

Terminado o *download*, você obterá um arquivo compactado (.zip) que já contém o Eclipse pronto para ser executado. Crie uma pasta no local que preferir no seu computador e descompacte o arquivo dentro dela, usando os recursos do seu sistema operacional ou um programa utilitário, como o WinZip ou o WinRAR. O Windows pode descompactar arquivos .zip sem necessitar de utilitários: basta clicar com o botão direito do mouse sobre o arquivo e selecionar Extrair Tudo (veja figura 5).



Figura 5 - Descompactando o pacote do Eclipse.

Na janela que será exibida, escolha (ou crie, após clicar em Procurar) a pasta na qual deseja colocar o Eclipse e clique sobre o botão Extrair.

## Aviso importante!

Não coloque o Eclipse em uma pasta (ou dentro de um caminho de pastas) que contenha espaços ou caracteres internacionais (como acentos e cedilha), pois ele pode apresentar problemas de execução.



Terminada a descompactação, navegue até a pasta em que os arquivos foram extraídos. Você verá que lá se encontra o programa executável da ferramenta, *eclipse.exe* (veja figura 6), com o ícone de um globo. Clicando duas vezes sobre ele, iniciamos o Eclipse. Mas não faça isso agora.

## Eclipse: IDE Java escrita em... Java!

Para executar o Eclipse, você precisa ter o Java instalado. Isso parece óbvio, já que o usaremos para criar programas nessa linguagem. Mas tem outra coisa...

O Eclipse também foi escrito em Java!



Por isso, você precisa do Java não somente para compilar e executar os programas, mas também para poder executar a ferramenta usada no desenvolvimento dos seus aplicativos.



	Documentos			
	Imagens			
	Músicas			
	Vídeos			
	Computador			
	Disco Local (C:)			
	Documents (\vboxsrv) (E:)			
	Rede			
	eclipse	Data de modificação: 09/10/2013 14:09	Data da criação: 09/10/2013 14:09	
		Aplicativo	Tamanho: 312 KB	

Figura 6 - O executável do Eclipse.

Como utilizaremos bastante o programa, crie um atalho para ele na Área de Trabalho. Para fazer isso rapidamente, clique com o botão direito do mouse sobre o arquivo *eclipse.exe*. Mantenha o botão clicado e arraste o arquivo até a Área de Trabalho do Windows. Ao soltar o botão, surgirá um pequeno menu *pop-up*. Selecione *Criar atalho aqui...* para que o atalho ao Eclipse seja criado.

Agora sim, pode clicar duas vezes sobre o atalho e abrir o programa para esta disciplina.

## Conhecendo o Eclipse

Ao abrir o Eclipse pela primeira vez, você será questionado sobre a localização do seu *workspace* (espaço de trabalho). Esse é o nome que o programa dá para uma pasta dentro do seu computador na qual ficarão armazenados os seus projetos. Todo projeto criado será, por padrão, gravado dentro dela.

O Eclipse sugere inicialmente que você use como seu *workspace* uma pasta nova com esse nome que será criada dentro da pasta pessoal do seu usuário (veja figura 7, item 1). Se quiser, você pode mudar a pasta para uma de sua preferência - apenas tome cuidado para não definir o *workspace* em um caminho que contenha espaços ou caracteres de acentuação/cedilha, para evitar problemas futuros.

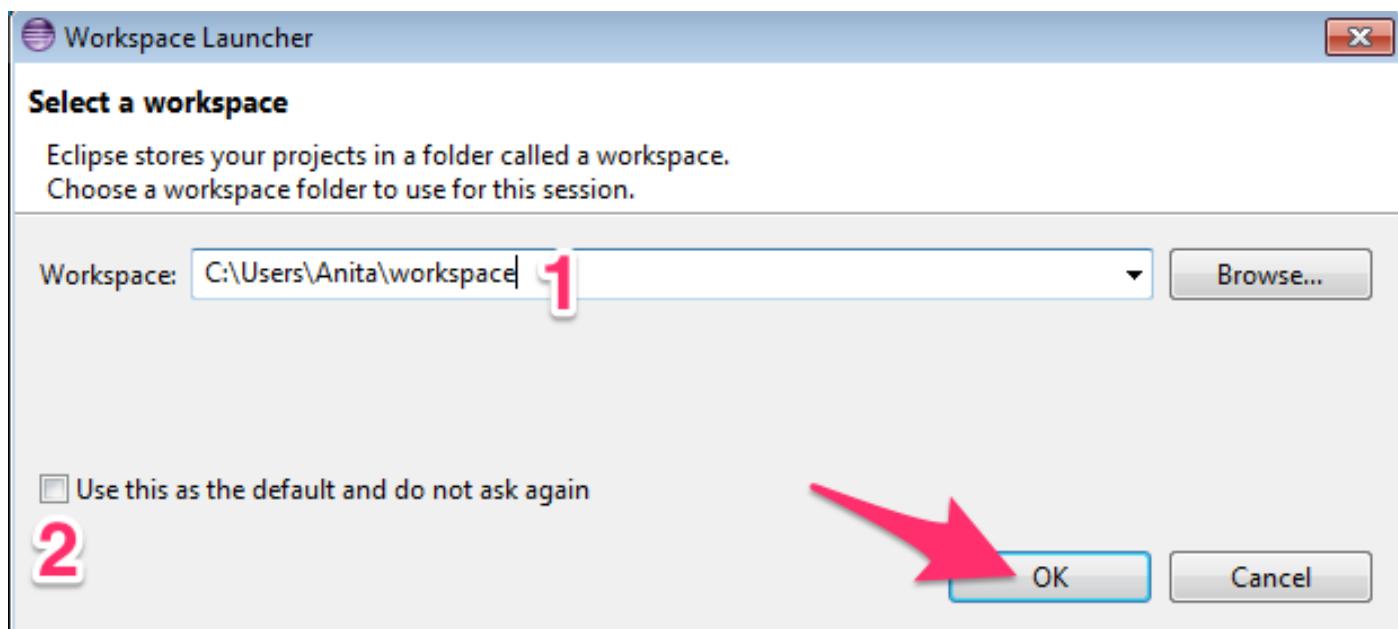


Figura 7 - Definindo o *workspace* do Eclipse. Fonte: Produção do autor.

Observe na figura 7 o item 2: trata-se de uma opção que, se marcada, definirá a pasta indicada como *workspace* padrão e não perguntará mais por isso nas próximas inicializações. Se você tem certeza de que esse será o melhor local para guardar seus projetos, marque essa caixa para que não tenha que confirmar o *workspace* toda vez que abrir o Eclipse. Clique em OK em seguida.

Se você quiser mudar o *workspace* mais tarde, pode fazer isso clicando no menu *File > Switch Workspace > Other*.

Depois de alguns segundos, você será apresentado à tela de boas-vindas do programa. Você pode fechá-la clicando no "X", indicado na figura 8.

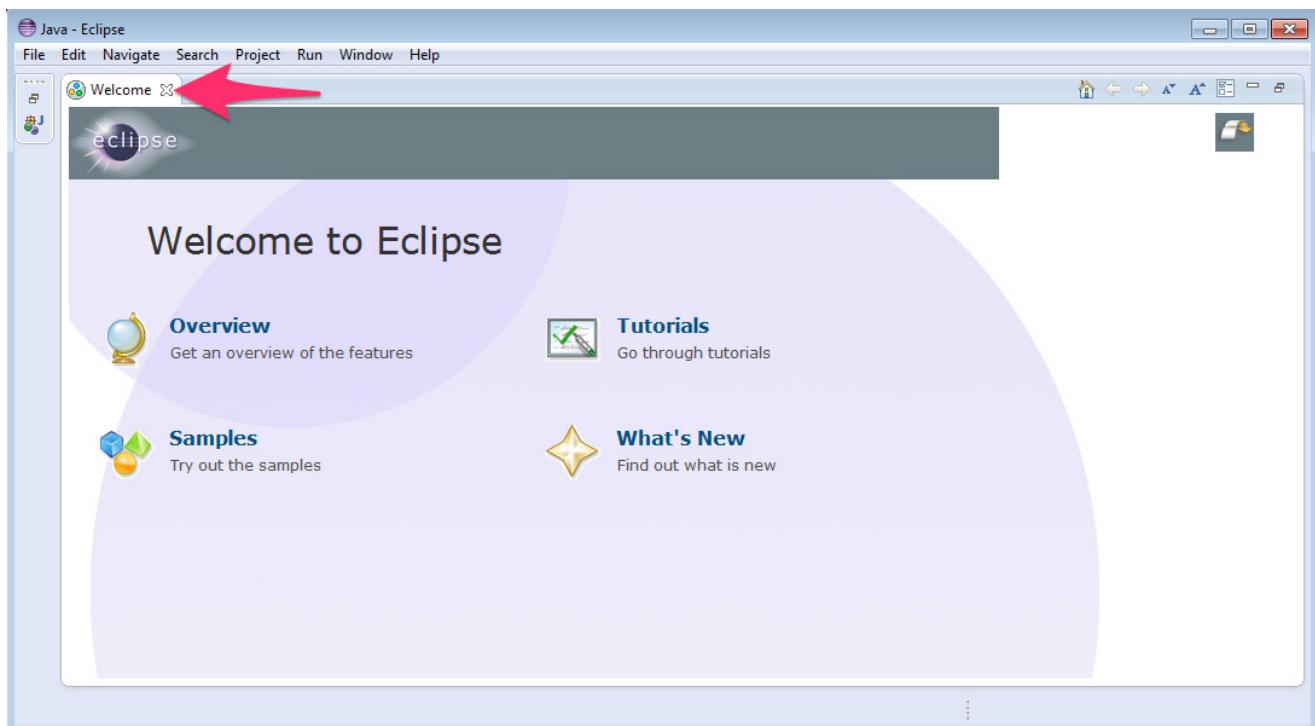


Figura 8 - Tela de boas-vindas do Eclipse. Fonte: Produção do autor.

Agora você poderá visualizar a tela principal do Eclipse. Essa é a chamada *perspectiva Java*. Como o Eclipse tem uma grande variedade de recursos (podendo ser até usado para se criar programas em C/C++ ou PHP), há uma série de possíveis configurações para sua tela principal. Cada uma é chamada de perspectiva e adequada a uma determinada situação. A perspectiva Java, a mais usada por programadores, é a que vemos na figura 9.

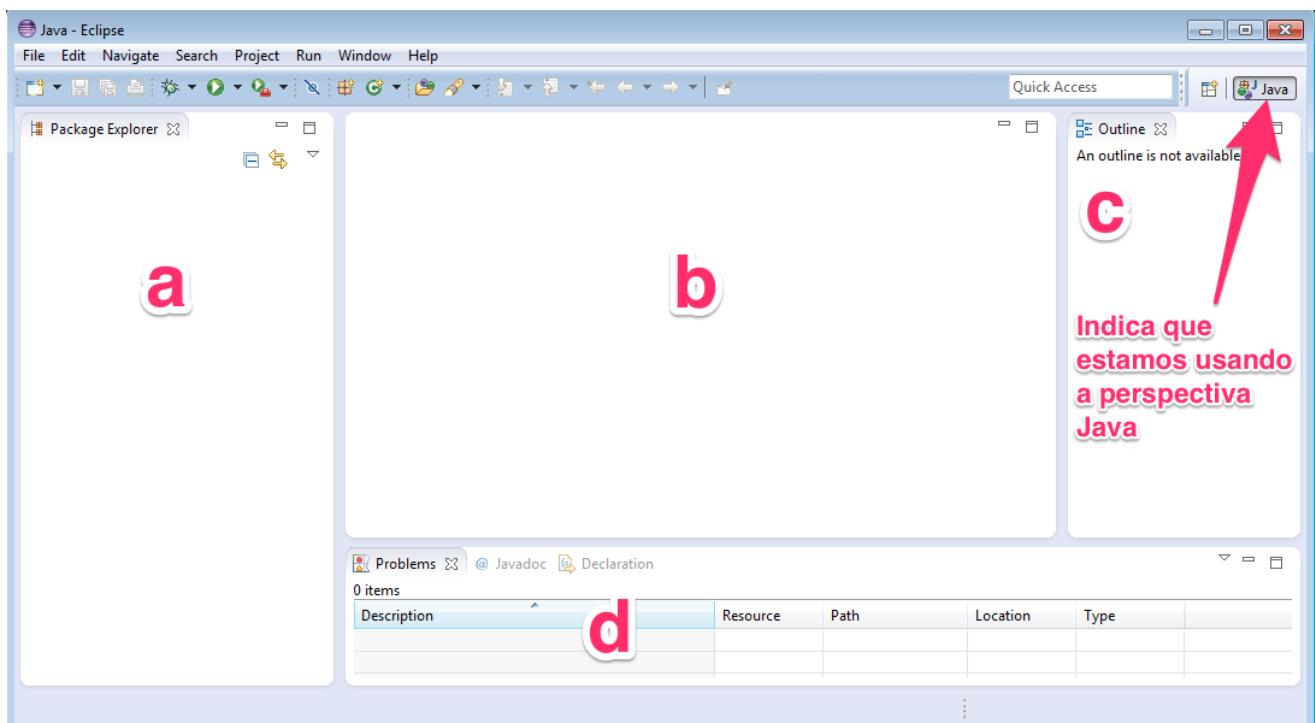


Figura 9 - Perspectiva Java do Eclipse. Fonte: Produção do autor.

Veja na figura que temos uma barra de menus e logo abaixo uma barra de ferramentas, da mesma maneira que na maioria dos aplicativos. A perspectiva Java possui ainda quatro partes internas, indicadas na figura 9 como a, b, c e d. São elas:

- a) Package Explorer:** é aqui que se visualizam os projetos criados no Eclipse e seus arquivos. Todo programa Java criado nele será definido como um projeto, que consiste em um conjunto de arquivos de código-fonte, configurações e bibliotecas, dentre outros.
- b) Editor de código:** nessa divisão central, são abertos os arquivos selecionados dentro de um projeto, por meio do *Package Explorer*. Na maior parte do tempo, você estará com arquivos com código Java abertos nesse local, em que eles poderão ser editados.
- c) Outline:** mostra uma estrutura hierárquica do código-fonte aberto. Por aqui, você pode localizar rapidamente elementos existentes no arquivo, como variáveis e métodos.
- d) Problems/Console/JavaDoc** - nessa área, podem ser abertas diferentes abas, porém aquelas com que você deverá ter mais contato são as seguintes:
  - o **Problems:** aqui serão exibidos os erros e os demais avisos sobre problemas existentes no código.
  - o **Console:** essa aba será aberta quando você executar um aplicativo Java não gráfico, que utiliza comandos de entrada e saída no console.
  - o **JavaDoc:** a importantíssima documentação da linguagem Java. Clicar em cima do nome de uma classe ou método no código-fonte faz com que sua documentação seja mostrada nessa aba (requer que você esteja conectado à internet).

Quanto à barra de menus e aos botões da barra de ferramentas, conheceremos suas funções aos poucos, à medida que desenvolvermos nossos aplicativos.

## Criando o primeiro projeto no Eclipse

Chegou a hora de testarmos esse IDE. Para confirmar se o ambiente está funcional, faremos uma versão do tradicional “Alô, mundo!” utilizando o Eclipse.

Crie um novo projeto Java no Eclipse clicando sobre a seta para baixo logo à direita do primeiro botão da barra de ferramentas e selecionando *Java Project* no menu que será exibido, como indicado na figura 10.

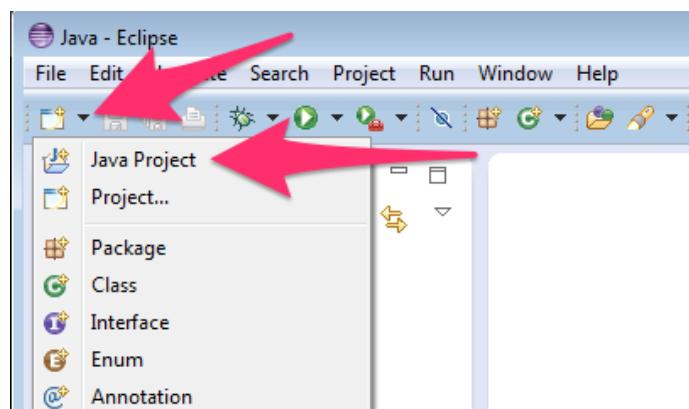


Figura 10 - Criando um novo projeto Java no Eclipse. Fonte: Produção do autor.

Em seguida, será mostrada uma janela, na qual deverá ser fornecido um nome para o projeto, em *Project Name*. Informe *AloEclipse* (dessa forma, sem espaço e sem acento na letra o) e clique em *Finish* (veja figura 11). Repare que, em *Location*, estará indicado que o projeto será gravado em uma subpasta com esse mesmo nome (*AloEclipse*) dentro do seu *workspace*.

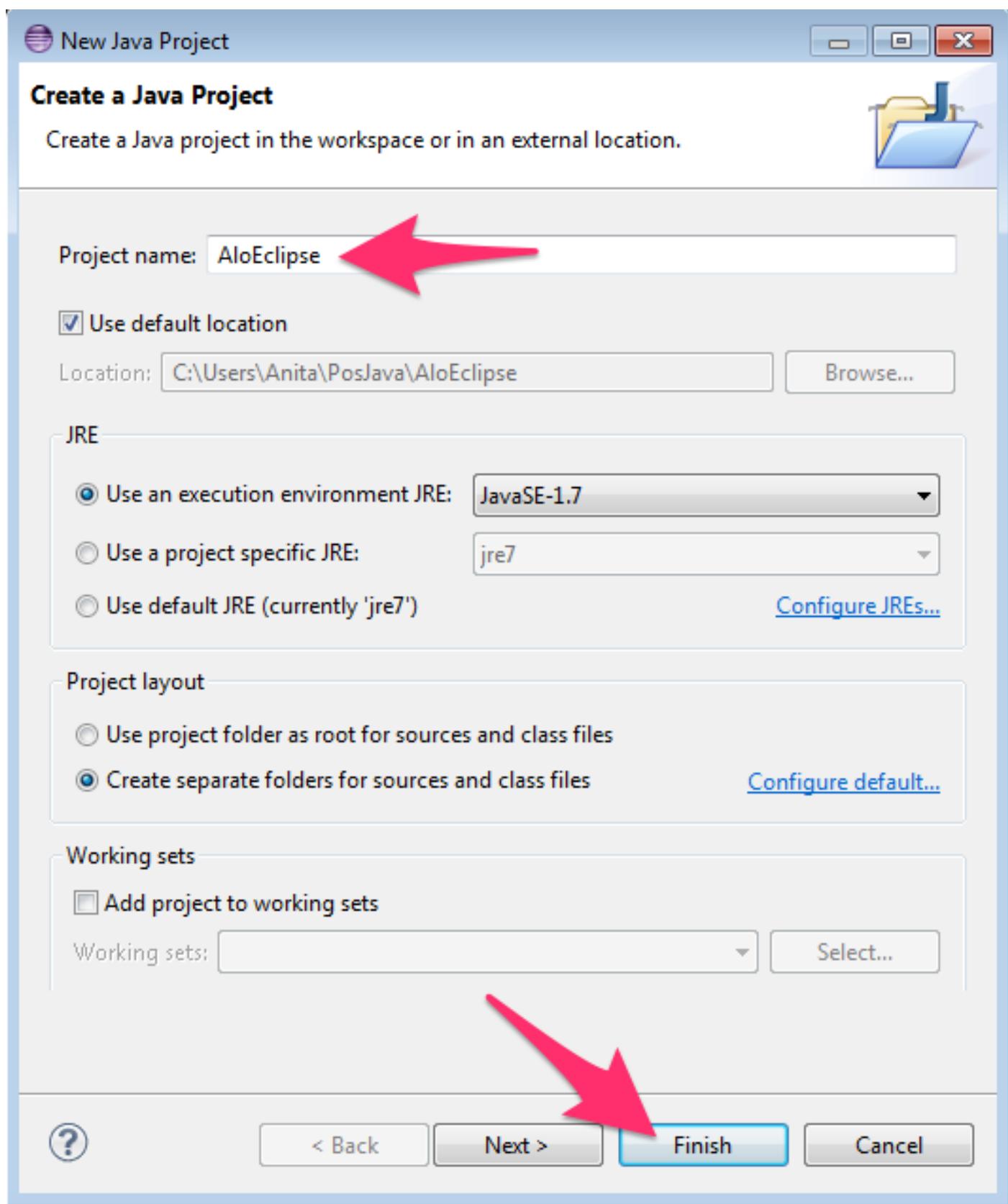


Figura 11 - Criando um novo projeto Java. Fonte: Produção do autor.

Depois de alguns segundos, você verá o projeto AloEclipse, recém-criado, listado no *Package Explorer*. Clique sobre a seta à esquerda do nome do projeto para exibir seu conteúdo (veja figura 12).

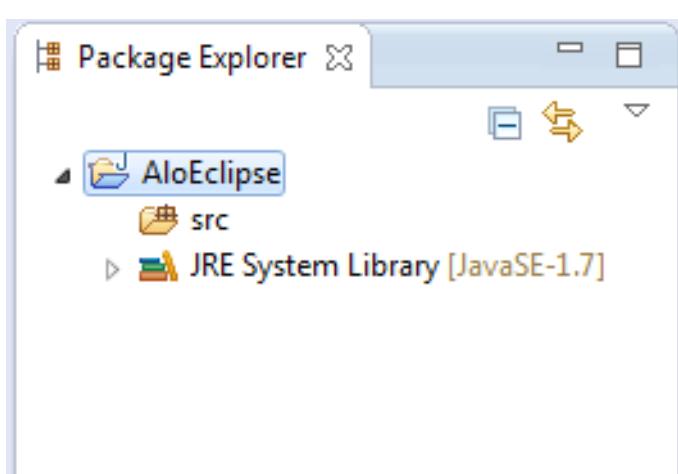


Figura 12 - Projeto AloEclipse e seu conteúdo. Fonte: Produção do autor.

Observe que nosso projeto contém apenas dois itens: uma pasta denominada *src* e uma referência à biblioteca básica do JavaSE, de onde vêm as classes fundamentais da linguagem. O que nos interessa neste momento é a pasta *src* - é nela que colocamos o código-fonte em Java, e podemos ver que por enquanto não temos nenhum.

Em Java, como em toda linguagem orientada a objetos, trabalhamos escrevendo nosso código em classes. Dentro destas, usamos outras classes (algumas criadas por nós mesmos, outras já existentes dentro da biblioteca básica do Java) para definir o funcionamento do nosso programa. Na maioria das vezes, devemos instanciar as classes em objetos para podermos usá-las. Em alguns casos, porém, não precisamos fazer isso: podemos usá-las diretamente.

Por que estamos falando nisso agora, quando ainda não temos nem mesmo uma classe? Porque precisamos ter uma que sirva como “pontapé inicial”, que não será necessariamente usada para criar objetos, mas que contenha dentro dela o início do programa.

Em Java, iniciamos o programa criando uma classe com um nome qualquer que contenha um método especial: *main()*. Esse método foi definido para que programas Java tenham um ponto de entrada, ou seja, possam ter um local determinado para começar. Basta escrevê-lo dentro da primeira classe do projeto (que se torna sua classe principal), seguindo algumas regras básicas com relação à sua sintaxe.

A “mágica” ocorre por meio do interpretador Java: quando executamos um aplicativo, ele busca pelo método *main()* (que deve ter exatamente esse nome) para poder iniciar a execução do programa. Por isso, criaremos uma classe de entrada no aplicativo (que chamaremos neste exemplo de Principal) e, dentro dela escreveremos esse método.

Mas, antes, é preciso outra coisa muito importante.

Já sabemos que a *classe* é a unidade básica de código em um programa Java. Como veremos mais adiante, Java organiza as classes dentro de identificadores, chamados *pacotes*. Um pacote é basicamente uma “classificação” para elas. Além de organizar melhor os aplicativos e bibliotecas, possibilita que várias classes sejam criadas com o mesmo nome sem que ocorram conflitos (veremos como isso é possível mais à frente).

Por isso, antes de criar uma classe, precisamos criar a categoria na qual ela será arquivada, ou seja, o seu pacote. Quando criamos um pacote dentro do Eclipse, é criada uma subpasta dentro da pasta principal de fontes do seu projeto. Quando uma classe for definida como pertencente ao pacote, será colocada dentro dessa pasta. Simples assim.

Para criar um pacote, clique sobre a pasta *src* e em seguida sobre o botão *New Java Package* na barra de ferramentas (como indicado na figura 13).

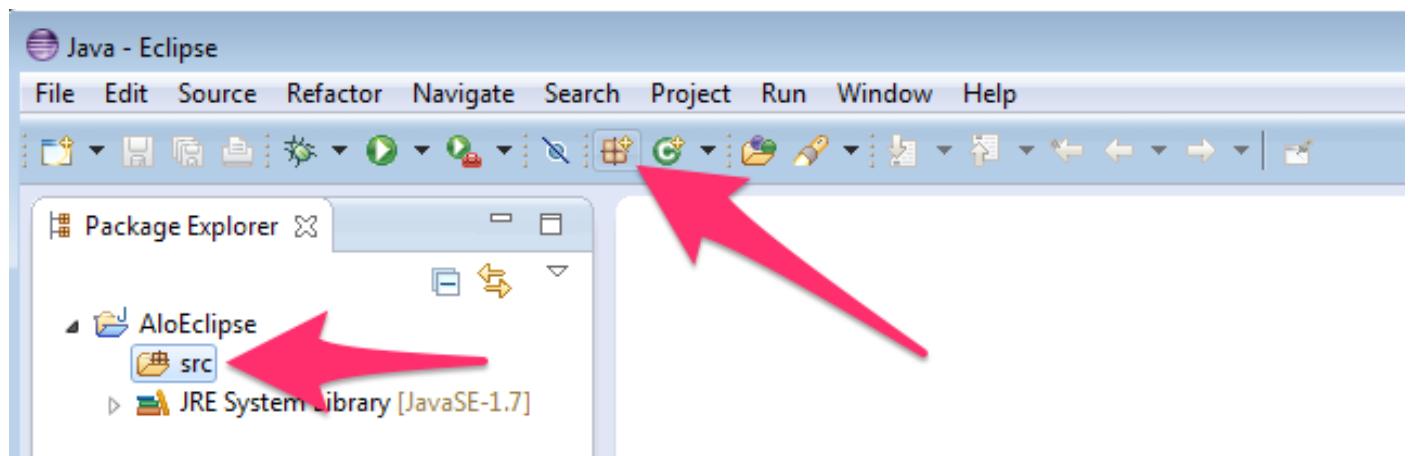


Figura 13 - Criando um novo pacote de classes Java. Fonte: Produção do autor.

Nomes de pacote devem ser definidos sempre com letras minúsculas - essa é a primeira de uma série de convenções usadas pela maioria dos programadores Java que você deve seguir. Outra coisa: existe uma forma padronizada de definir qual o nome de um pacote; veremos isso mais adiante. Por enquanto, defina-o com o mesmo do projeto, só que em letras minúsculas (*aloclipse*). Escreva o seu nome em *Name* e depois clique em *Finish* (veja figura 14).

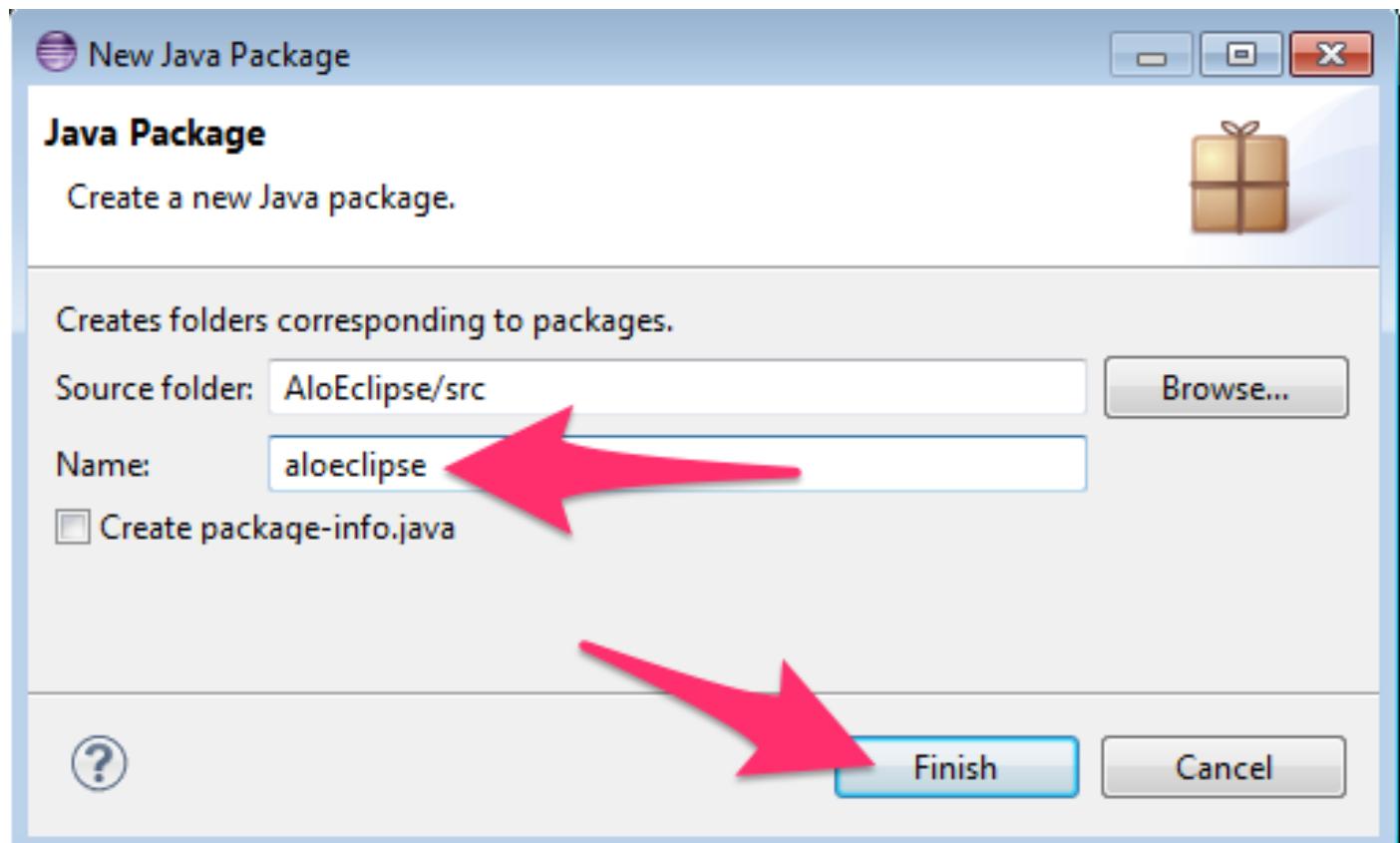


Figura 14 - Criando um novo pacote de classes Java. Fonte: Produção do autor.

Observe à esquerda, no Package Explorer, que agora possuímos um pacote denominado aloeclipse dentro da pasta src. Ele está vazio - precisamos colocar nossa classe dentro dele. No Package Explorer, clique sobre o pacote aloeclipse; em seguida, clique sobre o botão *New Java Class* na barra de ferramentas (como na figura 15). Com essa ação, indicamos ao Eclipse que queremos criar uma nova classe Java e que ela deverá ser colocada dentro do pacote que selecionamos.

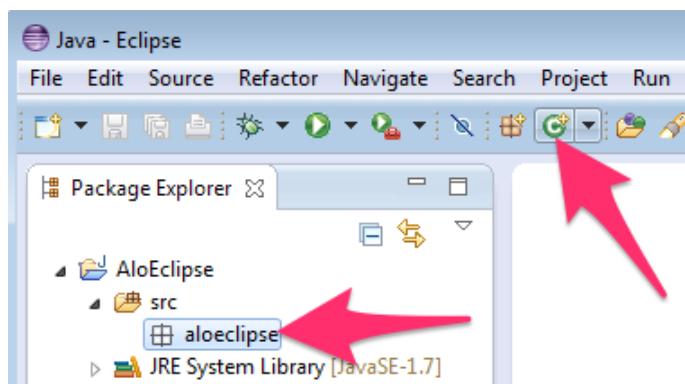


Figura 15 - Adicionando uma nova classe Java ao projeto. Fonte: Produção do autor.

A janela mostrada na figura 16 será exibida. Nela, podemos configurar vários aspectos da nova classe a ser criada. Você pode perceber que, em razão de termos selecionado o pacote aloeclipse no Package Explorer, o Eclipse já detectou que essa classe pertencerá a ele (veja a configuração *Package*) e que seu arquivo correspondente ficará na pasta src do projeto (indicado em *Source Folder*).

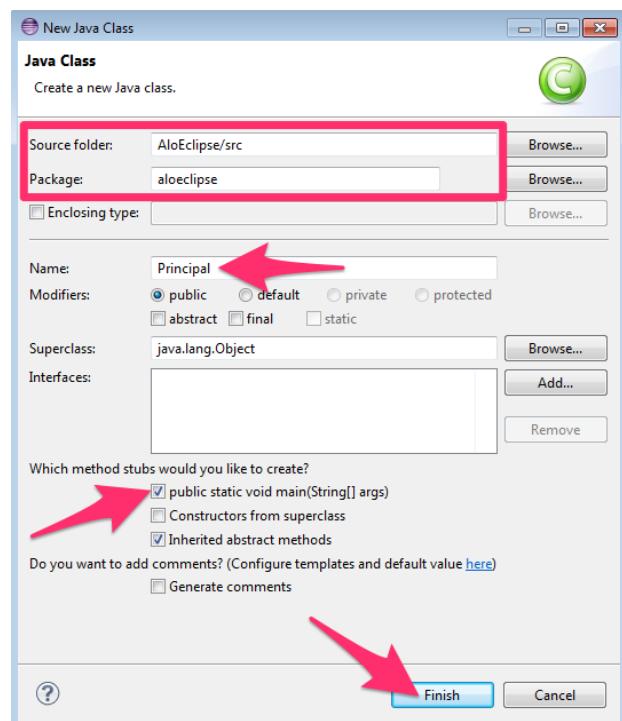


Figura 16 - Tela de criação de uma nova classe. Fonte: Produção do autor.

## Diferenciação de maiúsculas e minúsculas e convenções do código Java



### 1. Java diferencia letras maiúsculas e minúsculas!

A linguagem Java é case sensitive. Portanto, nomes de classes, variáveis, métodos, objetos, ou seja, identificadores em geral, devem sempre ser escritos levando em consideração essa diferenciação.

### 2. Siga as convenções sobre como usar maiúsculas e minúsculas.

Apesar de Java diferenciar maiúsculas das minúsculas nos nomes, o programador é livre para usá-las da maneira que bem entender. Isso quer dizer que você pode criar uma classe chamada **Principal** ou **PRINCIPAL**.

Mas não é tão simples assim.

A maioria dos programadores (e a Oracle, até) segue um padrão de como usá-las. Um deles é *sempre começar o nome de uma classe com uma letra maiúscula*. Esta disciplina foi escrita seguindo (e indicando) o uso desses padrões conforme necessário, para que você vá se habituando aos poucos.



Siga o que está indicado na figura 16 e informe o nome da classe (em *Name*) como Principal (dessa maneira exata, com a letra P maiúscula e as restantes minúsculas). Também não se esqueça de marcar a caixa indicada pela outra seta (*public static void main...*). Em seguida, clique sobre o botão Finish para que a classe seja criada.

Ao selecionar a caixa de marcação *public static void main...*, indicamos ao Eclipse que essa classe deverá ser criada contendo um método *main()* vazio, que servirá como “porta de entrada” ao nosso aplicativo. Como você já sabe, esse é um método muito especial da linguagem Java e deve sempre ser declarado e escrito seguindo uma sintaxe específica, para que a máquina virtual do JRE possa encontrá-lo (se ele existir) no seu programa. Poderíamos ter deixado essa opção desmarcada e posteriormente adicionado o método manualmente, mas assim teríamos que saber exatamente como ele deve ser declarado.

### O que são métodos?

Métodos são os nomes dados às funções desempenhadas pela classe, ou seja, às suas ações. Os métodos são as implementações do que foi denominado operações em UML.

Veja o novo arquivo criado no seu projeto, o *Principal.java*. Ele será aberto na tela central do Eclipse (veja figura 17).

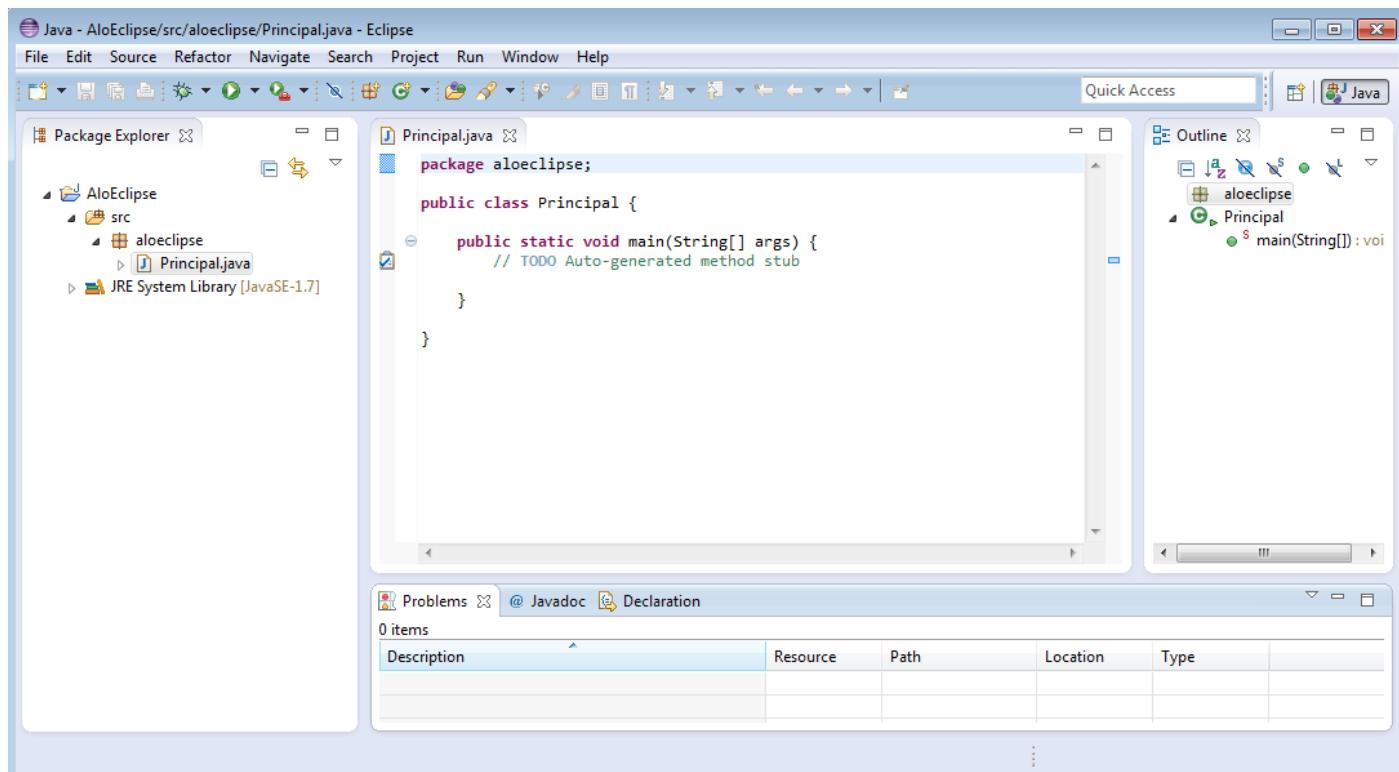


Figura 17 - Tela principal do Eclipse com o arquivo Principal.java aberto.

Um arquivo *.java* é um arquivo de texto sem formatação, que pode ser editado com qualquer editor. Mas, quando é aberto dentro do Eclipse, possui sua sintaxe colorizada e também passa a ser processado pelo editor do programa; com isso, você recebe alguns avisos e dicas na barra cinza à esquerda do código.

Veja, por exemplo, o ícone de um *bloco de notas* logo à esquerda da linha //TODO na figura 17. Isso indica que o Eclipse a reconheceu como um tipo de comentário especial, usado para designar tarefas a ser realizadas no *software*.

### Configure o Eclipse para exibir os números das linhas do código



Um recurso importante do Eclipse, que vem desabilitado por padrão, é a exibição dos números das linhas do código, à esquerda. Essa exibição ajuda em vários aspectos, como a localização de erros.

Habilite a exibição dos números de linha clicando no menu Window > Preferences do Eclipse. Na janela que será aberta, clique sobre a chave General > Editors > Text Editors, à esquerda, e marque, à direita, a opção Show line numbers.

Em seguida, clique em OK para que a numeração passe a ser exibida na coluna da esquerda do editor de código.

Aproveitando que os números de linhas apareceram, vamos entender o conteúdo desse arquivo a partir do código mostrado no quadro 1.

**Quadro 1** – Código-fonte do programa Principal.java

```

1 package aloclipse;
2 public class Principal {
3
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8
9     }
10 }
```

- **Linha 1:** definimos que a classe existente nesse arquivo pertence ao pacote (package) aloclipse.
- **Linha 3:** aqui, inicia-se a classe Principal. A palavra-chave public indica o seu encapsulamento (público, ou seja, essa classe é acessível por qualquer outra). O identificador class indica que se trata, obviamente, de uma classe, e deve ser seguido pelo nome dela. Ao final da linha, abrimos um bloco de chaves que somente será fechado na linha 10.
- **Linha 5:** dentro de uma classe, definimos seu conteúdo (ou seja, seus atributos e operações). Nesse caso, começamos a definir um método (operação). Esse método, como já vimos, é especial: serve como ponto de entrada para a execução do programa. Por isso, sua declaração deve seguir esse padrão: ser público (public), estático (static), não retornar valores (void) e chamar-se main, recebendo como parâmetro um vetor de strings (String[]). O método começa nessa linha e termina na linha 8.

Agora, vamos ao que interessa: programar. Logo abaixo da linha do comentário `//TODO` (que você pode remover, se quiser), insira o comando do quadro 2.

### Quadro 2 - Comando para exibição de caixa de mensagem

```
JOptionPane.showMessageDialog(null, "Alô, Eclipse!");
```

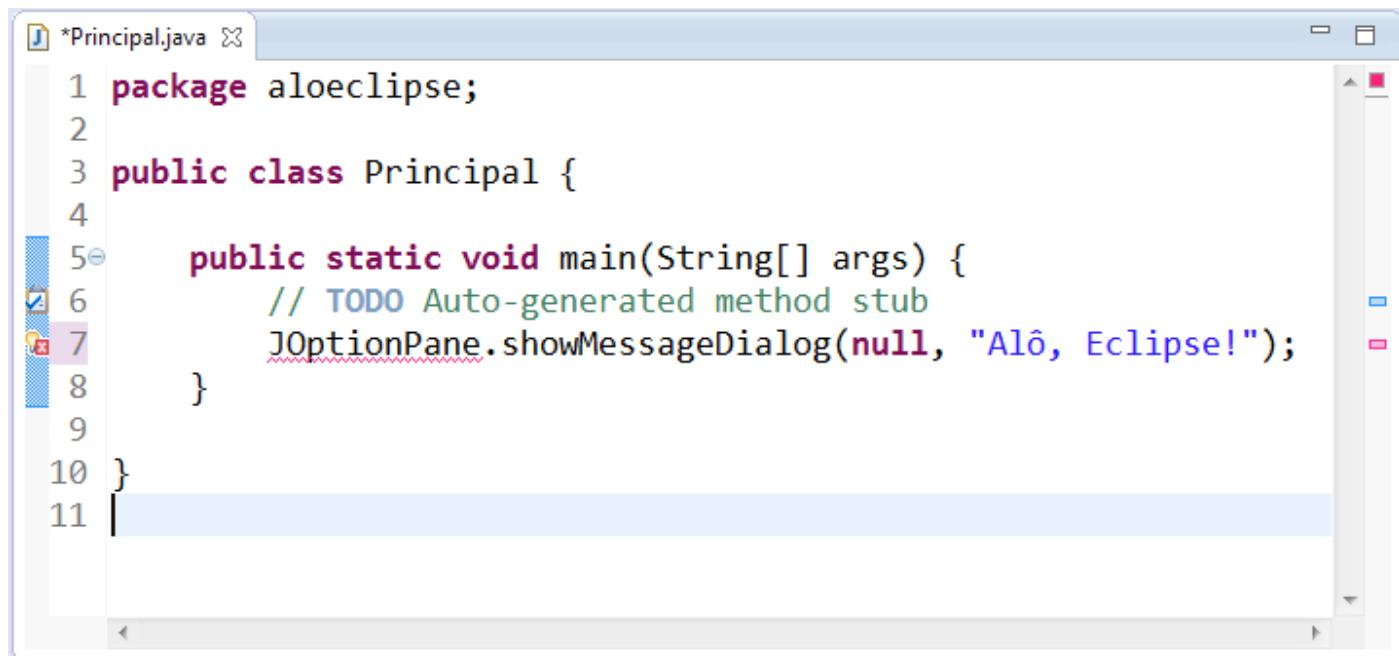
Poderíamos ter usado também `System.out.println`, que funcionaria normalmente. Mas vamos variar e ver como fica melhor.

Estamos usando aqui uma classe do Java (ou seja, ela faz parte da biblioteca padrão da linguagem) chamada `JOptionPane`. Ela é utilizada para exibição de uma pequena janela de mensagens e faz parte de um *framework* (um conjunto de classes) chamado *Swing* - esse é o nome do pacote de componentes gráficos que o Java usa para a criação de aplicativos em janelas.

O ponto (.) logo após `JOptionPane` indica que vamos usar um de seus métodos (ou seja, as ações que essa classe sabe fazer). Nesse caso, estamos chamando o método `showMessageDialog`.

Esse método exibe uma pequena janela contendo o texto passado como segundo argumento (uma *string* Java). O primeiro argumento pode ser mantido como *null*, pois somente é usado em casos em que a janela de mensagem será exibida por outra janela.

Não podemos executar nosso projeto ainda. Existe um erro nele, como você pode ver pelo sublinhado em vermelho na linha 7, como mostra a figura 18.



```
*Principal.java
1 package aloeclipse;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         JOptionPane.showMessageDialog(null, "Alô, Eclipse!");
8     }
9
10 }
11
```

Figura 18 - Código do arquivo Principal.java. Fonte: Produção do autor.

O Eclipse, além de detectar que existe um erro relacionado à classe `JOptionPane`, proporciona formas automatizadas de tentar resolver o problema (trata-se de um recurso denominado *Quick Fix*). Você pode ver

as sugestões de correção de duas formas: clicando e mantendo o cursor do mouse sobre o elemento com erro ou clicando diretamente no símbolo da pequena lâmpada com um “x” na coluna da esquerda. Das duas maneiras, será exibido um menu com sugestões de possíveis soluções, como na figura 19.

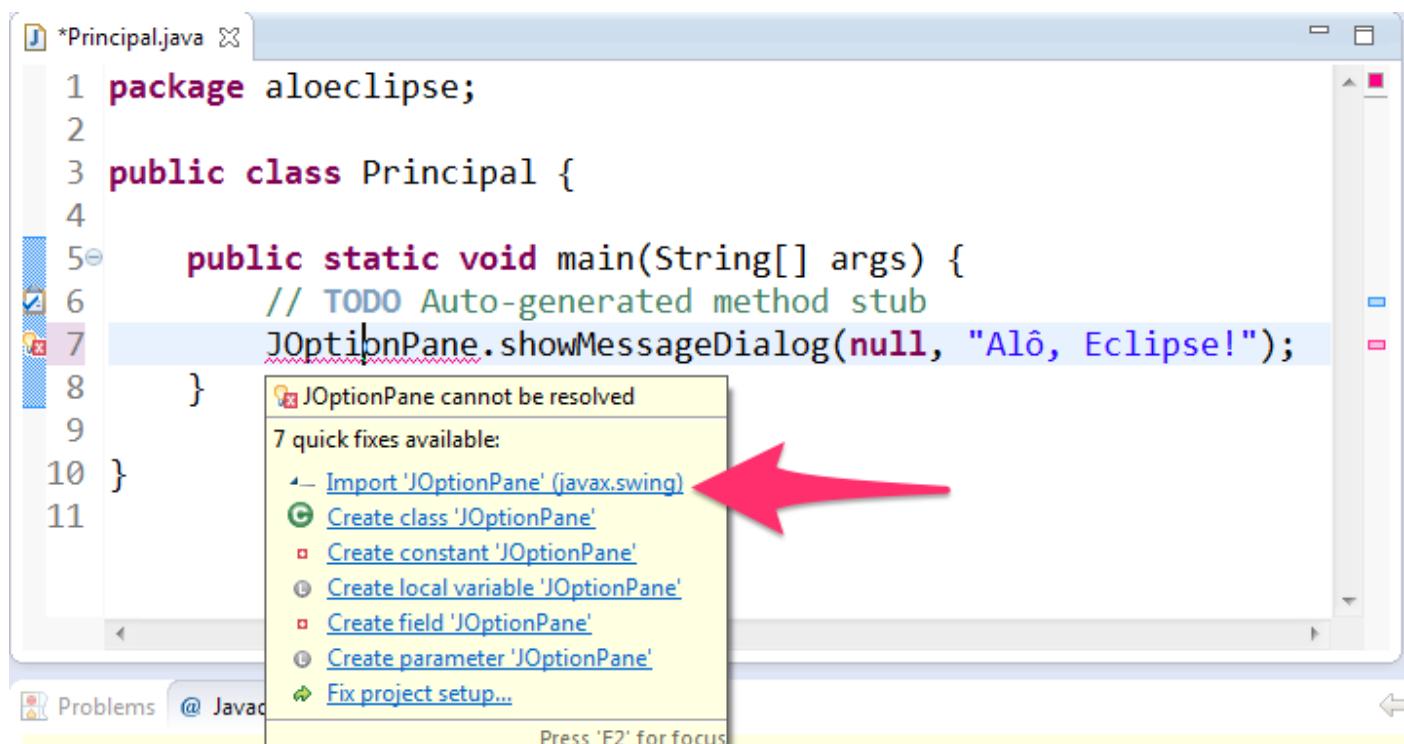


Figura 19 - Quick Fix para solução de erros no programa. Fonte: Produção do autor.

Não basta simplesmente clicar em qualquer uma das soluções e achar que tudo será resolvido. Você deve saber o que está corrigindo, antes de tudo.

Esse erro está relacionado a um fato muito simples: o Java está dizendo que não conhece a classe JOptionPane. “Mas como?”, você pergunta. “Você não disse antes que JOptionPane é parte da biblioteca padrão do Java?”. Ocorre que a classe está lá, mas a nossa classe Principal ainda não foi informada sobre sua existência. Isso é feito usando um comando da linguagem Java chamado *import*, que faz com que uma classe passe a ser “reconhecida” dentro da outra.

Clique sobre a primeira sugestão da lista, como indicado na figura 19, e veja o que mudou no seu código. Primeiro: o erro sumiu. Segundo, a linha exibida no quadro 3 foi adicionada ao código, logo depois de package.

#### Quadro 3 - Linha de importação da classe JOptionPane

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

Por que existe *javax.swing* antes do nome da classe? Trata-se do pacote ao qual ela pertence, pois não são só as classes criadas que pertencem a pacotes.

Agora sim podemos executar o programa e ver o que acontece. Só uma precaução antes: *salve!*

Você deve ter notado, no lado esquerdo do nome do arquivo *Principal.java*, na aba do editor de código, um pequeno asterisco (\*) (figura 20). Ele indica que o arquivo ainda não foi salvo.

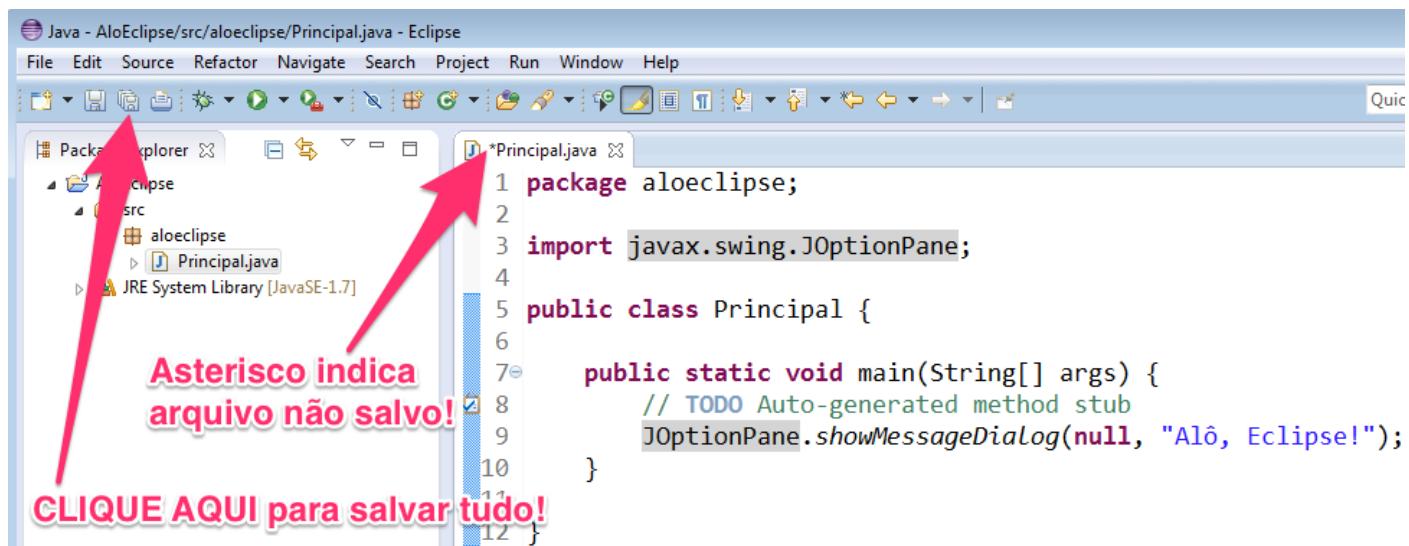


Figura 20 - Salvando todos os arquivos do seu projeto.

Habite-se a salvar todo o conteúdo do seu projeto de tempos em tempos. Para isso, prefira usar sempre o botão *Save All*, para salvar tudo de uma vez só, como indicado na figura.

Se não estivéssemos usando um IDE, agora seria o momento de compilar o arquivo .java para *bytecode* (arquivo .class) e executá-lo com o interpretador java. Mas tudo isso é automatizado pelo IDE, clicando sobre um simples botão do Eclipse: o *Run* (figura 21).

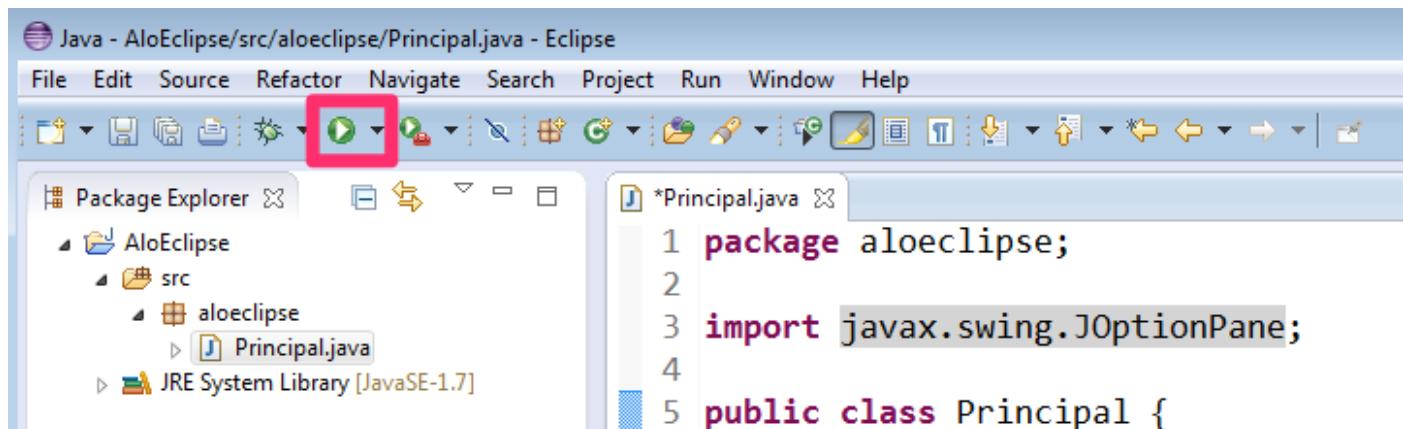


Figura 21 - Botão Run da barra de ferramentas do Eclipse. Fonte: Produção do autor.

Será aberta uma pequena janela, na qual o Eclipse pede que você informe o que está querendo executar. Clique sobre Java Application e, em seguida, em OK (figura 22).

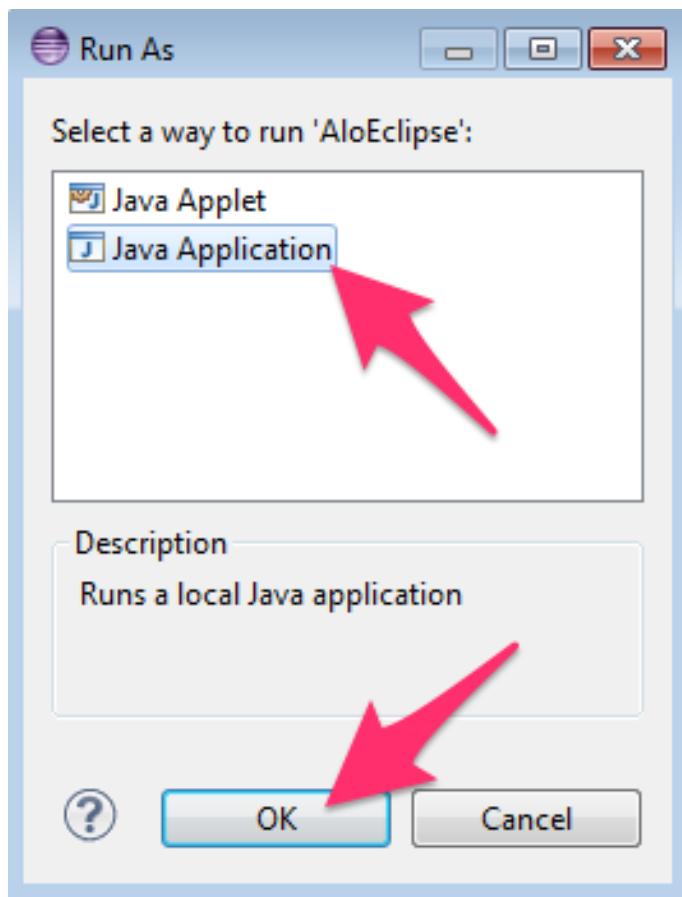


Figura 22 - Executando o projeto como um aplicativo Java. Fonte: Produção do autor.

Em poucos segundos, seu aplicativo será executado e compilado, e uma pequena janela com a mensagem “Alô, Eclipse!” será exibida (figura 23).

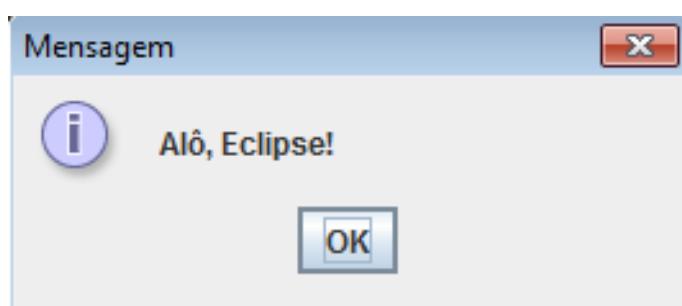
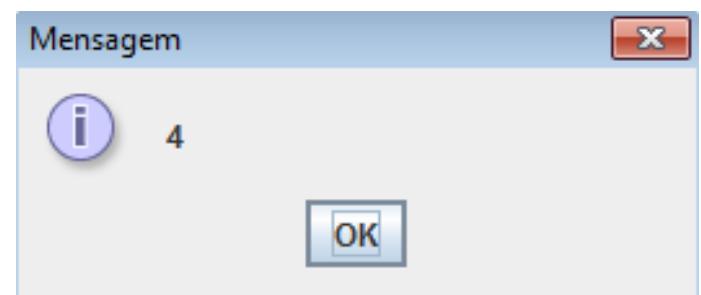


Figura 23 - Tela do aplicativo AloEclipse em execução. Fonte: Produção do autor.

Muito bem, agora você sabe como é criar um aplicativo Java usando uma ferramenta de desenvolvimento integrado.

### Exercício: dado virtual

- Crie um projeto no Eclipse chamado Dado.
- Crie um pacote chamado dado (em letras minúsculas).
- Adicione ao pacote dado uma classe chamada Jogada, com um método main().
- Dentro desse método main(), você deverá exibir ao usuário, usando um JOptionPane, um número sorteado aleatoriamente de 1 a 6 (incluindo estes), como se tivéssemos um dado de seis lados.
- Para fazer o sorteio, pesquise como gerar valores aleatórios com a classe Random.



## Aula 3: estrutura básica de programação em Java SE

### Instruções de entrada e saída em janelas

Para saída de dados no console (em modo apenas texto), você já foi apresentado ao comando `System.out.println()`, que imprime na tela o argumento passado a ele, seguido de uma quebra de linha.

Com a intenção de começar a usar recursos de interface gráfica, passaremos a utilizar a classe `JOptionPane`, que faz parte do framework Swing, o padrão atual para construção de aplicativos em janelas em Java.

A classe `JOptionPane` possui vários métodos estáticos para exibição de caixas de mensagens - pequenas janelas contendo algum tipo de informação.

## O que são métodos estáticos?

Em resumo, métodos estáticos são aqueles executados diretamente de uma classe e não de um objeto instanciado a partir dela. Falaremos mais sobre eles em breve.

O primeiro que conhecemos é o `showMessageDialog()`, que simplesmente exibe uma caixa de mensagem contendo um texto e um botão OK. Ao clicar no botão, a janelinha é fechada. Ele é usado basicamente para exibir informações e avisos, e você já conheceu seu funcionamento na aula anterior.

Existe também um outro método estático importante em JOptionPane que serve para a entrada de dados: o `showInputDialog()`. Chamando esse método, uma pequena janela semelhante à do showMessageDialog() será mostrada, com uma novidade: uma caixa de entrada de texto para que o usuário digite algo. Esse valor digitado será devolvido como uma *string* por esse método. Veja sua sintaxe básica no quadro 4.

**Quadro 4** - Sintaxe do método `showInputDialog()`

```
String nome = JOptionPane.showInputDialog(null, "Qual é o seu nome?");
```

Na execução dessa linha, a caixa de mensagem exibida na figura 24 será mostrada.

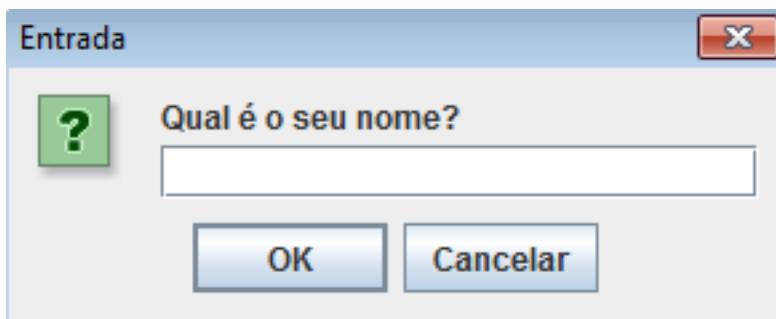


Figura 24 - Caixa de entrada de texto exibida pelo método `showInputDialog()`.

O usuário digita a informação na caixa e, após clicar no botão OK, o valor informado será retornado como uma *string*. No código do quadro 4, guardamos esse retorno em uma *string* chamada nome.

Acrescente a linha do quadro 5 ao seu código e veja como, dessa maneira, criamos um pequeno programa muito bem educado.

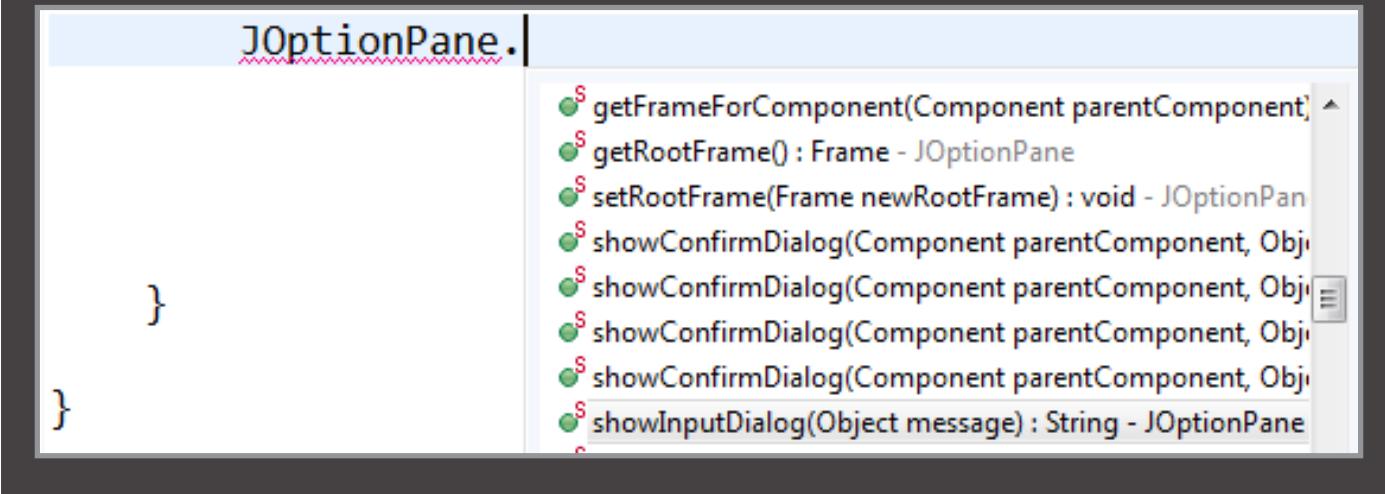
**Quadro 5** - Linha de mensagem com `showMessageDialog`

```
JOptionPane.showMessageDialog(null, "Prazer em conhecê-lo(a), " + nome);
```

## Use bastante o autocompletar do Eclipse



Neste ponto, você já deve ter visto o recurso de autocompletar em ação. Ele aparece alguns segundos após você digitar um ponto depois do nome de um objeto ou classe ou quando está escrevendo qualquer identificador e pressiona CTRL + ESPAÇO.



## Tipos primitivos do Java, strings e conversão de valores

Você já conhece os tipos básicos (também chamados tipos primitivos) de Java, como *int* (para valores inteiros), *double* (para valores decimais), *char* (para um único caractere) e *boolean* (para valores *true*, verdadeiro, ou *false*, falso).

É importante lembrar que Java trata números literais sem ponto decimal como *int* e aqueles que possuem ponto como *double*. Ou seja, *int* e *double* são os tipos *default* para valores numéricos colocados diretamente no código. Por exemplo, o compilador Java vai considerar que estamos passando um *int* para o método no quadro 6.

**Quadro 6** - Exemplo de uso de literal *int*

```
produto.adicionarQuantidadeAoEstoque(5);
```

No exemplo do quadro 7, como o valor passado possui ponto decimal, será considerado como um *double*:

**Quadro 7** - Exemplo de uso de literal *double*

```
aluno.gravarMedia(9.5);
```

Você também já conhece uma *string*. Em Java, ela (que é uma sequência de caracteres alfanuméricos) não é um tipo primitivo, mas, efetivamente, uma classe. Por isso, quando usamos e declaramos uma *string*,

não estamos lidando com uma variável simples, mas com um objeto. A diferença é que objetos são mais complexos e poderosos que variáveis de tipos primitivos.

Um recurso muito versátil das *strings* Java é sua possibilidade de concatenação, ou seja, de junção de duas ou mais em uma nova. A concatenação é realizada usando o sinal de adição (+) entre elas. Veja o exemplo no quadro 8.

#### Quadro 8 - Exemplos de concatenação de *strings*

```
String nome = "João";
String sobreNome = "Silva";
String nomeCompleto = nome + " " + sobreNome;
JOptionPane.showMessageDialog(null, "Meu nome completo é " +
nomeCompleto);
int idade = 22;
JOptionPane.showMessageDialog(null, "Tenho " + idade + " anos de
idade.");
```

O exemplo mostra várias situações em que usamos a concatenação de *strings*. O primeiro JOptionPane exibirá o texto "Meu nome completo é João Silva". Já o segundo mostrará "Tenho 22 anos de idade". Veja que, mesmo idade sendo uma variável *int*, foi convertida automaticamente para *string*. Isso sempre acontecerá em situações em que estivermos concatenando um valor numérico com outra *string*.

Apesar de a conversão no sentido valor numérico -> *string* ser automática quando esses valores são concatenados, há situações em que precisamos fazê-la manualmente. Isso ocorre principalmente na entrada de dados.

Sabemos que o método showInputDialog() do JOptionPane retorna o valor informado com uma *string*. Pois bem: suponha que estamos solicitando ao usuário um valor numérico que será usado em um cálculo matemático. Não poderemos realizar operações matemáticas em um objeto *string*. Precisamos obter um valor em um tipo numérico (*int* ou *double*, por exemplo) a partir desse valor.

Para obter o valor *int* equivalente a uma *string*, usamos o método *parseInt()*, existente na classe *Integer* (uma classe utilitária para inteiros, usada também quando precisamos tratar um valor inteiro como um objeto, e não como um tipo primitivo *int*). Passamos a esse método a *string* contendo o valor numérico, e ele nos devolve o *int* equivalente. O quadro 9 ilustra esse processo.

#### Quadro 9 - Exemplos de concatenação de *strings*

```
String dias = JOptionPane.showInputDialog(null, "Quantos dias atrasados p/ pagto.?");
int diasEmAtraso = Integer.parseInt(dias);
```

Já no caso de *double*, usamos o método *parseDouble()*, disponível na classe *Double* (outra classe utilitária, também utilizada nos casos em que é necessário um objeto contendo um valor com casas decimais), com funcionamento semelhante ao anterior. Veja exemplo no quadro 10.

#### Quadro 10 - Exemplos de concatenação de *strings*

```
String valor = JOptionPane.showInputDialog(null, "Qual é o valor total da conta?");
double valorDaConta = Double.parseDouble(valor);
```

#### Conversões podem dar errado? Infelizmente, sim



Suponha que, no exemplo anterior, o usuário inadvertidamente forneça um número inválido como valor de conta, incluindo, por exemplo, letras no campo de digitação. Exemplos: "12mmn", "3.4Stt", "asdasd".

Nesses casos, não será possível realizar a conversão. Ocorrerá o que chamamos de exceção em Java. Para impedir que o usuário receba uma mensagem de erro, precisamos tratar a exceção, ou seja, definir alguma ação caso ela ocorra. Fazemos isso em Java usando a instrução *try...catch*.

Para o caso de uma conversão numérica, poderíamos tratar o erro de fornecimento de um valor inválido da seguinte forma:

```
double conta = 0;
try {
    String valor = JOptionPane.showInputDialog(null, "Qual o valor da conta?");
    conta = Double.parseDouble(valor);
} catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(null, "VALOR INVÁLIDO! Digite novamente.");
}
```

### Conceitos básicos de memória, comparação de *strings* e *strings* mutáveis

Em Java, objetos, assim como variáveis, são alocados como espaços na memória do computador. Os nomes que definimos para eles servem apenas como referências a essas áreas. É por esse motivo que não devemos comparar duas *strings* em Java usando o sinal de igualdade (`==`), e sim o método *equals*, existente nos objetos dessa classe.

Quando comparamos duas *strings* usando `==`, não será feita uma comparação do conteúdo, mas sim da posição de memória ocupada pelo objeto. Mas, quando usamos *equals()*, a comparação é feita precisamente pelo conteúdo. Por isso, *sempre que precisar comparar strings por valor em Java, prefira usar equals()*.

Mas há uma outra questão importante, também relativa ao uso de memória em Java. Toda vez que declaramos um objeto da classe *String*, estamos definindo uma cadeia de caracteres imutável. Isso significa simplesmente que, após definir o conteúdo de uma *string*, não podemos mais modificar seu conteúdo. Ou

seja, não podemos acrescentar, remover ou trocar caracteres dentro dela. Isso só pode ser feito por meio de uma reatribuição.

Para entender melhor, veja o exemplo no quadro 11.

**Quadro 11** - Exemplo de mudança de uma *string* usando reatribuição

```
String resultado = “”;  
resultado = resultado + “Nome do candidato 1\n”;  
resultado = resultado + “Nome do candidato 2\n”;
```

Suponha que estejamos construindo uma *string* que vai conter uma lista de nomes de candidatos aprovados em um concurso. O código anterior resulta em uma lista impressa na tela semelhante a:

```
Nome do candidato 1  
Nome do candidato 2
```

Pode-se tirar duas conclusões nisso: obteve-se o resultado pretendido e foi possível mudar o valor da *string* resultado, pois concatenamos a ela duas linhas novas de texto. Mas isso não é realmente o que aconteceu.

A *string* resultado continua imutável. O que fizemos foi, nas duas últimas linhas do código, reatribuir um valor a ela, composto do seu conteúdo anterior concatenado com a nova linha. Isso resulta em duas cópias (para cada linha de concatenação) da variável na memória.

Em um exemplo como esse, ou com algumas poucas linhas a mais, não haveria grandes problemas em relação a uso de memória e desempenho do programa. Porém suponha que estejamos montando uma *string* resultante da concatenação (usando reatribuição) de centenas de linhas em um laço. Isso significa centenas de cópias desnecessárias da variável na memória, com seu conteúdo crescendo exponencialmente, ou seja, muita memória desperdiçada e sérios problemas de desempenho para o aplicativo.

Em casos como esse, é muito mais interessante usar a classe *StringBuilder* do Java, que foi criada justamente para essa finalidade: proporcionar a criação de uma *string* mutável sem degradação do desempenho. É, literalmente, um construtor de *strings*.

Para usar *StringBuilder*, basta instanciá-la em um objeto e usar o método *append* dela para que um novo conteúdo seja concatenado à *string*, com mínimo prejuízo de memória e desempenho. O exemplo do quadro 11, usando *StringBuilder*, ficaria como apresentado no quadro 12.

### Quadro 12 - Criando uma *string* concatenada com StringBuilder

```
StringBuilder resultado = new StringBuilder();
resultado.append("Nome do candidato 1\n");
resultado.append("Nome do candidato 2\n");
String stringResultante = resultado.toString();
```

Posteriormente, para obter do objeto `StringBuilder` uma *string* tradicional, basta usar o método `toString()`.

### Uso do operador ternário

Ignorado por muitos, o operador ternário `? :` é uma forma conveniente de se obter um resultado com base no teste verdadeiro ou falso de uma condição, tudo em uma única linha. Pode ser usado para substituir estruturas `if...else`, tornando o código mais conciso. Sua sintaxe é a seguinte:

```
resultado = condição ? valorSeVerdadeiro : valorSeFalso
```

Caso a condição seja avaliada como verdadeira, o resultado receberá `valorSeVerdadeiro`. Caso seja avaliada como falsa, receberá `valorSeFalso`. Exemplo prático em Java:

```
String resultadoDoAluno = (nota >= 6 ? "Aprovado" : "Reprovado");
```

### Projeto calculadora

Para praticar conversão de *string* para valores numéricos, uso de operadores aritméticos e concatenação de *strings*, vamos implementar um projeto de uma calculadora simples que solicitará inicialmente dois números ao usuário, por meio de `JOptionPane.showInputDialog()`, e exibirá, usando apenas um `JOptionPane.showMessageDialog()`, o resultado das quatro operações matemáticas básicas usando ambos. Ao final, o projeto deverá funcionar de modo semelhante à sequência de execução definida na figura 25.

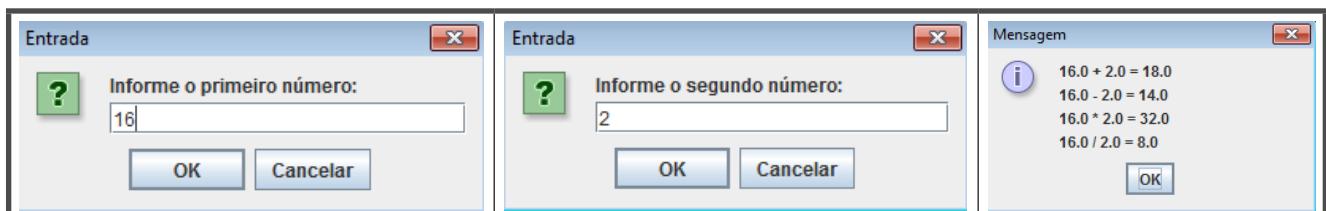


Figura 25 - Sequência de execução para a calculadora, recebendo os números 16 e 2.

Para começar, crie um projeto chamado *Calculadora*. Dentro da pasta `src` desse projeto, adicione um pacote chamado *calculadora*. Por fim, nesse pacote, crie uma classe chamada *Calc*, contendo um método `main()`. Você deverá ficar com uma estrutura de projeto como a exibida na figura 26.

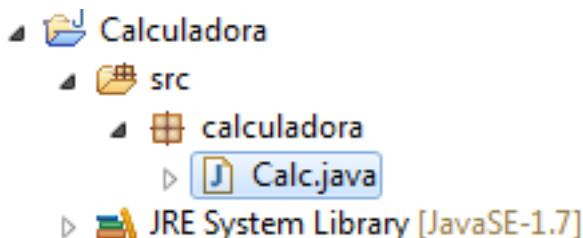


Figura 26 - Estrutura do projeto Calculadora.

Nosso programa deverá executar as seguintes etapas:

- Obter o primeiro número. Caso o usuário digite caracteres inválidos, será exibido um aviso e atribuído 0 ao valor.
- Obter o segundo número, fazendo a mesma validação realizada para o primeiro.
- Calcular soma, subtração, divisão e multiplicação desses números.
- Montar uma *string* que exiba os resultados das quatro operações.
- Exibir a *string* ao usuário em uma janela.

Conseguiremos realizar facilmente esses passos usando os conceitos aprendidos até agora. Um forma de implementar esse exercício é mostrada no quadro 13.

### Quadro 13 – Código-fonte do arquivo Calc.java

```

1 package calculadora;
2
3 import javax.swing.JOptionPane;
4
5 public class Calc {
6
7     public static void main(String[] args) {
8
9         double valor1 = 0;
10        double valor2 = 0;
11        double soma, sub, mult, div;
12
13        String valorStr;
14
15        try {
16            valorStr = JOptionPane.showInputDialog(null, "Informe o primeiro número:");
17            valor1 = Double.parseDouble(valorStr);
18        } catch (NumberFormatException ex) {
19            JOptionPane.showMessageDialog(null, "Número inválido. Considerando 0 para número 1.");
20        }
21
22        try {
23            valorStr = JOptionPane.showInputDialog(null, "Informe o segundo número:");
24            valor2 = Double.parseDouble(valorStr);
25        } catch (NumberFormatException ex) {
26            JOptionPane.showMessageDialog(null, "Número inválido. Considerando 0 para número 2.");
27        }
28
29        soma = valor1 + valor2;
30        sub = valor1 - valor2;
31        mult = valor1 * valor2;
32        div = valor1 / valor2;
33
34        StringBuilder resposta = new StringBuilder();
35        resposta.append(valor1 + " + " + valor2 + " = " + soma + "\n");
36        resposta.append(valor1 + " - " + valor2 + " = " + sub + "\n");
37        resposta.append(valor1 + " * " + valor2 + " = " + mult + "\n");
38        resposta.append(valor1 + " / " + valor2 + " = " + div + "\n");
39
40        JOptionPane.showMessageDialog(null, resposta);
41
42    }
43
44 }
```

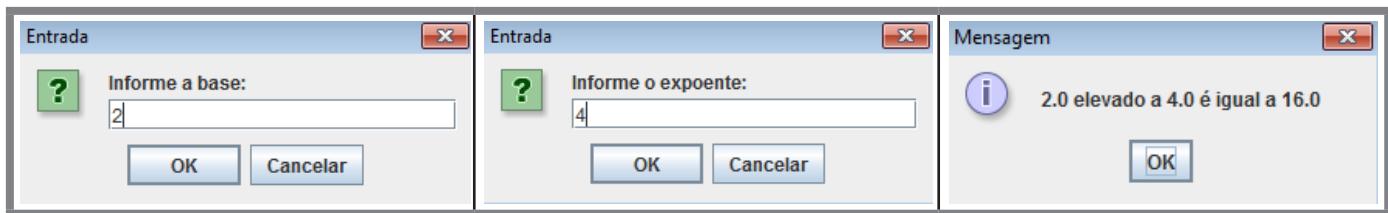
Vamos entender o que está sendo feito nos principais pontos da listagem.

- **Linhas 9 a 11:** declaração de variáveis *double* para os valores e para os resultados das operações matemáticas. Note que zeramos os valores de valor1 e valor2. Isso é necessário pois o uso do try...catch na conversão de *string* para *double* (que temos posteriormente) faz com que o compilador Java indique um possível cálculo com valor não inicializado nas operações. Além do mais, definimos anteriormente que valores fornecidos incorretamente ficarão zerados, então já começamos com essa consideração.
- **Linha 13:** declaramos uma única variável *string* para poder apenas guardar o valor retornado pelos métodos showInputDialog.
- **Linhas 15 a 20:** solicitamos o valor1 ao usuário. Caso seja informado um valor inválido, ocorrerá uma exceção de formatação. Nesse caso, exibimos uma mensagem de alerta, indicando que o valor para o número em questão ficará zerado.
- **Linhas 22 a 27:** fazemos o mesmo processo das linhas 15 a 20, só que agora para o segundo número (variável valor2).
- **Linhas 29 a 32:** realizamos aqui as quatro operações matemáticas básicas usando os valores fornecidos.
- **Linhas 34 a 38:** para montar a *string* que será exibida ao usuário, contendo as quatro operações, realizaremos várias concatenações. Por isso, como vimos anteriormente, nada melhor que utilizar um objeto *StringBuilder*, juntando valores e símbolos por meio do seu método *append*. Veja que, ao final da cada operação matemática, concatenamos também a *string* "\n". Esse é um caractere de escape, oriundo da linguagem C, que resulta em uma quebra de linha na impressão do texto.
- **Linha 40:** encerramos o programa exibindo a *string* criada no objeto *StringBuilder*.

## Exercícios do Módulo 1

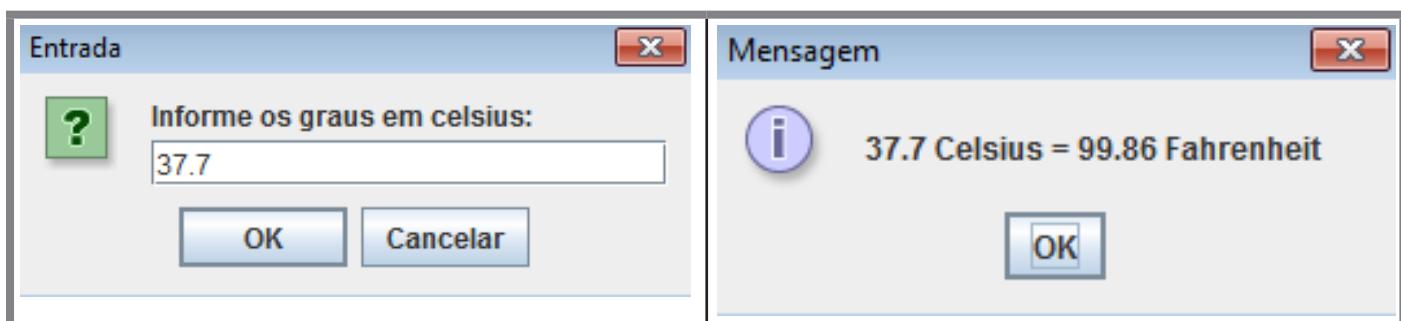
### 1) Projeto Potência:

- Crie um projeto no Eclipse chamado Potencia, com um pacote chamado potencia (em letras minúsculas) e uma classe também denominada Potencia, com um método main().
- Dentro de método main(), solicite, usando um JOptionPane por vez, uma base e um expoente. Armazene-os como tipo double.
- Caso o usuário forneça uma base ou um expoente inválidos, dê uma mensagem informando o fato e dizendo que o programa será fechado.
- Se a base e o expoente forem válidos, calcule a potência usando os valores passados.
- **Dica:** Java possui uma classe chamada Math, com diversos métodos estáticos para funções matemáticas. Um deles serve para calcular uma potência passando uma base e um expoente como parâmetro. Use o autocompletar do Eclipse ou a documentação da linguagem Java para descobrir qual é (<http://docs.oracle.com/javase/7/docs/api/>).
- **Dica:** para fechar um programa, basta usar return na função main() para encerrá-la.



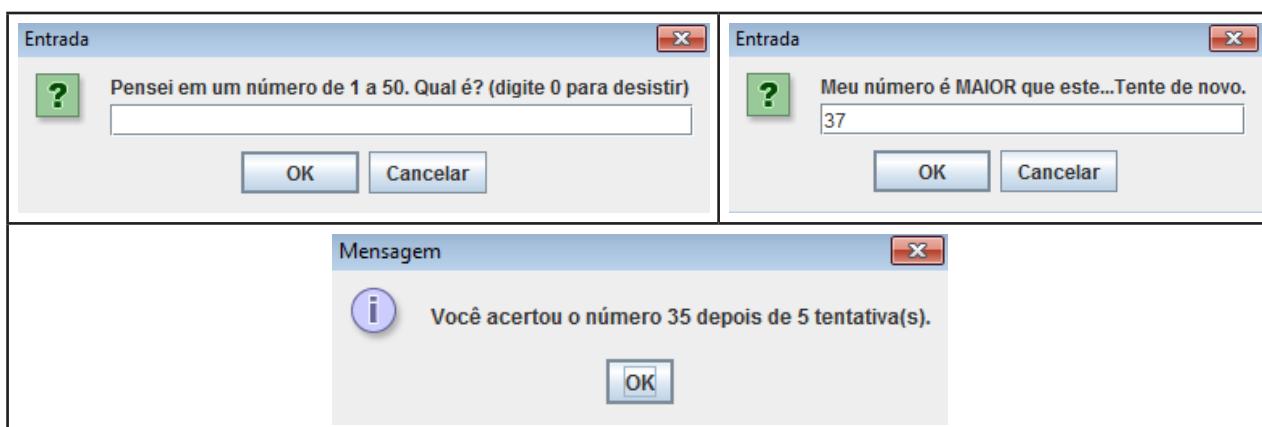
### 2) Projeto Conversor Celsius para Fahrenheit:

- No Brasil, usamos a escala Celsius para medição de temperatura. Nos Estados Unidos, contudo, é utilizada a escala Fahrenheit. Usando uma simples fórmula matemática, é possível obter os graus Fahrenheit a partir dos graus Celsius:  $Fahrenheit = ((Celsius * 9) / 5) + 32$ .
- Crie um programa Java no Eclipse que solicita os graus Celsius para o usuário e retorna os graus Fahrenheit equivalentes.
- A partir deste momento, você deverá decidir que nome usará para seu projeto, pacote e classe. Procure seguir as convenções no uso de letras maiúsculas e minúsculas.



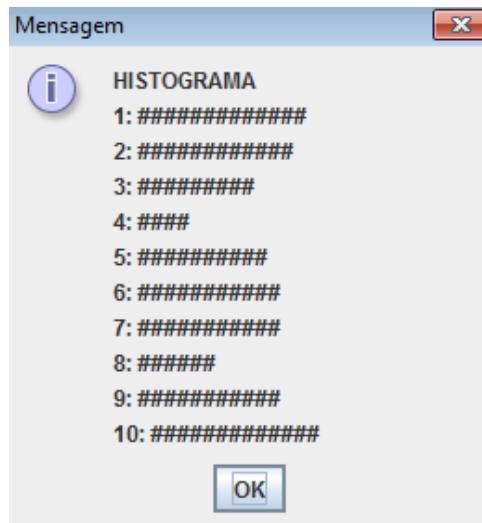
### 3) Projeto Adivinhe o número:

- Neste exercício, vamos elaborar um “mini-jogo” de adivinhação.
- Sorteie um número de 1 a 50 (inclusive).
- Solicite ao usuário que tente adivinhar qual é esse número.
- Se o palpite for superior ao número sorteado, informe-o de que o número é inferior ao que ele forneceu. Caso seja inferior, avise-o de que o número correto é superior.
- Quando ele acertar o valor, mostre quantas tentativas fez até aquele ponto e encerre o programa.
- Permita que o usuário desista de acertar ao fornecer o número 0 como palpite.



#### 4) Projeto Histograma:

- Um histograma (também chamado de diagrama de frequências) é um gráfico usado para visualizar a quantidade de ocorrências de um conjunto de dados agrupado em classes. Vamos criar um aplicativo que mostre um “histograma em modo texto”, exibindo quantas vezes números de 1 a 10 foram sorteados dentro de 100 sorteios possíveis.
- Você precisará repetir 100 vezes o sorteio de um número de 1 a 10 e guardar quantas vezes cada um foi sorteado. Por exemplo, se o primeiro sorteio retornar 5, o contador de sorteios para 5 passará de 0 para 1. Se o segundo obtiver 2, o contador para 2 subirá de 0 para 1. Caso o terceiro retorne novamente 5, incrementa-se mais uma vez o contador para 5, que ficará agora com 2. E assim por diante. Prefira usar um vetor a variáveis individuais.
- Após realizar o sorteio, crie uma *string* que mostrará a quantidade de ocorrências para cada número de 1 a 10. Ao invés de mostrar a quantidade de vezes que cada um foi sorteado, imprima uma quantidade de cerquilhas equivalente. No exemplo da figura a seguir, o número 1 foi sorteado 13 vezes; já o número 4, apenas quatro vezes.



- **Dica:** o “gráfico” é apenas uma *string* exibida com JOptionPane. Monte-a usando a classe StringBuilder.
- **Dica:** para ocasionar uma quebra de linha em uma *string*, concatene a ela um “\n”. Você precisará disso para mostrar a frequência de cada número na sua própria linha.

## Módulo 2 – Classes e objetos em Java

A API do Java SE é riquíssima em número de classes prontas. As que vimos até agora são um pequeno exemplo. Mas, mesmo contando com esse conjunto, a linguagem não traz classes relativas ao domínio de um sistema, para produtos ou clientes em um sistema de vendas, por exemplo. O que Java contém são classes que auxiliarão o desenvolvedor a criar as suas, relativas à implementação do modelo conceitual de um sistema projetado por meio da engenharia de requisitos e modelagem UML. Neste módulo, começaremos a implementar em Java classes relativas ao sistema bancário, para que você tenha o entendimento essencial de como transportar o modelo conceitual para a programação.

### Aula 4 – Conceitos básicos de classes e objetos

#### Classes, objetos, métodos e instância

Você já sabe que, em orientação a objetos, os blocos fundamentais de um sistema são suas *classes*. Elas servem como moldes para os objetos, que são criados instanciando-as.

O *objeto* é como uma cópia ou instância de uma classe, mas que possui um estado (também chamado de conjunto de atributos, propriedade ou campo) que dá a ele “vida própria” e características específicas. Além do estado, o objeto também possui operações (também chamadas de ações ou *métodos*) que foram recebidas de sua classe e que permitem que realize suas funções.

Em um sistema bancário, certamente teríamos objetos representando seus clientes. Assim, concluímos que precisamos de um molde para eles: uma classe *Cliente*.

#### Convenções de código Java para classes



Você já sabe que, em Java, devemos nomear as classes com a primeira letra maiúscula e as demais minúsculas. Mas também precisa ficar sabendo que:

1. Nomes de classe devem ser definidos no singular (Cliente é o correto, e não Clientes)
2. Para classes com duas ou mais palavras no nome, capitalize sempre a primeira letra de cada palavra. Exemplo: ContaCorrente.

A classe Cliente terá, por enquanto, três campos (o nome pelo qual chamamos os atributos em Java) e um método (que é como chamamos as operações). Veja o diagrama UML para Cliente na figura 27.

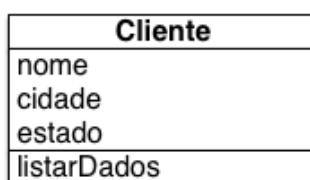


Figura 27 - Diagrama para a classe Cliente. Fonte: Produção do autor.



Se você olhou para a figura 27 e percebeu a falta de alguma coisa, tem toda a razão: não colocamos os símbolos que especificam a visibilidade dos campos (público, privado ou protegido). Vamos deixar a classe sem eles por enquanto - falaremos mais sobre a visibilidade em breve.

Aí vem a pergunta: como vamos criar essa classe em Java? Os campos serão variáveis dentro dela. Já o método será uma função semelhante à main() que utilizamos até agora.

Com relação às variáveis, há um detalhe: as que criamos até o momento eram colocadas dentro de main(). Isso faz delas locais ao método (ou seja, elas só existem e são visíveis dentro dele). Por isso, não são consideradas campos das classes. As variáveis que representarão os campos são colocadas fora de seus métodos. Geralmente, elas são declaradas no início da classe, logo após a chave de abertura.

A declaração básica de uma classe em Java obedece à sintaxe do quadro 14.

**Quadro 14** - Sintaxe de declaração de uma classe em Java

```
class NomeDaClasse {  
    tipoDoCampo1 nomeDoCampo;  
    ...  
    tipoDoCampoN nomeDoCampoN;  
  
    tipoDoRetornoDoMetodo1 nomeDoMetodo1(parametrosDoMetodo1) {  
        comando1;  
        comando2;  
        ...  
    }  
}
```

Uma classe pode conter quantos campos precisar. Cada objeto instanciado a partir dela possuirá seus próprios valores armazenados em cada um dos campos. Por essa razão, eles também são chamados de variáveis de instância.

O tipo de um campo pode ser um tipo primitivo (*int, double, char, boolean*), uma classe Java (*string, random...*) ou até mesmo classes criadas por você dentro do seu programa.

No nosso exemplo do Cliente, os campos nome, cidade e estado podem ser declarados como *string*.

## Convenções de código Java para variáveis



Nomes de variáveis (tanto as locais em um método quanto as que são campos de uma classe) devem começar com letras minúsculas, capitalizando as demais palavras.

Também devem ser evitados ao máximo nomes resumidos, que prejudiquem a legibilidade do código. Exemplos:

```
// Correto! É um bom nome de variável!
```

```
double saldoChequeEspecial;
```

```
// Errado! Não use maiúscula no início do nome - confunde com as classes!
```

```
double SaldoChequeEspecial;
```

Os métodos são como funções em linguagens de programação estruturadas, como C. Eles executam um bloco de comandos colocados dentro de suas chaves (delimitadores de início e fim) e podem retornar um valor, que pode ser um dos tipos primitivos da linguagem ou objetos de classes. Métodos que não retornam valores devem ser especificados com o tipo de retorno *void*.

Podemos ter um método em um objeto chamado calculadora (que pode ser, por sua vez, uma instância da classe Calculadora), por exemplo, que retorna um resultado de um fatorial. Chamamos a execução colocando um ponto após o nome do objeto e escrevendo seu nome, como no exemplo:

```
int resultado = calculadora.calcularFatorial(6);
```

Nesse exemplo, chamamos o método calcularFatorial existente no objeto calculadora. Passamos a ele o número 6 como parâmetro. Dentro do corpo do método, será feito o cálculo do fatorial desse valor, e o resultado será retornado para ser guardado na variável resultado, do tipo *int*.

No caso da classe Cliente na figura 27, especificamos um método chamado listarDados, que vai trazer uma ficha cadastral do cliente. Nesse caso, então, podemos assumir um retorno do tipo *string* para ele.

*Uma classe pode conter vários métodos.* Todo objeto instanciado a partir dela também os possuirá e os poderá utilizar. Dentro de um método, podem ser usadas variáveis locais (declaradas e válidas apenas dentro do método), valores passados ao método como parâmetro (válidos também somente dentro do método) e campos da classe (lembrando que cada objeto instanciado possuirá seus próprios valores para eles) nos quais o método está definido. Os argumentos ou parâmetros de um método, assim como seu retorno e variáveis locais, também podem ser tipos primitivos ou objetos.

## Implementando a classe no sistema bancário

Para criar nossa classe Cliente, vamos começar a implementação do nosso sistema bancário. Ele terá inicialmente duas classes: *Principal* e *Cliente* (mesmo tendo outra classe, vamos continuar precisando de uma classe Principal com um método main para termos um ponto de entrada e uma interface com o usuário).

### O que são pacotes?

Pacotes são usados em Java para organizar classes e demais identificadores, além de possibilitar que duas ou mais classes tenham o mesmo nome (desde que pertençam a pacotes distintos). Podemos nomeá-los usando subníveis, dividindo-os com pontos. Neste exemplo, nosso sistema possui dois subníveis de pacotes: modelo e tela, ambos pertencentes a banco.

Siga os passos:

- 1) Crie um novo projeto Java no Eclipse chamado SistemaBancario (sem acento e espaços no nome).
- 2) Após criar o projeto, clique sobre a pasta src e crie um pacote chamado banco.tela. Em seguida, adicione um segundo pacote, chamado banco.modelo.
- 3) No pacote banco.modelo, colocaremos a classe Cliente. O termo modelo é usado quando nos referimos aos elementos de um programa relativos aos seus dados principais.
- 4) Já o pacote banco.tela vai conter a classe Principal, que por sua vez possuirá o método main. Além de porta de entrada ao sistema, essa classe e esse método atuarão como a interface do aplicativo com o usuário. Por essa razão, ela será colocada em um pacote próprio, que representa melhor sua finalidade.
- 5) Crie a classe Principal (clique sobre o pacote banco.tela antes de criá-la, para que ela seja

colocada dentro dele) clicando no botão  da barra de ferramentas do Eclipse. Na janela que será aberta, especifique o nome em Name e não deixe de marcar a opção *public static void main* durante a criação para que tenhamos uma função main dentro dela.

- 6) Em seguida, clique sobre o pacote banco.modelo e crie uma segunda classe. Dessa vez, dê a ela o nome Cliente e não marque a opção *public static void main*.
- 7) Terminando essas etapas, a estrutura do seu projeto deverá ficar semelhante à exibida na figura 28.

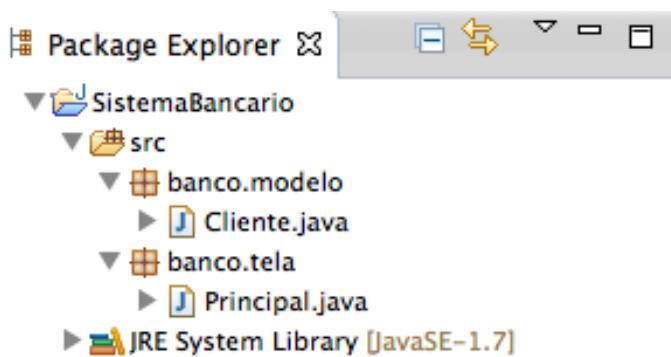


Figura 28 - Estrutura inicial do projeto SistemaBancario.

Agora, clique duas vezes sobre a classe Cliente.java e vamos escrever seu código, de acordo com a listagem mostrada no quadro 15.

### Quadro 15 - Código-fonte da classe Cliente

```

1 package banco.modelo;
2
3 public class Cliente {
4
5     String nome;
6     String cidade;
7     String estado;
8
9     String listarDados() {
10        return "NOME: " + nome + "\n" +
11                    "CIDADE: " + cidade + "\n" +
12                    "ESTADO: " + estado;
13    }
14
15 }
```

Vamos agora entender o que está sendo feito:

- **Linha 1:** está indicado que a classe que será definida a seguir pertence ao pacote banco.modelo.
- **Linha 3:** nessa linha, inicia-se o código da classe. Veja que a chave aberta ao final dela é fechada apenas na linha 15. Tudo que estiver contido dentro desse par de chaves pertence à classe. Aí a iniciamos indicando que ela é pública (public). A palavra public determina que poderá ser usada por qualquer outra. Depois de public, class Cliente indica que estamos definindo uma classe chamada Cliente.
- **Linhas 4 a 7:** como tínhamos colocado no diagrama da classe Cliente, declaramos aqui os três atributos que ela possui. Veja que todos são do tipo *string*, adequado para o tipo de informação que conterão. Todo objeto instanciado da classe Cliente possuirá seus próprios valores para esses campos.
- **Linhas 9 a 13:** definimos um único método para essa classe. Sua função é retornar uma *string* listando os dados do cliente. A palavra *string*, no início da linha 9 antes do nome do método, indica que ele retornará um objeto desse tipo. Veja que ele se chama obterDados e, nesse caso, não recebe parâmetros (ou seja, nenhum valor precisará ser passado a ele quando for executado). A chave no final da linha 9 indica o início do método, que é encerrado pela outra na linha 13.

No caso de métodos que retornam valores (ou seja, aqueles que não possuem void na sua declaração), devemos usar o comando return para devolvê-los a quem os invocou. Nesse exemplo, montamos uma *string* concatenando os dados do cliente (obtidos das variáveis locais do objeto: nome, cidade e estado) e retornamos para que possa então ser impressa ao usuário. Note que o método listarDados() pode acessar os campos, pois está contido na mesma classe.

Agora vamos colocar nossa nova classe em uso. Abra a classe Principal.java e escreva dentro do seu método main a seguinte linha:

```
Cliente cliente = new Cliente();
```

Inicialmente, aparecerão sublinhados vermelhos no lugar em que escrevemos Cliente. Isso ocorre pois nossa classe Principal não a reconhece. Como ela está em um pacote diferente, devemos acrescentar um import para ela.

Você pode fazer isso de duas formas: usando a ajuda do Eclipse ou manualmente, adicionando a linha seguinte ao código, abaixo do comando package banco.tela e antes do início da classe:

```
import banco.modelo.Cliente;
```

Perceba que, assim que você terminar de escrever o comando, o erro na linha anterior irá desaparecer.

Agora que não temos mais erros, vamos entender o que fizemos. Você simplesmente acabou de declarar e instanciar (ou seja, criar) um objeto de uma classe sua. A sintaxe básica para declarar um objeto é a mesma da declaração de variáveis:

```
NomeDaClasse nomeDoObjeto;
```

A diferença aqui é que declaramos e instanciamos o objeto em uma única linha. O instanciamento é feito pelo operador *new* e pelo método construtor.

O *método construtor* é um comando especial da classe, que serve para inicializá-la e que sempre tem o mesmo nome da classe. Poderíamos também ter escrito a linha:

```
Cliente cliente = new Cliente();
```

...da seguinte forma, com o mesmo resultado:

```
Cliente cliente;
cliente = new Cliente();
```

Geralmente, é usada a primeira forma, que deixa o código mais sucinto. É ela que usaremos nos nossos exemplos.

Agora, faça um teste: na linha seguinte, escreva o nome do objeto (cliente) seguido de um ponto ( . ) e aguarde a exibição da janela *pop-up* do autocompletar. Você deverá ver algo semelhante ao mostrado na figura 29.

```
Cliente cliente = new Cliente();
```

```
cliente.
```

```
}
```

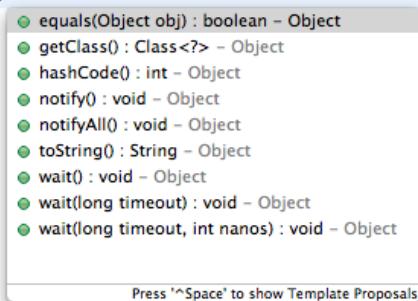


Figura 29 – Autocompletar ativo para o objeto cliente. Fonte: Produção do autor.

Provavelmente você esperava ver os membros do objeto cliente que ele possui, já que é uma instância da classe Cliente. Esses membros seriam os campos nome, cidade e estado e o método listarDados. Mas o que você encontra? Um monte de nomes estranhos. Mas não há nada de errado. Temos duas questões importantes aqui:

- ✓ Você não está visualizando os campos e os métodos da classe Cliente pois, como não definimos a visibilidade deles, foi assumido um valor padrão. Conhecido no Java como package private, indica que um campo ou método sem especificador de visibilidade só poderá ser acessado por classes pertencentes ao mesmo pacote. Infelizmente, não é o caso aqui: Clien-

te pertence ao pacote banco.modelo e Principal, ao pacote banco.tela. Resumindo: acesso negado a campos e método.

- ✓ Toda classe criada em Java sem uma superclasse (que é o caso aqui, por enquanto) herda automaticamente a chamada “classe raiz” da linguagem, denominada *Object* e pertencente ao pacote java.lang. É dela que vêm esses métodos “estranhos” exibidos na figura 29.

Para poder usar os campos e o método existentes em cliente de dentro da classe Principal, teremos que colocar um modificador de visibilidade diferente neles. Abra o código da classe Cliente e acrescente a palavra *public* no início da declaração das variáveis e do método, como indicado na listagem do quadro 16.

#### Quadro 16 - Listagem da classe Cliente com modificadores de visibilidade nos seus membros

```
1 package banco.modelo;
2
3 public class Cliente {
4
5     public String nome;
6     public String cidade;
7     public String estado;
8
9     public String listarDados() {
10        return "NOME: " + nome + "\n" +
11            "CIDADE: " + cidade + "\n" +
12            "ESTADO: " + estado;
13    }
14
15 }
```

Agora sim volte ao Principal.java, coloque o cursor ao lado do ponto em cliente e pressione as teclas CTRL+ESPAÇO para ativar o autocompletar. Você verá os campos e o método disponíveis para uso.

Vamos colocar valores nos campos e executar o método listarDados. Adicione ao seu método main o código novo exibido na listagem do quadro 17.

**Quadro 17** - Classe Principal.java

```
1 package banco.tela;
2
3 import javax.swing.JOptionPane;
4
5 import banco.modelo.Cliente;
6
7 public class Principal {
8
9     public static void main(String[] args) {
10
11         Cliente cliente = new Cliente();
12
13         cliente.nome = "Anita";
14         cliente.cidade = "Marília";
15         cliente.estado = "São Paulo";
16
17         JOptionPane.showMessageDialog(null, cliente.listarDados());
18
19     }
20 }
```

Ao executar, você deverá obter uma janela semelhante à mostrada na figura 30.

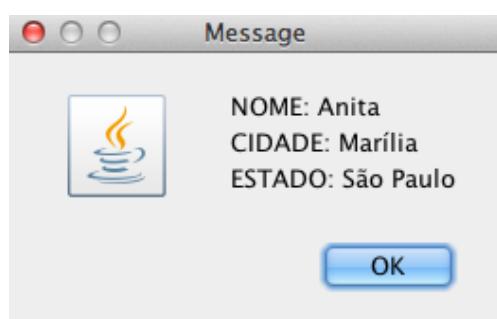


Figura 30 - Tela de SistemaBancario em execução. Fonte: Produção do autor.

## Métodos construtores

### **Garbage collector**



Toda vez que criamos um objeto com new, um espaço de memória é reservado para ele. Quando ele não é mais usado, esse espaço deve ser liberado, por motivos óbvios de desempenho e economia de recursos do computador.

Nos primórdios das linguagens orientadas a objetos, o programador deveria se preocupar com esse fato: era obrigação dele “jogar fora” um objeto que não estava mais sendo usado, para liberar memória. Com o tempo, percebeu-se que essa prática prejudicava a produtividade, além de poder causar erros nos programas (por exemplo, quando um programador colocava uma instrução para excluir um objeto quando ele ainda estava sendo usado).

Foram criadas técnicas para resolver essa questão, e Java, como boa linguagem moderna orientada a objetos, implementa uma delas: a coleta de lixo.

Um elemento importante da máquina virtual do Java, o coletor de lixo (garbage collector, literalmente, lixeiro) procura por objetos não mais utilizados no programa para que possam ser “jogados no lixo”, liberando a memória por eles usada. Assim, você só precisa se preocupar em criar o objeto (usando o new). Deixe a preocupação de excluí-lo com o “lixeiro” do Java.

Mas atenção: isso não significa que você pode “abusar”. Evite ao máximo instanciar objetos sem necessidade, para não prejudicar o desempenho do aplicativo e do sistema no qual ele está sendo executado.

Um *método construtor* serve para inicializar os dados de um objeto recém-criado. No exemplo da classe Cliente, não definimos um método construtor. Isso significa que o compilador Java vai criar um “construtor vazio” automaticamente na nossa classe, com o seguinte conteúdo:

```
public Cliente() {  
}
```

Repare em alguns detalhes:

- ✓ O construtor é público (e é assim na maioria das vezes, apesar de que você pode criá-lo com outro tipo de visibilidade).
- ✓ Um construtor nunca possui retorno (nem mesmo void).
- ✓ Seu conteúdo, nesse caso (dentro de chaves), está vazio.

Isso faz com que um objeto seja criado com seus campos totalmente vazios. Se quisermos, podemos definir esse mesmo construtor dentro da classe Cliente colocando valores *default* nos campos. Suponha, por exemplo, que a maioria dos clientes do banco seja de São Paulo, SP. Podemos então já iniciar o objeto com esses valores, como exemplificado no quadro 18.

**Quadro 18** - Construtor padrão com inicialização de campos

```
public Cliente() {
    nome = "INDEFINIDO";
    cidade = "São Paulo";
    estado = "São Paulo";
}
```

A partir de agora, o construtor *default* vazio não será mais usado, e sim esse do quadro 18.

Podemos também escrever mais de um construtor, sempre com o mesmo nome da classe, mas usando parâmetros diferentes. Isso se chama *sobrecarga de métodos* em Java.

Por exemplo, podemos manter esse construtor padrão sem parâmetros, mas termos também um segundo no qual passamos os dados iniciais do cliente como parâmetros para que o objeto seja criado já com eles, sem necessidade de definirmos um campo de cada vez. Para fazermos isso, adicione, dentro da sua classe Cliente, o novo construtor exibido no quadro 19.

**Quadro 19** - Novo construtor com parâmetros

```
public Cliente(String nome, String cidade, String estado) {
    this.nome = nome;
    this.cidade = cidade;
    this.estado = estado;
}
```

Agora podemos usar qualquer um dos dois construtores. Se instanciarmos um objeto usando o sem parâmetros, devemos fazer desta forma:

```
Cliente cliente = new Cliente();
```

Assim, teremos um objeto cliente com os valores padronizados para os campos (por exemplo, com cidade igual a São Paulo). Podemos modificar esses valores depois usando a atribuição direta. Mas podemos economizar tempo usando agora o segundo construtor da seguinte maneira:

```
Cliente cliente = new Cliente("Anita", "Marília", "São Paulo");
```

Nesse exemplo, o objeto cliente já começará com os valores para nome, cidade e estado, sem necessidade de atribuição manual para cada um deles.

O que significa o *this* no construtor?

Repare que os parâmetros recebidos por esse construtor mostrado no quadro 19 (que vão trazer os valores iniciais para o objeto) possuem o mesmo nome dos campos da classe.

Se dentro do código do construtor fizéssemos simplesmente

```
nome = nome;
```

surgiria uma ambiguidade: estamos atribuindo quem a quem? O argumento ao campo ou o campo ao argumento?

Usamos o *this* para eliminar a ambiguidade. A variável especificada com *this* será considerada como o campo da classe (*this* quer dizer este/esta - as variáveis existentes nesta classe). Assim, a atribuição será feita no sentido correto.

## Métodos estáticos

Em alguns casos, não há necessidade de instanciarmos as classes em objetos para usarmos seus métodos. Exemplos dessa situação que você já conhece são os métodos de conversão de *string* para *int* e *double*: *Integer.parseInt* é uma chamada a um método estático da classe *Integer*, assim como *Double.parseDouble* invoca outro método estático, agora da classe *Double*. Geralmente, métodos estáticos são usados como métodos utilitários, pois não dependem da existência de um objeto da classe.

Você pode até possuir métodos estáticos junto de não estáticos (*listarDados*, que criamos em *Cliente*, é um destes) na mesma classe, mas os primeiros não podem acessar variáveis de instância (ou seja, os campos) de um objeto. Para que um método estático possa usar um campo da classe, este deve também ser configurado como estático.

Um campo ou método é definido como estático colocando a palavra-chave *static* na sua declaração. Ao definir uma variável de uma classe como *static*, você está dizendo que ela manterá um valor único - não existirão cópias específicas para cada objeto.

Por exemplo, poderíamos criar na nossa classe *Cliente* um campo estático do tipo *int* chamado *quantidade*, que começará com zero e será incrementado em um toda vez que um novo cliente for instanciado. A existência desse campo abre também mais duas possibilidades interessantes:

- Podemos usá-lo para definir um código único para cada cliente instanciado.
- Podemos criar também em *Cliente* um método estático que retorna o valor desse campo *quantidade* - assim, poderemos saber a qualquer momento quantos objetos *Cliente* foram criados.

Adapte sua classe *Cliente* para que contenha o código apresentado no quadro 20.

## Quadro 20 - Classe Cliente com campo e método estático

```

1 package banco.modelo;
2
3 public class Cliente {
4
5     public int codigo;
6     public String nome;
7     public String cidade;
8     public String estado;
9
10    static int quantidade;
11
12    public Cliente() {
13        quantidade++;
14        codigo = quantidade;
15        nome = "INDEFINIDO";
16        cidade = "São Paulo";
17        estado = "São Paulo";
18    }
19
20    public Cliente(String nome, String cidade, String estado) {
21        quantidade++;
22        codigo = quantidade;
23        this.nome = nome;
24        this.cidade = cidade;
25        this.estado = estado;
26    }
27
28    public String listarDados() {
29        return "CÓDIGO: " + codigo + "\n" +
30               "NOME: " + nome + "\n" +
31               "CIDADE: " + cidade + "\n" +
32               "ESTADO: " + estado;
33    }
34
35    public static int qtdClientes() {
36        return quantidade;
37    }
38
39 }
```

Vamos entender o que está sendo feito:

- Como agora teremos um campo para o código do cliente, ele é declarado na linha 5.
- Na linha 10, temos o campo estático do tipo *int* chamado *quantidade*. Essa variável vai guardar na classe a quantidade de clientes já cadastrados. Não precisamos inicializá-la, pois, em Java, variáveis *int* já começam contendo 0.
- Na linha 13, dentro do primeiro construtor (sem parâmetros), incrementamos a quantidade em 1. Isso significa que, quando esse construtor for executado, terá sido realizada a criação de um novo cliente. Na linha 14, aproveitamos e usamos o valor atual de *quantidade* como código do cliente. Repetimos isso no outro construtor, nas linhas 21 e 22.
- Como agora temos uma informação adicional de um cliente (seu código), acrescentamo-la ao retorno de *listarDados()*, na linha 29.
- Ao final, nas linhas 35 a 37, implementamos um método estático -*qtdClientes()* - que retorna a quantidade de clientes guardada na variável.

Para poder utilizar os novos recursos da classe Cliente, adapte sua classe Principal de acordo com a listagem do quadro 21.

#### Quadro 21 - Classe Principal.java

```

1 package banco.tela;
2
3 import javax.swing.JOptionPane;
4
5 import banco.modelo.Cliente;
6
7 public class Principal {
8
9     public static void main(String[] args) {
10
11         Cliente cliente1 = new Cliente("Anita", "Marília", "São Paulo");
12         Cliente cliente2 = new Cliente("Marcos", "Garça", "São Paulo");
13
14         JOptionPane.showMessageDialog(null, cliente1.listarDados());
15         JOptionPane.showMessageDialog(null, cliente2.listarDados());
16
17         JOptionPane.showMessageDialog(null, "Possuímos " +
18             Cliente.qtdClientes() + " cliente(s) cadastrados");
19     }
20 }
```

Execute seu projeto e veja o resultado.

#### Guarde bem seu sistema bancário!



Usaremos daqui em diante o sistema bancário em todos os próximos exemplos (inclusive os exercícios). Por isso, não se esqueça de guardá-lo com cuidado. Faça sempre um backup de seu código!

#### Exercício prático - complementando o sistema bancário

- Crie uma classe Conta, dentro do pacote banco.modelo, com os campos numero, cliente e saldo. O campo cliente será um objeto da classe Cliente.
- Escreva um construtor para criar um objeto conta passando apenas um objeto cliente como parâmetro. Nesse construtor, atribua como número da conta um número inteiro sequencial gerado automaticamente dentro da classe Conta. Atribua zero para o saldo inicial.
- Inclua em Conta um método para exibição dos dados dela, contendo: número, nome do correntista e saldo.
- Em seguida, implemente na função main do sistema:

Três telas para solicitar as três informações do cliente: nome, cidade e estado.

Uma tela para exibição da ficha de dados do cliente (você já tem esse comando no seu código, se seguvi os exemplos anteriores).

Criação do objeto conta para esse cliente (não se esqueça de atribuir o objeto cliente criado ao respectivo campo da conta) e exibição dos dados da conta logo em seguida, usando o método listarDados de conta.

## Aula 5 - Encapsulamento

### Definição e exemplo de encapsulamento

O exercício que implementamos na aula anterior não está incorreto, mas possui um detalhe que pode ser melhorado para que se encaixe melhor ainda na “vida real”. Você pode ter percebido que o campo saldo foi declarado como público na classe Conta. Esse fato implica que, a qualquer momento, o usuário de um objeto da classe Conta pode fazer o seguinte:

```
conta.saldo = 1000000000000;
```

Ou seja, o dono da conta pode modificar o saldo da conta quando quiser, para qual valor desejar, o que com certeza é péssimo para o banco, o dono do sistema. Mas é principalmente ruim porque fere um princípio importantíssimo da orientação a objetos: o *encapsulamento*.

O encapsulamento determina que os detalhes internos da implementação de uma classe devem ficar ocultos ao usuário (ou seja, encapsulados), de forma que ele não possa prejudicar o seu funcionamento ou sua lógica interna.

O exemplo da mudança arbitrária do saldo de uma conta é um exemplo de quebra do encapsulamento. Sabemos que, em sistema bancário real, isso não pode ocorrer. O saldo só pode ser modificado mediante o uso de transações, principalmente saque e depósito, e o banco (assim como o cliente) precisa que elas fiquem registradas para que se possa controlar a movimentação da conta.

### Modificadores de visibilidade: private, public e protected

Para ocultar ou limitar o acesso aos elementos de uma classe (tanto campos, como o saldo no exemplo bancário, quanto métodos), usamos os modificadores de visibilidade. Em Java, temos as seguintes opções:

- **public**: o acesso ao membro da classe é irrestrito. Qualquer outra classe pode ter acesso livre a ele. Isso implica, no caso dos campos, poder obter e modificar seus valores.
- **private**: é o modificador mais restritivo, pois oculta totalmente o campo ou método do “mundo exterior”. Campos e métodos private só podem ser usados dentro da própria classe em que são definidos.

- **protected**: permite o acesso ao membro dentro da sua própria classe (assim como `private`) e também nas suas subclasses (classes criadas a partir dela - falaremos sobre herança na próxima aula).
- **Sem modificador**: quando não colocamos um modificador na declaração do campo ou método, o Java assume-o como `package private`. Isso significa que o membro só pode ser acessado dentro da própria classe e por outras classes pertencentes ao mesmo pacote.

Deve-se evitar ao máximo definir campos de uma classe como `public`, pois assim se abre mão totalmente do controle do que pode ser colocado dentro dela. Para seguir o princípio do encapsulamento, o ideal é definir os campos sempre como `private` ou `protected` (neste último caso, para quando se for criar uma subclasse que precisará usar um campo da sua superclasse).

Mas, se colocarmos `private` em um campo, não poderemos atribuir um valor a ele, nem mesmo acessar seu conteúdo. Se você abrir a classe `Cliente` e definir os campos como `private`, como no quadro 22, perceberá que o Eclipse indica agora vários erros no código da classe `Principal` (veja quadro 23).

**Quadro 22** - Trecho inicial da classe `Cliente` com os campos modificados com `private`

```

1 package banco.modelo;
2
3 public class Cliente {
4
5     private int codigo;
6     private String nome;
7     private String cidade;
8     private String estado;
9
10    private static int quantidade;
```

**Quadro 23** - Trecho do código da classe `Principal` em que surgem os erros

```

21     Cliente cliente = new Cliente();
22     cliente.nome = JOptionPane.showInputDialog(null, "Nome do Cliente: ");
23     cliente.cidade = JOptionPane.showInputDialog(null, "Cidade do Cliente: ");
24     cliente.estado = JOptionPane.showInputDialog(null, "Estado do Cliente: ");
```

Não se preocupe. Faça essa mudança para que depois possamos contornar esse problema de maneira muito mais eficiente do que simplesmente definir o campo como `public`: usando os métodos `getters` e `setters`.

## Usando getters e setters

Um método `getter` - que poderia ser traduzido como “pegador” - é um método público existente dentro de uma classe que é usado para outra obter um valor interno da primeira, ou seja, ele *retorna um campo private*.

É um redirecionador: ao invés de obtermos o valor do campo diretamente (se for público), usamos o `getter` como intermediário. Como? Adicione *apenas* as linhas 28 a 30 da listagem exibida no quadro 24 ao seu arquivo `Cliente.java`. Você pode adicionar o método em qualquer ponto da classe, mas a convenção mais usada é colocá-lo após o(s) construtor(es).

#### Quadro 24 - Classe Cliente com método getter para o campo nome

```

20 public Cliente(String nome, String cidade, String estado) {
21     quantidade++;
22     codigo = quantidade;
23     this.nome = nome;
24     this.cidade = cidade;
25     this.estado = estado;
26 }
27
28 public String getNome() {
29     return nome;
30 }
31
32 public String listarDados() {
33     return "CÓDIGO: " + codigo + "\n" +
34         "NOME: " + nome + "\n" +

```

Agora, quando uma outra classe precisar obter o nome de um cliente, deverá executar o método `getNome()` dentro do objeto. Veja na linha 29 que esse método retorna o conteúdo do campo nome.

Exemplo de uso:

```
String nomeDoCliente = cliente1.getNome();
```

#### Convenções de código Java: dando nome a um *getter*



É uma convenção de Java que métodos getter devem ter seu nome definido da seguinte forma:

- Começando com `get` em letras minúsculas.
- Seguindo com o nome do campo com sua primeira letra maiúscula.

Por exemplo, considere o seguinte campo:

```
private int idade;
```

Seu getter deverá se chamar `getIdade` (repare na letra I maiúscula), e seu código ficaria dessa forma:

```
public int getIdade() {
    return idade;
}
```

Você pode perguntar: qual a utilidade então de se usar um *getter*, já que o valor do campo será retornado de qualquer jeito?

Dessa maneira padrão, realmente não há mudança. Mas, dessa forma, o acesso ao campo fica controlado e, se quisermos, podemos dentro do *getter* determinar como o conteúdo deverá ser apresentado ao usuário. Por exemplo, podemos fazer uma formatação do valor antes de ele ser retornado.

Suponha que quiséssemos retornar apenas o primeiro nome do cliente (apesar de estar armazenado em *nome* seu nome completo). Podemos modificar o getter para o exemplo do quadro 25.

#### Quadro 25 - Implementação de *getter* para nome com retorno formatado

```
28 public String getName() {
29     return (nome.contains(" ")) ? nome.substring(0, nome.indexOf(' ')) : nome;
30 }
```

O método *substring*, existente em todo objeto da classe *String*, retorna uma parte dela existente entre o índice do primeiro parâmetro, 0 no exemplo, e o índice anterior ao segundo parâmetro. *nome.indexOf(' ')* será a posição dentro da *string* em que ocorre o primeiro espaço (se ele contiver um espaço; caso contrário, retornamos o valor completo - tudo isso testado com um operador ternário).

Para permitir a modificação de um valor de um campo oculto, usamos os métodos *setters* (poderíamos traduzir como “definidores” ou “modificadores”). Assim como os *getters*, os *setters* são geralmente definidos como públicos (*public*). Como nesse caso não há retorno de valor, colocamos a palavra chave *void* na sua declaração.

O valor que vai ser colocado no campo privado é passado como um parâmetro ao *setter*. Dentro do seu corpo, atribuímos o valor do parâmetro ao campo da classe. Veja no quadro 26, nas linhas 32 a 34, o *setter* para o nome do cliente, colocado dentro da classe *Cliente* logo após seu *getter*.

#### Quadro 26 - Trecho de código da classe *Cliente* com *setter* para o campo nome implementado

```
28 public String getName() {
29     return (nome.contains(" ")) ? nome.substring(0, nome.indexOf(' ')) : nome;
30 }
31
32 public void setName(String nome) {
33     this.nome = nome;
34 }
```

Perceba que, assim como fizemos no construtor que criamos para *Cliente*, usamos o mesmo nome do campo local da classe para o parâmetro (*nome*). Por isso, para não causar ambiguidade e definirmos que o campo local é que receberá o parâmetro (e não o contrário), usamos *this* na linha 33.

## Convenções de código Java: dando nome a um *setter*



Assim como nos getters, é uma convenção de Java que métodos *setter* devem ter seu nome definido da seguinte forma:

- Começando com set em letras minúsculas.
- Seguindo com o nome do campo com sua primeira letra maiúscula.

Por exemplo, considere o seguinte campo:

```
private double media;
```

Seu setter deverá se chamar setMedia (repare na letra M maiúscula), e seu código ficaria dessa forma:

```
public void setMedia(double media) {
    this.media = media;
}
```

Novamente, você pode achar que dessa maneira estamos apenas “trocando seis por meia-dúzia”. A forma default do *setter*, como implementada no quadro 26, realmente não faz nenhum controle do que é colocado em nome, mas permite que facilmente implementemos isso, sua grande vantagem, como no exemplo do quadro 27.

### Quadro 27 - Implementação de *setter* para o campo nome de Cliente com validação

```
32 public void setNome(String nome) {
33     if (nome.isEmpty())
34         this.nome = "NÃO FORNECIDO";
35     else
36         this.nome = nome;
37 }
```

No exemplo, antes de atribuir o parâmetro ao campo, o *setter* verifica se o primeiro foi passado vazio. Caso tenha sido, atribui “não fornecido” ao nome; caso contrário, coloca nesse campo o valor fornecido como argumento.

Agora que temos um *setter* para o campo nome, podemos começar a corrigir os problemas na classe Principal. Troque a linha em que colocávamos diretamente o valor retornado pelo JOptionPane no campo nome pela chamada ao *setter*, como na linha 23 da listagem do quadro 28.

### Quadro 28 - Classe Principal com uso do *setter* para o nome do cliente

```
21     Cliente cliente = new Cliente();
22     //cliente.nome = JOptionPane.showInputDialog(null, "Nome do Cliente: ");
23     cliente.setNome(JOptionPane.showInputDialog(null, "Nome do Cliente: "));
24     cliente.cidade = JOptionPane.showInputDialog(null, "Cidade do Cliente: ");
25     cliente.estado = JOptionPane.showInputDialog(null, "Estado do Cliente: ");
```

## Exercício adicional

Adicione *getters* e *setters* para os demais campos da classe Cliente e modifique a classe Principal para usá-los. Você notará que também há um erro na classe Conta. Consegue descobrir por quê? Após terminar, confirme o resultado nas listagens dos quadros 29, 30 e 31.

### Não crie getters e setters sem necessidade!



Você deve sempre deixar os campos das classes como private ou protected, mas nem sempre precisa criar getters e setters para eles. Apenas crie se for realmente usá-los, ou seja, se alguma classe externa precisar obter e/ou modificar os valores.

Por exemplo: na classe Cliente, não precisamos criar getters e setters para os campos código e quantidade, pois em nenhum momento em Principal ou Conta fazemos referência direta a esses valores.

### Quadro 29 - Trecho da classe Cliente com *getters* e *setters* necessários

```

28  public String getNome() {
29      return (nome.contains(" ")) ? nome.substring(0, nome.indexOf(' ')) : nome;
30  }
31
32  public void setNome(String nome) {
33      if (nome.isEmpty())
34          this.nome = "NÃO FORNECIDO";
35      else
36          this.nome = nome;
37  }
38
39  public String getCidade() {
40      return cidade;
41  }
42
43  public void setCidade(String cidade) {
44      this.cidade = cidade;
45  }
46
47  public String getEstado() {
48      return estado;
49  }
50
51  public void setEstado(String estado) {
52      this.estado = estado;
53  }

```

### Quadro 30 - Trecho da classe Principal usando *setters* para nome, cidade e estado

```

21  Cliente cliente = new Cliente();
22  cliente.setNome(JOptionPane.showInputDialog(null, "Nome do Cliente: "));
23  cliente.setCidade(JOptionPane.showInputDialog(null, "Cidade do Cliente: "));
24  cliente.setEstado(JOptionPane.showInputDialog(null, "Estado do Cliente: "));

```

### Quadro 31 - Trecho da classe Conta usando getter para o campo nome de cliente

```

20  public String listarDados() {
21      return "NÚMERO: " + numero + "\n" +
22          "CORRENTISTA: " + cliente.getNome() + "\n" +
23          "SALDO: " + DecimalFormat.getCurrencyInstance().format(saldo);
24  }

```

### Encapsulamento de campos na prática: exercício prático guiado

Agora, precisamos fazer algumas modificações no nosso sistema bancário, por três motivos:

- Para que ele fique ainda mais semelhante ao modelo UML.
- Para prepará-lo para a implementação prática do assunto da próxima aula: herança.
- Para podermos praticar mais um pouco o conceito de encapsulamento.

Vamos impedir que o usuário da classe Conta possa manipular o saldo diretamente. A partir de agora, ele deverá obrigatoriamente usar métodos para tanto (saque ou depósito). Futuramente, quando você for apresentado ao conceito de interfaces em Java, poderá transformar esses métodos saque e depósito em classes que implementam a interface Transação.

Deixaremos as classes do nosso sistema semelhantes às apresentadas na figura 31.

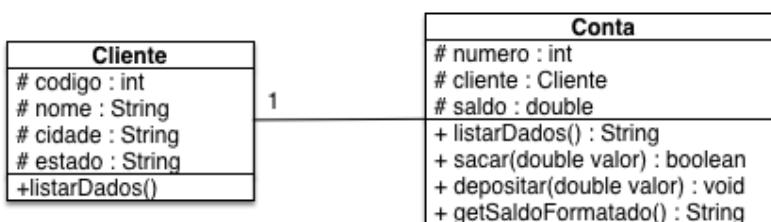


Figura 31 - Diagrama UML para as classes Cliente e Conta.

Veja que agora temos o símbolo indicando a visibilidade desejada para os campos e métodos, assim como seus tipos. Não foram colocados todos os métodos no diagrama (como *getters*, *setters* e construtores) para nos focarmos naquilo que precisaremos incluir ou modificar, ou seja:

- Como criaremos subclasses para Conta e Cliente na próxima aula, mudaremos a visibilidade de seus campos para *protected*.
- Para manipulação do saldo, deveremos implementar dois métodos públicos em Conta:
  - *sacar()*, que receberá um valor *double* de saque como parâmetro e somente o realizará (subtraindo o valor do parâmetro do saldo) se houver saldo suficiente. Note que esse método deverá retornar um *boolean*: *true*, se o saque foi realizado (havia saldo disponível), ou *false*, caso contrário (o saldo era insuficiente). Podemos usar esse retorno na interface para avisar ao usuário sobre a realização ou não.
  - *deposito()*, que receberá uma valor *double* de depósito como parâmetro. Nesse caso, não há retorno no método - basta adicionar o valor do parâmetro ao saldo.

Após movimentar sua conta, o usuário desejará ver como ficou seu saldo. Para isso, teremos o método *getSaldoFormatado*, que retornará uma *string* com o saldo da conta em formato monetário.

Antes de continuar, tente fazer essas mudanças sozinho, implementando um procedimento de depósito e saque na classe Principal. A seguir, teremos um passo a passo de como implementar o que foi pedido.

## 1º passo: adaptando a classe Cliente

Abra a classe Cliente e altere seu código seguindo o exemplo da listagem do quadro 32. Nesse caso, é bastante simples: basta mudar os modificadores de visibilidade dos campos. Repare que manteremos o campo estático quantidade como private; não é necessário colocá-lo como protected, e na próxima aula você descobrirá o porquê.

**Quadro 32** - Mudanças na classe Cliente

```

1 package banco.modelo;
2
3 public class Cliente {
4
5     protected int codigo;
6     protected String nome;
7     protected String cidade;
8     protected String estado;
9
10    private static int quantidade;
```

## 2º passo: adaptando a classe Conta

Vamos agora passar à classe Conta. Primeiro mudaremos os modificadores de visibilidade dos campos, como indicado no quadro 33.

**Quadro 33** - Modificações de visibilidade dos campos na classe Conta

```

1 package banco.modelo;
2
3 import java.text.DecimalFormat;
4
5 public class Conta {
6
7     protected int numero;
8     protected Cliente cliente;
9     protected double saldo;
10
11    private static int contador;
```

Veja que colocamos o contador como private, pela mesma razão de termos definido o de Cliente dessa forma.

O principal vem agora: a implementação dos métodos de obtenção do saldo formatado, de saque e de depósito. Acrescente-os logo no final da classe Conta, após o método listarDados(), de acordo com o quadro 34.

### Quadro 34 - Novos métodos na classe Conta

```

26 public String getSaldoFormatado() {
27     return DecimalFormat.getCurrencyInstance().format(saldo);
28 }
29
30 public void depositar(double valor) {
31     saldo += valor;
32 }
33
34 public boolean sacar(double valor) {
35     if (saldo >= valor) {
36         saldo -= valor;
37         return true;
38     } else {
39         return false;
40     }
41 }
42
43 }
```

Explicações do código do quadro 34:

- Como definimos o saldo como protected, não poderemos acessá-lo diretamente a partir da classe Principal. Seria interessante ao cliente visualizar seu saldo atual após uma operação de saque ou depósito. Por isso, adicionamos o método getSaldoFormatado(), definido nas linhas 26 a 28, cuja função é retornar uma *string* com o saldo da conta em formato monetário.
- O método depositar (linhas 30 a 32) é bastante simples: recebe um valor *double* como parâmetro e adiciona-o ao saldo da conta.
- Para realizar um saque, será usado o método sacar (linhas 34 a 41) que também recebe um valor como argumento. Porém o saque somente será possível se o saldo for superior ou igual ao valor passado. Nesse caso, subtraímos o valor do saque e retornamos *true*, sinalizando que a operação foi realizada com sucesso. Se o saldo for inferior, caímos no else da linha 34 - nesse caso, não é feita mudança no saldo, e o método retorna *false*.

### 3º passo: testando as implementações na classe Principal

Para testar a implementação, vamos adicionar à classe Principal, logo após a criação dos objetos cliente e conta, um pequeno “menu de movimentação” da conta, com opções para realizar um saque ou um depósito ou finalizar o programa.

Acrescente à classe Principal, logo após a exibição dos dados da conta instanciada, o código exibido no quadro 35.

### Quadro 35 - Código da classe Principal com menu de movimentação da conta

```

34 int opcao = 0;
35 String ret;
36 do {
37     String mensagem = "SALDO EM CONTA: " + conta.getSaldoFormatado() + "\n\n" +
38     "OPÇÕES: \n1 - Depositar valor\n2 - Sacar valor\n3 - Finalizar";
39     try {
40         opcao = Integer.parseInt(JOptionPane.showInputDialog(null, mensagem));
41         switch (opcao) {
42             case 1:
43                 ret = JOptionPane.showInputDialog(null, "Valor do depósito:");
44                 conta.depositar(Double.parseDouble(ret));
45                 JOptionPane.showMessageDialog(null, "Depósito realizado!");
46                 break;
47             case 2:
48                 ret = JOptionPane.showInputDialog(null, "Valor do saque:");
49                 if (conta.sacar(Double.parseDouble(ret))) {
50                     JOptionPane.showMessageDialog(null, "Saque realizado!");
51                 } else {
52                     JOptionPane.showMessageDialog(null, "FALHA NO SAQUE!");
53                 }
54             } catch (NumberFormatException ex) {
55                 JOptionPane.showMessageDialog(null, "VALOR INVÁLIDO!");
56             }
57         } while ((opcao == 1) || (opcao == 2));
58     }
59 }

```

Explicações do código do quadro 35:

- Linhas 34 e 35: declaramos duas variáveis auxiliares - uma *int* para a opção escolhida no menu e uma *string* para receber o valor digitado pelo usuário no JOptionPane.
- Linha 36: iniciamos um laço que termina na linha 58 e que se repetirá enquanto a opção de menu escolhida pelo usuário for 1 (depósito) ou 2 (saque).
- Linhas 37 e 38: montamos uma *string* com o menu de opções.
- Linha 39: abrimos um bloco try para tratar erros de conversão de valores. Caso o usuário forneça um valor que não possa ser convertido para número (nas opções menu ou valores), o fluxo de execução do programa será direcionado para o catch na linha 55, que exibe um aviso.
- Linha 40: usando um JOptionPane, solicitamos a opção do usuário. O retorno já é convertido para *int* e armazenado na variável opcao.
- Linha 41: abrimos um switch...case que é encerrado apenas na linha 54. Ele fará o tratamento adequado da opção do menu escolhida pelo usuário.
- Linhas 42 a 46: se a opção for igual a 1, o usuário escolheu fazer um depósito. Após obter o valor com um JOptionPane e convertê-lo para *double*, passamos para o método sacar existente no objeto conta, que faz a mudança no saldo. Avisamos o usuário de que o saque foi realizado com outro JOptionPane. O comando break logo em seguida impede que o fluxo de execução passe ao case seguinte.
- Linhas 47 a 53: no caso da opção 2 (saque), inicialmente solicitamos o valor desejado ao usuário e fazemos sua conversão para *double*. Como o método sacar retorna um *boolean* indicando seu o saque teve sucesso ou não, fazemos a chamada a ele diretamente na condição do if. Se o retorno for verdadeiro, o saque foi realizado; caso contrário, falhou. Em ambos os casos, avisamos o usuário sobre o ocorrido.

Assim, finalizamos juntos a tarefa de melhorarmos ainda mais nosso sistema bancário.

## Exercício do Módulo 2

Crie um aplicativo Java para um jogo de forca. Nele, você deverá ter uma classe Principal, na qual acontecerá a interação com o usuário, e uma classe Forca, em que ficarão os dados principais do jogo.

- Na classe Forca, você deverá implementar:
- Uma lista de palavras disponíveis para o jogo (um vetor de *strings*, por exemplo).
- Um contador de erros.
- Um construtor, em que será sorteada a palavra a ser adivinhada dentro da lista de palavras disponíveis.
- Um *getter* que trará uma representação do “estágio de enforcamento”, dependendo da quantidade de erros do usuário. Serão sete estágios, indo da forca vazia (nenhum erro) à completa (seis erros):
  - Sem erro: desenho da forca vazia.
  - Um erro: forca com cabeça.
  - Dois erros: forca com cabeça e tronco.
  - Três erros: forca com cabeça, tronco e braço direito.
  - Quatro erros: forca com cabeça, tronco, braço direito e braço esquerdo.
  - Cinco erros: forca com cabeça, tronco, braço direito, braço esquerdo e perna direita.
  - Seis erros: forca com cabeça, tronco, braço direito, braço esquerdo, perna direita e perna esquerda.
- Você pode exibir uma *string* contendo o desenho ASCII da forca diretamente em um JOptionPane, porém o desenho sairá “torto”, já que essas janelas não usam fontes de largura fixa. O ideal é usar um componente JTextArea junto do JOptionPane, pois pode ser configurado para uso de fonte de largura fixa.
- Na classe Principal, comece mostrando a forca vazia e um conjunto de traços representando a palavra sorteada. Por exemplo, se a palavra for JAVA, exiba \_ \_ \_ \_.
- Peça uma letra ao usuário.
- Veja se a letra informada existe dentro da palavra sorteada na classe Forca. Use métodos de objetos *string* para isso, como *contains()*.
- Se existir, mostre abaixo da forca os traços da palavra sorteada com a substituição do(s) traço(s) respectivos pela letra.

Caso não exista, incremente o contador de erros e mostre o novo estágio de enforcamento.

Se o usuário não descobrir até o último estágio, encerre o jogo e mostre a palavra secreta.

Veja a seguir um exemplo inicial da classe Forca e da classe Principal, para você começar sua implementação. Não se esqueça de que o exemplo mostra apenas o desenho do primeiro e do último estágio da forca. Você deverá “desenhar” os demais.

Logo após o exemplo de classe para a Forca, você encontrará também um exemplo de código que mostra como usar o JTextArea junto do JOptionPane, de modo que o desenho ASCII apareça alinhado corretamente.

```

package forca;

import java.util.Random;

public class Forca {

    private String[] palavras = new String[]{"JAVA", "COMPILADOR", "COMPUTADOR",
        "PROGRAMA", "DUKE", "ECLIPSE"};

    private String[] forcas = new String[7];

    private int erros;
    private int indicePalavraSorteada;

    public Forca() {
        erros = 0;
        Random random = new Random();
        indicePalavraSorteada = random.nextInt(palavras.length);

        forcas[0] = " ----- \n"
            + " |   | \n"
            + "     | \n"
            + "     | \n"
            + "     | \n"
            + "     | \n"
            + " ===\n";
    }
}

```

```

public class Principal {

    public static void main(String[] args) {

        Forca forca = new Forca();

        JTextArea textoForca = new JTextArea();
        textoForca.setFont(new Font("Monospaced", Font.BOLD, 20));

        textoForca.setText(forca.getEstagioEnforcamento());
        String letra = JOptionPane.showInputDialog(null, textoForca);
    }
}

```

```

    forcas[6] = " ----- \n"
        + " |   | \n"
        + " 0   | \n"
        + "/|\\" | \n"
        + "/ \\\" | \n"
        + "     ===\n";
    }

    public String getEstagioEnforcamento() {
        return forcas[erros];
    }
}

```

## Módulo 3 – Herança e polimorfismo

Neste módulo, conhiceremos como Java implementa dois dos mais poderosos recursos da programação orientada a objetos: herança e polimorfismo. Ao final das próximas duas aulas, você terá evoluído seu sistema bancário ainda mais dentro da modelagem realizada na disciplina de UML. Assim como na última aula, as implementações serão feitas na forma de exercícios guiados, que você poderá tentar realizar sozinho ou seguindo as orientações do material.

### Aula 6 – Herança em Java

#### Conceito de herança

A *herança* possibilita que você crie uma classe utilizando outra como base. É uma das formas mais frequentes de reutilização de código e uma maneira de tornar os programas mais organizados e fáceis de ser mantidos e atualizados.

Para criar uma relação de herança, estabelecemos primeiro uma generalização de um elemento de um sistema. Em seguida, criamos as versões especializadas desse elemento, que recebem então, de forma direta e automática, as características do primeiro.

Na figura 32, você pode verificar um diagrama de classes UML com dois exemplos de herança e generalização já adequados à implementação que estamos fazendo até este ponto.

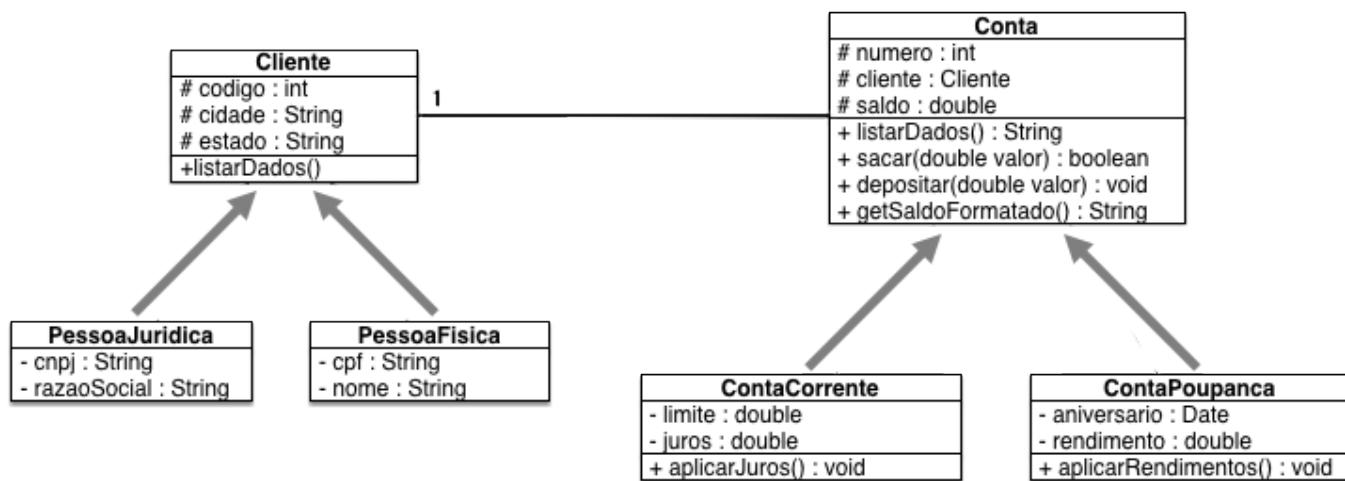


Figura 32 - Sistema bancário com herança de classes. Fonte: Produção do autor.

No caso de **Cliente**, criamos duas classes especializadas: uma para cliente pessoa física (denominada **PessoaFisica**) e outra para cliente pessoa jurídica (chamada **PessoaJuridica**). Deixamos essa classe ainda mais genérica, contendo apenas os campos comuns aos dois tipos (**codigo**, **cidade** e **estado**) - perceba que movemos o campo **nome**, específico de pessoas físicas, para a classe correspondente.

Assim, quando instanciarmos um objeto da classe PessoaFisica, teremos, além dos campos e métodos existentes em Cliente, campos adicionais para nome e CPF. Já quando criarmos um objeto da classe PessoaJuridica, receberemos, além do que já existe em Cliente, um campo para o CNPJ e outro para a razão social.

A classe *Conta* também é um exemplo que pode ser especializado com herança. Isso porque, em um banco, o cliente pode abrir tanto uma conta corrente quanto uma conta poupança, sendo que cada uma possui características próprias.

Observe na figura 32 que criamos uma classe *ContaCorrente*, que herda da classe Conta e que a complementa com os campos limite e juros, assim como traz o método *aplicarJuros()*. Já a classe *ContaPoupanca* também herda as características básicas da classe Conta, mas acrescenta campos para a data de aniversario (tipo classe *Date* de Java - um objeto dessa classe armazena uma data específica) e o rendimento, assim como um método para aplicar os rendimentos ao saldo (*aplicarRendimentos()*). Implementaremos as classes ContaPoupanca e ContaCorrente na próxima aula, quando tratarmos de polimorfismo em Java.

## Superclasses e subclasses

Existe uma terminologia para nos referirmos às classes participantes em um esquema de herança. Chamamos a classe mais genérica, aquela que possui as características que serão herdadas pelas demais, de *superclasse*. No exemplo da figura 32, Cliente é a superclasse.

As classes especializadas, que herdam da superclasse, são chamadas de *subclasses*. Portanto, PessoaFisica e PessoaJuridica são subclasses da classe Cliente.

Há a possibilidade de, dentro da subclass, utilizar recursos na superclasse, desde que sejam declarados como *protected* ou *public*. Para tanto, utiliza-se a palavra chave *super*.

Vamos agora implementar juntos no sistema bancário a herança para os dois tipos básicos de cliente.

## Exercício prático guiado: herança de cliente no sistema bancário

Vamos começar a implementação da nossa herança em Java generalizando mais a classe Cliente. Abra seu código-fonte e exclua a declaração do campo nome, assim como seu *getter* e seu *setter*. Também remova a utilização deste dentro dos dois construtores existentes na classe e seu uso dentro do método listarDados. Sua classe Cliente deverá ficar como a listagem exibida no quadro 36.

**Quadro 36** - Listagem da classe Cliente sem o campo nome

```

29  public void setCidade(String cidade) {
30      this.cidade = cidade;
31  }
32
33  public String getEstado() {
34      return estado;
35  }
36
37  public void setEstado(String estado) {
38      this.estado = estado;
39  }
40
41  public String listarDados() {
42      return "CÓDIGO: " + codigo + "\n" +
43          "CIDADE: " + cidade + "\n" +
44          "ESTADO: " + estado;
45  }
46
47  public static int qtdClientes() {
48      return quantidade;
49  }
50
51 }
```

```

1 package banco.modelo;
2
3 public class Cliente {
4
5     protected int codigo;
6     protected String cidade;
7     protected String estado;
8
9     private static int quantidade;
10
11    public Cliente() {
12        quantidade++;
13        codigo = quantidade;
14        cidade = "São Paulo";
15        estado = "São Paulo";
16    }
17
18    public Cliente(String cidade, String estado) {
19        quantidade++;
20        codigo = quantidade;
21        this.cidade = cidade;
22        this.estado = estado;
23    }
24
25    public String getCidade() {
26        return cidade;
27    }
28

```

Não se preocupe com erros que podem aparecer nas outras classes: vamos corrigi-los daqui a pouco.

Agora, vamos adicionar a classe PessoaFisica. Clique em cima do nome do pacote `banco.modelo` à esquerda, no Project Explorer, e clique sobre o botão *New Java Class* do Eclipse. Se não se lembra de como fazer isso, oriente-se pela figura 33.

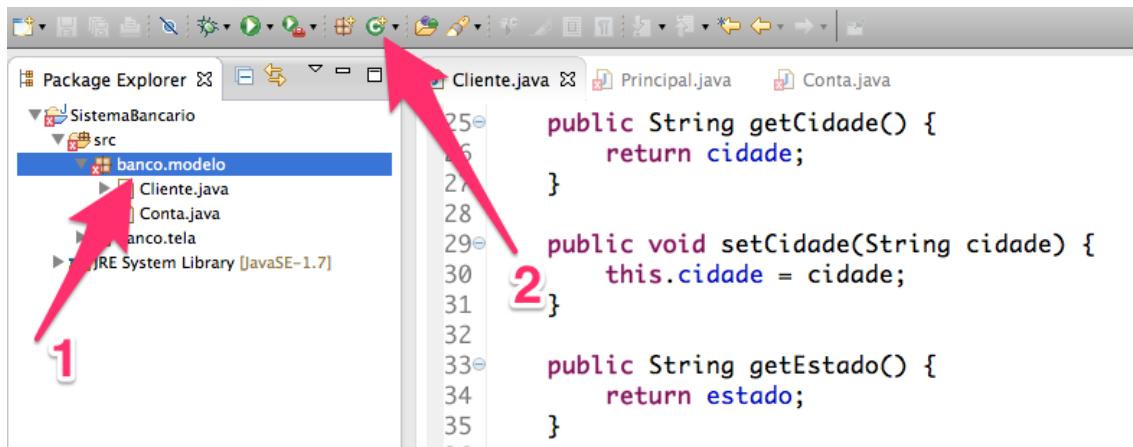


Figura 33 - Adicionando uma nova classe Java ao pacote `banco.modelo` do projeto. Fonte: Produção do autor.

Na janela que será mostrada, especifique o nome da classe como `PessoaFisica` em Name e clique em *Finish* para que a classe seja criada.

A classe recém-criada inicia-se sem especificação de herança (mas, mesmo assim, ela ainda herda a classe básica do Java, `java.lang.Object`). Nesse caso, precisamos especificar que `PessoaFisica` herda `Cliente`.

Em Java, a palavra-chave `extends` é usada para indicar a superclasse de uma classe, ou seja, de quem ela herda suas características iniciais. Para fazer isso, inclua o código `extends Cliente` na sua declaração, como na linha 3 do quadro 37.

**Quadro 37** - Classe PessoaFisica herdando a classe Cliente

```

1 package banco.modelo;
2
3 public class PessoaFisica extends Cliente {
4
5 }
```

Pronto, PessoaFisica passou a conter tudo que já existia em Cliente.

Agora, implemente o restante do código para PessoaFisica seguindo a listagem do quadro 38.

**Quadro 38** - Código completo da classe PessoaFisica

```

1 package banco.modelo;
2
3 public class PessoaFisica extends Cliente {
4
5     private String nome;
6     private String cpf;
7
8     public PessoaFisica() {
9         super();
10        nome = "INDEFINIDO";
11        cpf = "";
12    }
13
14    public PessoaFisica(String cidade, String estado, String nome, String cpf) {
15        super(cidade, estado);
16        this.nome = nome;
17        this.cpf = cpf;
18    }
19
20    public String getNome() {
21        return nome;
22    }
23
24    public void setNome(String nome) {
25        this.nome = nome;
26    }
27
28    public String getCpf() {
29        return cpf;
30    }
31
32    public void setCpf(String cpf) {
33        this.cpf = cpf;
34    }
35
36    public String listarDados() {
37        return "NOME: " + nome + "\n" + "CPF: " + cpf + "\n" + super.listarDados();
38    }
39 }
```

No quadro 38, temos em PessoaFisica os campos adicionais nome e cpf, assim como seus respectivos *getters* e *setters*. Até aí, nenhuma novidade. Mas veja os dois métodos construtores: um sem parâmetro, que cria o objeto com valores *default*, e outro em que o usuário pode passar os dados do cliente para que seja criado já com todas as informações necessárias.

Percebeu a novidade? O comando *super* do Java! Como seu próprio nome dá a entender, ele é usado para nos referirmos aos elementos existentes na nossa superclasse.

O comando *super()*, na linha 9 dentro do construtor PessoaFisica(), por exemplo, executa antes de tudo o construtor sem parâmetros existente na superclasse Cliente. Isso ocorre para inicializar também cidade e estado, assim como incrementar a quantidade de clientes criados e atribuir um código para eles.

Já na linha 15, dentro do construtor PessoaFisica (*String* cidade, *String* estado, *String* nome, *String* cpf), chamamos o segundo construtor de Cliente, aquele que recebe uma cidade e um estado como parâmetros. Fazemos isso pois também precisamos criar o cliente PessoaFisica com uma cidade e um estado, de acordo com aquilo que o usuário passou como argumentos no construtor dessa classe.

Veja que, nas linhas 36 a 38, implementamos uma nova versão de listarDados(), específica para clientes pessoa física. Ela vai trazer os dados específicos de PessoaFisica (nome e cpf) junto daqueles que já existiam em Cliente, obtidos pela chamada a esse método usando *super.listarDados*.

Para completar, vamos agora criar e escrever o código para PessoaJuridica. Siga a listagem do quadro 39.

### Quadro 39 - Código-fonte da classe PessoaJuridica

```

1 package banco.modelo;
2
3 public class PessoaJuridica extends Cliente {
4
5     private String cnpj;
6     private String razaoSocial;
7
8     public PessoaJuridica() {
9         super();
10        cnpj = "";
11        razaoSocial = "INDEFINIDA";
12    }
13
14     public PessoaJuridica(String cidade, String estado, String cnpj, String razaoSocial) {
15         super(cidade, estado);
16         this.cnpj = cnpj;
17         this.razaoSocial = razaoSocial;
18     }
19
20     public String getCnpj() {
21         return cnpj;
22     }
23
24     public void setCnpj(String cnpj) {
25         this.cnpj = cnpj;
26     }
27
28     public String getRazaoSocial() {
29         return razaoSocial;
30     }
31
32     public void setRazaoSocial(String razaoSocial) {
33         this.razaoSocial = razaoSocial;
34     }
35
36     public String listarDados() {
37         return "RAZÃO SOCIAL: " + razaoSocial + "\n" + "CNPJ: " + cnpj + "\n" + super.listarDados();
38     }
39
40 }

```

A classe PessoaJuridica é estruturada de forma bem semelhante à PessoaFisica: acrescentamos seus campos específicos (cnpj e razaoSocial) *getters* e *setters* para eles e construtores que também utilizam o comando super. Também temos uma versão específica de listarDados(), que agora trará também a razão social e o CNPJ.

Para testar o uso das duas classes, vamos modificar o método main, dentro da classe Principal, para que o usuário possa escolher, assim que o sistema for iniciado, se irá cadastrar um cliente pessoa física ou pessoa jurídica.

Altere seu método main, na classe Principal, para que ele fique como na listagem do quadro 40.

## Quadro 40 - Código da classe Principal com uso das classes PessoaFisica e PessoaJuridica

```

1 package banco.tela;
2
3 import javax.swing.JOptionPane;
4
5 import banco.modelo.Cliente;
6 import banco.modelo.Conta;
7 import banco.modelo.PessoaFisica;
8 import banco.modelo.PessoaJuridica;
9
10 public class Principal {
11
12     public static void main(String[] args) {
13
14         Cliente cliente = new Cliente();
15
16         String tipoCliente = JOptionPane.showInputDialog(null, "Escolha o tipo de cliente:\n" +
17             "F - Pessoa Física\nJ - Pessoa Jurídica");
18
19         if (tipoCliente.equals("F")) {
20             cliente = new PessoaFisica();
21             ((PessoaFisica)cliente).setNome(JOptionPane.showInputDialog(null, "Nome do Cliente: "));
22             ((PessoaFisica)cliente).setCpf(JOptionPane.showInputDialog(null, "CPF do Cliente: "));
23         } else if (tipoCliente.equals("J")) {
24             cliente = new PessoaJuridica();
25             ((PessoaJuridica)cliente).setRazaoSocial(JOptionPane.showInputDialog(null, "Razão Social: "));
26             ((PessoaJuridica)cliente).setCnpj(JOptionPane.showInputDialog(null, "CNPJ do Cliente: "));
27         } else {
28             JOptionPane.showMessageDialog(null, "OPÇÃO INVÁLIDA! Encerrando o programa...");
29             return;
30         }
31
32         cliente.setCidade(JOptionPane.showInputDialog(null, "Cidade do Cliente: "));
33         cliente.setEstado(JOptionPane.showInputDialog(null, "Estado do Cliente: "));
34
35         JOptionPane.showMessageDialog(null, "DADOS DO CLIENTE\n\n" +
36             cliente.listarDados());
37
38         Conta conta = new Conta(cliente);
39
40         JOptionPane.showMessageDialog(null, "DADOS DA CONTA\n\n" +
41             conta.listarDados());
42
43         int opcao = 0;
44         String ret;
45         do {
46             String mensagem = "SALDO EM CONTA: " + conta.getSaldoFormatado() + "\n\n" +
47                 "OPÇÕES: \n1 - Depositar valor\n2 - Sacar valor\n3 - Finalizar";
48             try {
49                 opcao = Integer.parseInt(JOptionPane.showInputDialog(null, mensagem));
50                 switch (opcao) {
51                     case 1:
52                         ret = JOptionPane.showInputDialog(null, "Valor do depósito:");
53                         conta.depositar(Double.parseDouble(ret));
54                         JOptionPane.showMessageDialog(null, "Depósito realizado!");
55                         break;
56                     case 2:
57                         ret = JOptionPane.showInputDialog(null, "Valor do saque:");
58                         if (conta.sacar(Double.parseDouble(ret))) {
59                             JOptionPane.showMessageDialog(null, "Saque realizado!");
60                         } else {
61                             JOptionPane.showMessageDialog(null, "FALHA NO SAQUE!");
62                         }
63                 }
64             } catch (NumberFormatException ex) {
65                 JOptionPane.showMessageDialog(null, "VALOR INVÁLIDO!");
66             }
67         } while ((opcao == 1) || (opcao == 2));
68     }
69 }
```

Os principais pontos nos quais ocorreram mudanças significativas são:

- **Linha 14:** começamos instanciando um objeto da classe Cliente. Criamos um objeto genérico, pois ainda não sabemos se o cliente será pessoa física ou pessoa jurídica. De qualquer maneira, será um cliente, por isso esse objeto será útil em qualquer um dos casos.
- **Linha 16:** exibimos um JOptionPane para que o usuário escolha o tipo de cliente que está cadastrando. Se informar a letra “F”, é um cliente pessoa física; caso digite “J”, será pessoa jurídica. O valor digitado fica guardado na variável tipoCliente, uma *string*.
- **Linhas 19 a 22:** se o tipo de cliente for pessoa física, iniciamos aqui um bloco que solicitará ao usuário os dados específicos. Assim, invocamos o construtor dessa classe específica para inicializar nosso objeto cliente (linha 20). Porém o compilador Java conhece a variável cliente como um objeto da classe Cliente - foi assim que a declaramos na linha 14. Apesar de o que estamos fazendo aqui ser totalmente válido – afinal, um objeto da classe PessoaFisica é também um objeto da classe Cliente (em virtude da herança) -, precisamos “avisar” o compilador Java que nas linhas 21 e 22 utilizaremos cliente como um objeto da classe PessoaFisica. Assim, usamos nessas linhas um outro recurso poderoso (e prático) de Java, o *casting*, que permite que um objeto de um tipo (classe) seja interpretado como de outro tipo (classe). Isso só será possível se houver uma relação de herança ligando esse objeto aos dois tipos, como é o caso aqui. Você não pode, por exemplo, tentar usar *casting* em objetos que não possuem relação “hereditária” um com o outro (por exemplo, um da classe Cliente com um de Conta). Nesse caso, como sabemos que o cliente também é uma pessoa física, o comando - *(PessoaFisica)cliente* - diz ao compilador que cliente também possui os métodos de um objeto PessoaFisica. Por essa razão, podemos usar setNome e setCpf nele.
- **Linha 23 a 26:** aqui fazemos a verificação de se é um cliente do tipo pessoa jurídica. Caso seja, inicializamos cliente de modo mais específico como um objeto Pessoajuridica. Em seguida, usamos novamente o *casting* para poder chamar métodos específicos de Pessoajuridica em cliente (setRazaoSocial e setCnpj).
- **Linhas 27 a 30:** caso o usuário tenha digitado um valor inválido, que não indique nem cliente pessoa física, nem pessoa jurídica, exibimos um aviso e encerramos o método main (consequentemente encerrando o programa) com return.
- **Linhas 32 e 33:** já definimos em cliente, nas linhas anteriores, os valores específicos do seu tipo (por exemplo, nome para pessoa física e razão social para pessoa jurídica). Faltam agora os dados genéricos para qualquer tipo: cidade e estado. Definimos esses valores usando agora, diretamente, o objeto cliente, sem *casting* e interpretado genericamente como do tipo Cliente.
- **Linha 35:** mostramos a “ficha cadastral” de cliente, usando o método *listarDados()*. Note que, durante a execução, será exibida a versão de *listarDados()* específica para o tipo de cliente, inicializado como pessoa física (linha 20) ou pessoa jurídica (linha 24). Aqui, estamos usando o recurso do polimorfismo, que veremos em mais detalhes na próxima aula.

Antes de executar, você perceberá que surgiu também um erro na classe Conta. Trata-se do uso do getter *getNome()* em *listarDados*. Como agora só teremos nome em cliente pessoa física (e não mais no cliente genérico, como está declarado em Conta), precisaremos fazer um pequeno ajuste nessa classe. Modifique o método *listarDados* em Conta seguindo a listagem no quadro 41.

**Quadro 41** - Nova versão do método listarDados na classe Conta

```

20 public String listarDados() {
21     String nome;
22     if (cliente instanceof PessoaFisica) {
23         nome = ((PessoaFisica)cliente).getNome();
24     } else {
25         nome = ((PessoaJuridica)cliente).getRazaoSocial();
26     }
27     return "NÚMERO: " + numero + "\n" +
28         "CORRENTISTA: " + nome + "\n" +
29         "SALDO: " + DecimalFormat.getCurrencyInstance().format(saldo);
30 }
```

Explicando o código do quadro 41, temos:

- Na linha 22, usamos um comando especial de Java chamado *instanceof*. Ele serve para verificarmos se um determinado objeto é uma instância de uma determinada classe. Por exemplo, *cliente instanceof PessoaFisica* retorna true se o objeto cliente for dessa classe ou false se ele for uma instância de PessoaJuridica (a outra possibilidade).
- Se ele for pessoa física, na linha 23, usamos novamente o *casting* para poder interpretar o cliente genérico como PessoaFisica e usar seu método *getNome*. Caso seja pessoa jurídica, usamos também *casting* na linha 25 para obter sua razão social.
- O valor obtido é usado, então, no retorno desse método para impressão dos dados da conta.

Acabamos usando aqui novamente o assunto da próxima aula: polimorfismo. Se você ainda não entendeu completamente como ele funciona, não se preocupe. Vamos posteriormente implementá-lo em mais um local importante.

## Aula 7 – Polimorfismo

### Conceito de polimorfismo

A etimologia da palavra polimorfismo já dá uma boa ideia do seu significado: originada do grego *poli morphos*, significa literalmente muitas formas. Em programação OO, o termo é aplicado às várias maneiras com que um mesmo método pode ser executado em diversas classes.

Nós já usamos o polimorfismo no exercício da aula anterior e vamos aplicá-lo novamente nesta. Veja a nova versão do diagrama de classes do sistema bancário na figura 34.

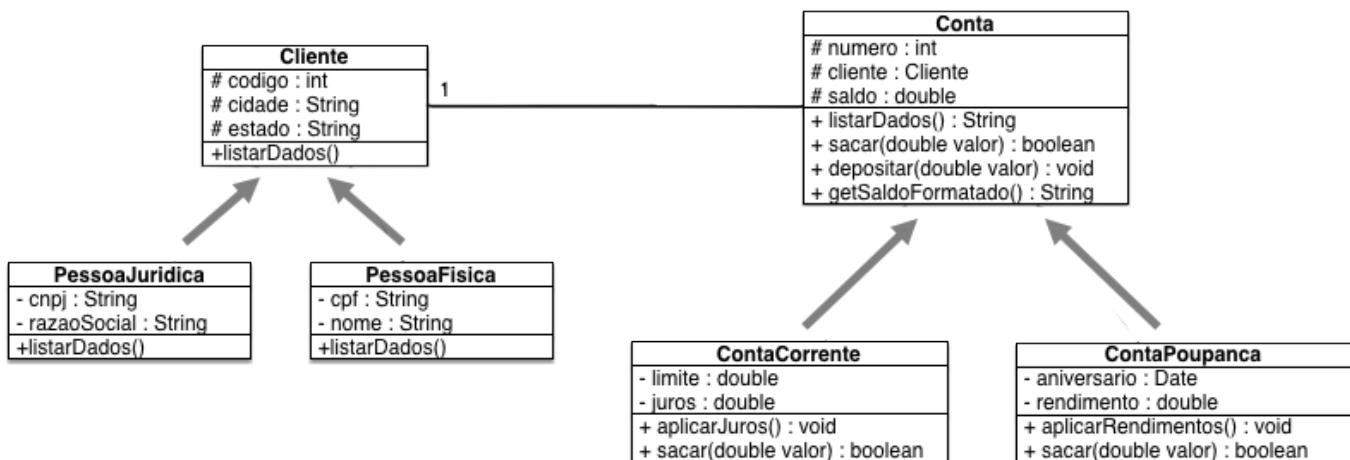


Figura 34 - Diagrama de classes do sistema bancário com polimorfismo.

Na figura, temos dois casos evidentes de polimorfismo: o método `listarDados()` e o método `sacar()`.

O primeiro caso de polimorfismo já está implementado no código Java do nosso sistema: trata-se do método `listarDados()`, existente em `Cliente`, `PessoaJuridica` e `PessoaFisica`. É um mesmo método que possui três formas diferentes de executar - semelhantes, mas com conteúdo específico à classe no qual é definido.

Se você abrir o código da classe `Cliente`, verá que ela contém a versão básica de `listarDados()`, que traz apenas uma `string` com o código, a cidade e o estado do cliente (veja no quadro 42).

#### Quadro 42 - Método `listarDados()` da classe `Cliente`

```

public String listarDados() {
    return "CÓDIGO: " + codigo + "\n" +
        "CIDADE: " + cidade + "\n" +
        "ESTADO: " + estado;
}
  
```

Como a classe `PessoaFisica` herda a classe `Cliente`, podemos concluir que um cliente `PessoaFisica` já receberá o método `listarDados` proveniente dela. Mas aí vem a pergunta: esse `listarDados` atende às necessidades de `PessoaFisica`? Não, pois não mostra duas outras informações importantes sobre esse tipo: nome e CPF. Agora, veja a nova implementação de `listarDados()` existente em `PessoaFisica`, exibida no quadro 43.

#### Quadro 43 - Implementação de `listarDados()` em `PessoaFisica`

```

public String listarDados() {
    return "NOME: " + nome + "\n" + "CPF: " + cpf + "\n" + super.listarDados();
}
  
```

Quando implementamos `listarDados()` em `PessoaFisica`, nós sobrescrevemos a versão original existente em `Cliente`. Em Java, esse ato é conhecido como *override*. Isso significa que essa nova versão do quadro 43 é que será executada quando um objeto `PessoaFisica` executar o método `listarDados`.

Mas isso não significa que perdemos o `listarDados` original. Veja no quadro 43 que, dentro de `listarDados()`, fazemos uma chamada ao existente em `Cliente`, nossa superclasse, usando o comando `super`. Assim, podemos fazer com que essa nova versão funcione como um complemento da versão anterior.

## Sobrecarga de métodos

A *sobrecarga de métodos* é um poderoso recurso existente na linguagem Java. Com ela, podemos definir versões diferentes de um mesmo método (ou seja, com um mesmo nome) em uma mesma classe. Podemos diferenciá-los por seus parâmetros.

Vamos supor que precisássemos ter em nossa classe `Conta` um método para depositar em dinheiro e outro para depositar em cheque. A ideia inicial de quem não conhece a possibilidade da sobrecarga de métodos é implementar dois métodos com nomes distintos:

```
depositarEmDinheiro(double valor);
depositarEmCheque(double valor, int numeroDoCheque);
```

Com essa abordagem, estamos dificultando um pouco mais a manutenção e a legibilidade do sistema. Usando a sobrecarga de métodos, poderíamos criar os dois métodos da seguinte forma:

```
depositar(double valor);
depositar(double valor, int numeroDoCheque);
```

Quando o programador quiser invocar um depósito em dinheiro, basta passar ao método `depositar` um único parâmetro, o valor do depósito:

```
depositar(5000);
```

Quando for feito em cheque, passamos dois parâmetros: o valor e o número do cheque.

```
depositar(2000, 73612);
```

O compilador Java saberá que, no primeiro caso, estamos chamando a versão de depósito para dinheiro, já que só um valor está sendo passado, e que, no segundo, queremos a versão de depósito para cheques, uma vez que dois parâmetros estão sendo informados, um jeito muito mais fácil e sucinto.

Nós também já fizemos sobrecarga de métodos. Quando? Nos construtores das classes. Veja, no quadro 44, os dois construtores para `Cliente` que criamos.

**Quadro 44:** Construtores sobrearc  
regados para a classe `Cliente`

```
public Cliente() {
    quantidade++;
    codigo = quantidade;
    cidade = "São Paulo";
    estado = "São Paulo";
}

public Cliente(String cidade, String estado) {
    quantidade++;
    codigo = quantidade;
    this.cidade = cidade;
    this.estado = estado;
}
```

Quando instanciamos um objeto cliente, a quantidade de parâmetros passada ao construtor determina qual versão será usada:

```
Cliente cli1 = new Cliente(); // cria um cliente usando o primeiro construtor
Cliente cli2 = new Cliente("Campinas", "São Paulo"); // cria usando a segunda versão
```

## Polimorfismo na prática em Java: exercício prático guiado

O exercício desta aula consiste em criar as duas subclasses de Conta mostradas no diagrama da figura 34: *ContaCorrente* e *ContaPoupanca*. O mais importante a ser implementado nelas é o conceito de polimorfismo nas diferentes versões do método *sacar*.

No caso de *ContaCorrente*, o saque deverá considerar o limite na verificação do saldo disponível (ou seja, o valor sacado deverá ser inferior ou igual ao saldo acrescido do limite). Para *ContaPoupanca*, deverá incluir uma verificação: só será permitido se o dia atual for superior ao dia do aniversário da conta, para que não haja prejuízo de rentabilidade (não é necessário verificar se é dia útil, como seria o caso em um sistema real).

Não é necessário implementar os métodos *aplicarJuros()* (de *ContaCorrente*) e *aplicarRendimentos()* (de *ContaPoupanca*), mas, se você quiser tentar, vá em frente.

Não se esqueça de adaptar a classe Principal para utilizar as novas subclasses de Conta. Dê ao usuário a possibilidade de escolher a criação de uma conta corrente ou de uma conta poupança.

**Dica:** para facilitar, crie o campo *aniversario* de *ContaPoupanca* como um *int* (ao invés de *Date*, como indicado na figura 34) e armazene nele somente o dia do aniversário da conta poupança. Para obter o dia atual em Java, use a classe *Calendar* com a seguinte linha de código:

```
int hoje = Calendar.getInstance().get(Calendar.DAY_OF_MONTH);
```

Tente fazer essa prática sozinho e depois volte para conferir o resultado, mostrado a seguir no passo a passo da solução.

### 1º passo: implementando a classe *ContaCorrente*

Vamos começar implementando a classe *ContaCorrente*. Clique sobre o pacote *banco.modelo* no Project Explorer e, em seguida, no botão *New Java Class* da barra de ferramentas do Eclipse. Dê a essa classe o nome *ContaCorrente* e escreva seu código seguindo a listagem do quadro 45. Não se esqueça de indicar que ele herda a classe *Conta*, por meio do uso da palavra *extends* na sua declaração (linha 3).

**Quadro 45** - Código da classe ContaCorrente

```

1 package banco.modelo;
2
3 import java.text.DecimalFormat;
4
5 public class ContaCorrente extends Conta {
6
7     private double limite;
8     private double juros;
9
10    public ContaCorrente(Cliente cliente) {
11        super(cliente);
12        limite = 1000;
13        juros = 5.9;
14    }
15
16    public ContaCorrente(Cliente cliente, double limite, double juros) {
17        super(cliente);
18        this.limite = limite;
19        this.juros = juros;
20    }
21
22    @Override
23    public String getSaldoFormatado() {
24        return DecimalFormat.getCurrencyInstance().format(saldo + limite);
25    }
26
27    @Override
28    public boolean sacar(double valor) {
29        if (saldo + limite >= valor) {
30            saldo -= valor;
31            return true;
32        } else {
33            return false;
34        }
35    }
36
37    @Override
38    public String listarDados() {
39        String nome;
40        if (cliente instanceof PessoaFisica) {
41            nome = ((PessoaFisica)cliente).getNome();
42        } else {
43            nome = ((PessoaJuridica)cliente).getRazaoSocial();
44        }
45        return "NÚMERO: " + numero + "\n" +
46               "CORRENTISTA: " + nome + "\n" +
47               "SALDO: " + DecimalFormat.getCurrencyInstance().format(saldo + limite);
48    }
49
50    public void aplicarJuros() {
51        if (saldo < 0)
52            saldo = saldo - (saldo * juros / 100);
53    }
54
55 }
```

A explicação do código do quadro 45 é:

- Linhas 7 e 8: declaramos aqui os dois campos privados específicos dessa classe, o limite e a taxa de juros, ambos do tipo double.
- Linhas 10 a 14: o primeiro construtor para ContaCorrente recebe um objeto Cliente apenas como parâmetro, invoca o construtor equivalente na superclasse e atribui valores *default* para o limite e a taxa de juros.
- Linhas 16 a 20: o segundo construtor permite a especificação de valores para o limite e a taxa de juros da nova conta corrente.
- Linhas 22 a 25: nessas linhas, está declarada uma reimplementação do método *getSaldoFormatado()*, que já existia na superclasse Conta. Na linha 22, inserimos uma *annotation* (anotação) Java, que funciona como uma espécie de “comentário inteligente” - nesse caso, a *annotation* `@Override` sinaliza ao compilador que esse método está sobrescrevendo outro existente na superclasse. Apesar de não ser obrigatório, o uso dela é conhecido como uma boa prática de programação Java. A nova versão de *getSaldoFormatado()* foi escrita para trazer o valor do limite somado ao saldo, obtendo assim o real saldo disponível (linha 24).
- Linhas 27 a 35: aqui está contida a reimplementação do método *sacar()*. O método *sacar()* de ContaCorrente deve considerar também o limite disponível no cheque especial, o que fazemos na linha 29.
- Linhas 37 a 48: a reimplementação de *listarDados()* contém o acréscimo do limite na exibição do saldo disponível.
- Linhas 50 a 53: caso você tenha implementado o método *aplicarJuros()*, essa seria uma das maneiras de escrevê-lo - subtraindo o valor dos juros do saldo apenas se este estiver negativo (sem considerar o limite do cheque especial, nesse caso).

## 2º passo: implementando a classe ContaPoupanca

Vamos agora implementar a classe *ContaPoupanca*. Repita o procedimento de criação de uma classe, dando a ela agora o nome *ContaPoupanca* (sem usar cedilha) e implementando seu código de acordo com a listagem mostrada no quadro 46.

### Quadro 46 - Código da classe ContaPoupanca

```

1 package banco.modelo;
2
3 import java.util.Calendar;
4
5 public class ContaPoupanca extends Conta {
6
7     private int aniversario;
8     private double rendimento;
9
10    public ContaPoupanca(Cliente cliente) {
11        super(cliente);
12        aniversario = Calendar.getInstance().get(Calendar.DAY_OF_MONTH);
13        rendimento = 0.52;
14    }
15
16    public ContaPoupanca(Cliente cliente, int aniversario, double rendimento) {
17        super(cliente);
18        this.aniversario = aniversario;
19        this.rendimento = rendimento;
20    }
21
22    @Override
23    public boolean sacar(double valor) {
24        int hoje = Calendar.getInstance().get(Calendar.DAY_OF_MONTH);
25        if ((hoje > aniversario) && (saldo >= valor)) {
26            saldo -= valor;
27            return true;
28        } else {
29            return false;
30        }
31    }
32
33    public void aplicarRendimentos() {
34        if (aniversario == Calendar.getInstance().get(Calendar.DAY_OF_MONTH)) {
35            saldo = saldo + (saldo * rendimento / 100);
36        }
37    }
38}
39

```

A explicação do código do quadro 46 é:

- Linhas 7 e 8: declaramos os dois campos específicos de uma conta poupança - seu dia de aniversário e sua taxa de rendimento.
- Linhas 10 a 20: nelas, estão contidos os dois construtores para ContaPoupanca, um com valores *default* para os campos e o outro permitindo que eles possam ser passados como parâmetro.
- Linhas 22 a 31: a implementação do método *sacar()*, em ContaPoupanca, deverá verificar se o saque estará sendo feito após o dia do aniversário, garantindo que os rendimentos tenham sido aplicados. Na realidade, a implementação prática deveria considerar também o mês para ficar mais condizente com a realidade, mas simplificamos nesse caso para podermos nos concentrar no uso do polimorfismo. Você pode ajustar o código, caso julgue adequado, mas veja a nota importante abaixo.

- Linhas 33 a 37: essa é uma das formas como você pode implementar o método `aplicarRendimentos()`, creditando a aplicação da taxa de rendimentos sobre o saldo da conta.
- **Importante:** além da questão da consideração do mês na data de aniversário, em uma situação real, precisaríamos também possuir um *setter* para o campo rendimento (`setRendimento`), uma vez que essa taxa varia de mês a mês.

### 3º passo: usando as novas classes implementadas

Podemos agora implementar o uso das novas classes no sistema.

Vamos acrescentar à interface, no método main da classe Principal, a possibilidade de o usuário escolher qual tipo de conta está criando. Adicione a ela o trecho de código indicado na listagem exibida no quadro 47. Ele dever ser inserido no local em que estava sendo instanciada a classe Conta, logo após a exibição dos dados do Cliente.

**Quadro 47** - Inclusão da escolha do tipo de conta na interface do sistema

```

cliente.setCidade(JOptionPane.showInputDialog(null, "Cidade do Cliente: "));
cliente.setEstado(JOptionPane.showInputDialog(null, "Estado do Cliente: "));

JOptionPane.showMessageDialog(null, "DADOS DO CLIENTE\n\n" +
    cliente.listarDados());

// INÍCIO DO NOVO TRECHO

Conta conta;

String tipoConta = JOptionPane.showInputDialog(null, "Tipo de conta a ser criada:\n" +
    "C - Conta Corrente\nP - Conta Poupança");

if (tipoConta.equals("P")) {
    conta = new ContaPoupanca(cliente);
} else {
    conta = new ContaCorrente(cliente);
}

// FIM DO NOVO TRECHO

JOptionPane.showMessageDialog(null, "DADOS DA CONTA\n\n" +
    conta.listarDados());

```

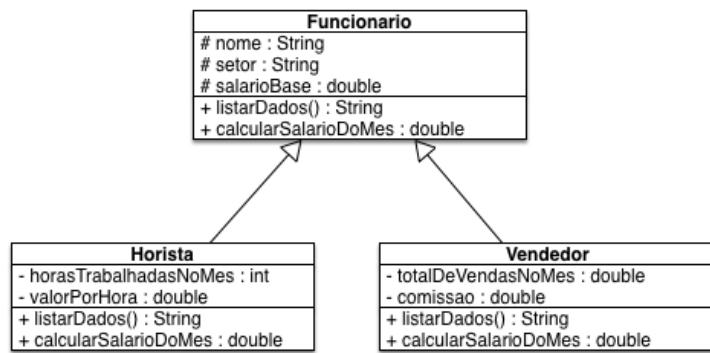
O mais interessante é que não precisamos modificar mais nada. O uso do polimorfismo faz o resto.

Se for instanciada uma conta corrente, a exibição do saldo e o saque considerarão o limite disponível, o que não ocorrerá no caso de uma conta poupança. Faça o teste e veja na prática.

Você verá que, no caso de ContaPoupanca, o saque não será permitido, pois usamos o construtor que define o dia atual como aniversário. A condição implementada em sacar() impede que ele seja realizado, pois verificará que o dia atual não é superior ao aniversário. Para que o saque em uma ContaPoupanca funcione, modifique a linha relativa à criação do objeto dessa classe para usar o construtor com mais parâmetros e passe um dia de aniversário inferior ao atual.

## Exercício do Módulo 3

Para praticar o conceito de herança, implemente no Eclipse um projeto que contenha as seguintes classes:



Neste exercício, você deverá fazer uso da herança na criação das classes, assim como do polimorfismo nos métodos `listarDados` e `calcularSalarioDoMes`.

- Para **Funcionario**, o salário do mês será apenas seu salário base.
- Para **Horista**, o salário do mês será o total de horas trabalhadas multiplicadas pelo valor por hora.
- Já no caso do **Vendedor**, o salário será a soma do salário base com a comissão aplicada do total de vendas realizado no mês.

## Módulo 4 – Arrays de objetos e coleções

---

Existe um momento em que um conjunto limitado de variáveis não é suficiente nem adequado para armazenar um grupo de informações. Nos casos em que precisamos ter em um sistema uma série de dados de um determinado tipo, precisamos recorrer a estruturas que possibilitem o armazenamento de muitos valores ao mesmo tempo. Você já conhece os vetores em Java usados para armazenar um conjunto de valores de tipos primitivos, como *int* ou *double*. Mas e no caso de objetos? O que fazemos quando precisamos guardar, por exemplo, um conjunto de objetos da classe Conta? Veremos neste último módulo como resolver essa questão.

### Aula 8 - Arrays e coleções em Java

#### Arrays de objetos e coleções

Um vetor em Java de valores de um tipo primitivo, como *int*, pode ser declarado, inicializado e utilizado da forma indicada no quadro 48.

**Quadro 48** - Uso de um vetor de inteiros

```
int[] numerosDaSorte = new int[6];

numerosDaSorte[0] = 6;
numerosDaSorte[1] = 19;
numerosDaSorte[2] = 21;
numerosDaSorte[3] = 23;
numerosDaSorte[4] = 53;
numerosDaSorte[5] = 59;

JOptionPane.showMessageDialog(null, Arrays.toString(numerosDaSorte));
```

No exemplo, declaramos um vetor de seis valores *int*, chamado numerosDaSorte, atribuímos a cada posição um valor e exibimos seu conteúdo com um JOptionPane. Sabemos que, em Java, os índices de um vetor são baseados em zero, o que indica que o primeiro elemento terá índice 0 e o último, n - 1 (sendo n a quantidade de elementos do vetor). Também sabemos que vetores devem ser inicializados com a especificação da quantidade de elementos que ele conterá para que a memória necessária seja alocada. Assim, não podemos aumentar ou diminuir o tamanho de um vetor após ser inicializado.

O que você ainda não sabe é que podemos ter em Java vetores de objetos, assim como de tipos primitivos. Por exemplo, poderíamos declarar no nosso sistema bancário um vetor para Clientes e armazenar objetos dessa classe em suas posições, como é mostrado no quadro 49.

**Quadro 49** - Declarando, inicializado e utilizando um vetor de objetos Cliente

```

Cliente[] clientes = new Cliente[5];

clientes[0] = new Cliente();
clientes[1] = new Cliente();
clientes[2] = new Cliente();
clientes[3] = new Cliente();
clientes[4] = new Cliente();

clientes[0].setCidade("Marília");
clientes[2].setCidade("Campinas");

```

O vetor clientes possui espaço para cinco objetos da classe Cliente. Perceba que, para cada objeto referenciado por seu respectivo índice, podemos usar seus respectivos métodos individualmente. Essa é uma abordagem interessante quando conhecemos previamente a quantidade de objetos que serão utilizados. Mas e quando não sabemos?

Uma solução seria declarar um “megavetor”, com espaços além da capacidade máxima possível. Mas assim ainda teríamos problemas, como o uso desnecessário de espaço de memória para os elementos que não forem utilizados e ainda o risco de chegarmos ao limite declarado - mesmo que isso seja improvável, seria um grande problema se acontecesse.

O ideal, nesses casos, é usar as classes de coleções da linguagem Java, que possibilitam que criemos objetos *containers* de outros objetos - ou seja, vetores também, mas com tamanho dinâmico. Objetos de coleção podem conter N objetos, sem precisar declarar a sua quantidade inicial de elementos e sem desperdício de espaço.

Existem várias classes de coleções de objetos em Java, mas vamos aprender apenas uma aqui, a mais usada de todas: *ArrayList*. Na próxima seção, veremos como usá-la para implementar uma outra parte do sistema bancário.

### **Uso de ArrayList e métodos para manipulação**

Vamos praticar o uso das coleções de objetos em Java com *ArrayList* ao mesmo tempo em que aprendemos como implementar mais uma parte do sistema bancário. Veja a nova versão do nosso diagrama de classes na figura 35.

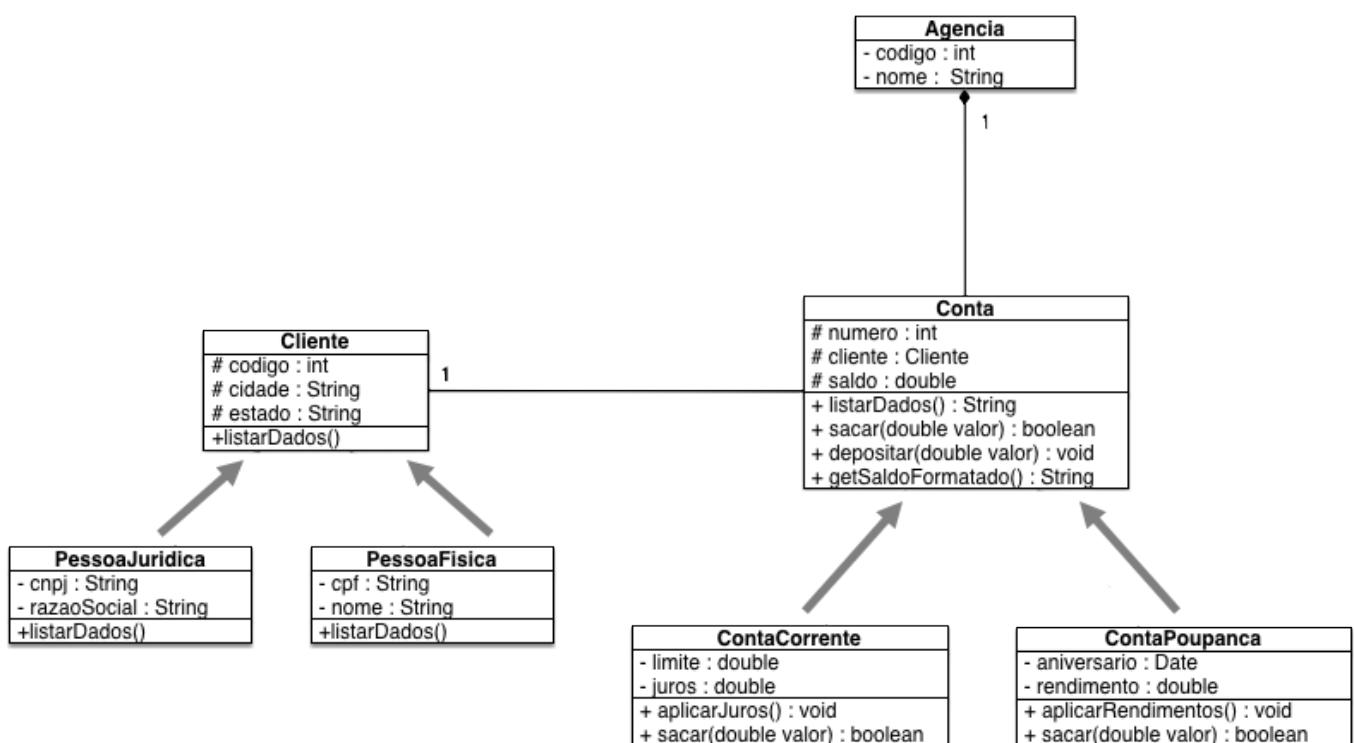


Figura 35 - Diagrama de classes do sistema bancário.

A primeira diferença que você deve ter notado é a inclusão da classe *Agencia*, para representar uma agência dentro da rede bancária. Agora veja a linha que liga *Agencia* a *Conta* e tente se lembrar do seu significado.

O losango preenchido na ponta de *Agencia* indica que ela possui uma *relação de composição* com *Conta*, ou seja, uma agência é composta por várias contas (o asterisco na ponta conectada à *Conta* indica essa multiplicidade). Relembrando, a composição indica que um todo (nesse caso, a *Agencia*) é composta por várias partes (as *Contas*). A agência não pode existir sem contas, por isso temos nesse caso uma composição e não uma agregação.

Como, em Java, vamos saber quais as contas que compõem uma agência? Uma maneira simples e eficiente é usar um *ArrayList* de objetos da classe *Conta* em *Agencia*.

### Importante!

Seguindo as regras exatas da composição, não poderíamos permitir que uma agência existisse sem contas. Porém não vamos implementar aqui esse tipo de controle e validação, visando facilitar o entendimento do código do sistema.



Declaramos um *ArrayList* da seguinte forma:

```
List<Classe> lista = new ArrayList<>();
```

Classe é o nome da classe de objetos que serão colocados na lista. Você deve importar a classe `java.util.ArrayList` e a interface `java.util.List` no seu código. `List` é uma interface para a implementação de listas em Java; `ArrayList` é uma classe que implementa essa interface. Quando você for apresentado às interfaces em Java, verá que elas são maneiras de se estabelecer o que uma classe deverá ter em sua implementação.

Para armazenar um objeto no `ArrayList`, usamos seu método `add`:

```
Classe objeto = new Classe();
lista.add(objeto);
```

Para remover um objeto em um `ArrayList`, é utilizado o método `remove`:

```
lista.remove(objeto);
```

Conseguimos saber quantos objetos existem no `ArrayList` usando seu método `size()`:

```
int totalDeObjetos = lista.size();
```

Podemos usar uma versão do comando `for` que percorre automaticamente os objetos existentes em um `ArrayList`, um de cada vez:

```
for (Classe objeto : lista) {
    objeto.método();
}
```

Esse laço se repetirá pela quantidade de objetos existentes em lista, percorrendo do primeiro ao último. Em cada passagem, o objeto representará o elemento atual.

## Uso de coleções no sistema bancário: exercício prático guiado

Para entender melhor como funciona um `ArrayList`, vamos implementar a classe `Agencia` e modificar a classe `Principal` para possibilitar a inclusão de múltiplas contas.

Crie um nova classe Java dentro do pacote `banco.modelo`, chamada `Agencia`, e escreva seu código de acordo com a listagem do quadro 50.

**Quadro 50** – Código-fonte da classe `Agencia`

```
1 package banco.modelo;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Agencia {
7
8     private int numero;
9     private String nome;
10    private List<Conta> contas;
11
12    public Agencia(int numero, String nome) {
13        this.numero = numero;
14        this.nome = nome;
15        contas = new ArrayList<>();
16    }
17
18    public int getNumero() {
19        return numero;
20    }
21
22    public String getNome() {
23        return nome;
24    }
25
26    public List<Conta> getContas() {
27        return contas;
28    }
29
30    public void incluirConta(Conta conta) {
31        contas.add(conta);
32    }
33
34    public void excluirConta(Conta conta) {
35        contas.remove(conta);
36    }
37}
```

O código do quadro 50 pode ser assim explicado:

- **Linhas 8 a 10:** um objeto da classe `Agencia` possuirá número, nome e uma lista de objetos da classe `Conta`, denominada `contas`. Seguindo o princípio do encapsulamento, todos esses campos são privados.

- **Linhas 12 a 16:** a classe Agencia possui um único construtor, que recebe o número e o nome da agência como parâmetros e inicializa a sua lista de contas vazia.
- **Linhas 18 a 28:** temos aqui um conjunto de getters para os campos. Repare que não definimos setters; assim, os valores deles não poderão ser modificados após serem criados com o construtor. Apenas no caso da lista de contas poderemos acrescentá-las ou removê-las, por meio dos métodos que serão definidos.
- **Linhas 30 a 36:** definimos aqui um método para incluir um objeto à lista de contas e outro para remover. Como são públicos, poderão ser utilizados para adicionarmos e excluirmos as contas de uma agência.

Na tela Principal, vamos criar um objeto agencia e acrescentar um laço externo no qual o usuário poderá optar entre incluir nova conta e cliente, listar as contas existentes ou sair do sistema. Modifique a sua classe Principal, começando pelo trecho mostrado no quadro 51.

**Quadro 51 – Classe Principal: linhas 1 a 28**

```

1 package banco.tela;
2
3 import javax.swing.JOptionPane;
4
5 import banco.modelo.Agencia;
6 import banco.modelo.Cliente;
7 import banco.modelo.Conta;
8 import banco.modelo.ContaCorrente;
9 import banco.modelo.ContaPoupanca;
10 import banco.modelo.PessoaFisica;
11 import banco.modelo.PessoaJuridica;
12
13 public class Principal {
14
15     public static void main(String[] args) {
16
17         Agencia agencia = new Agencia(1, "São Paulo - Centro");
18         String opcaoPrincipal = "";
19
20         do {
21             opcaoPrincipal = JOptionPane.showInputDialog(null, "Cadastro de Contas para a agência "
22                     + agencia.getNumero() + " - " + agencia.getNome() + "\n"
23                     + "\nOPÇÕES:\n"
24                     + "1 - Incluir cliente e conta\n"
25                     + "2 - Listar contas cadastradas\n"
26                     + "3 - Sair do sistema");
27
28             if (opcaoPrincipal.equals("1")) {

```

Conforme mostra o quadro 51:

- Nesse início, declaramos e instanciamos agencia na linha 17. Também declaramos uma nova variável para a escolha do usuário no menu principal (a string opcaoPrincipal, na linha 18).
- Começamos um laço na linha 20 que vai englobar todo o código até a linha 109 (mostrada adiante, no quadro 54) e que se repetirá enquanto o usuário escolher as opções 1 ou 2 do menu criado entre as linhas 21 e 26.
- Na linha 28, caso o usuário escolha a opção 1 do menu (criar cliente e conta), caímos em um bloco que vai até a linha 95 (exibido adiante no quadro 54).

O trecho exibido no quadro 52 é o mesmo que tínhamos anteriormente. A diferença fundamental é que agora ele está contido dentro do laço principal iniciado na linha 20.

**Quadro 52** – Classe Principal: linhas 29 a 56

```
29  
30  
31 Cliente cliente = new Cliente();  
32 String tipoCliente = JOptionPane.showInputDialog(null, "Escolha o tipo de cliente:\n" +  
     "F - Pessoa Física\nJ - Pessoa Jurídica");  
33  
34 if (tipoCliente.equals("F")) {  
35     cliente = new PessoaFisica();  
36     ((PessoaFisica)cliente).setNome(JOptionPane.showInputDialog(null, "Nome do Cliente: "));  
37     ((PessoaFisica)cliente).setCpf(JOptionPane.showInputDialog(null, "CPF do Cliente: "));  
38 } else if (tipoCliente.equals("J")) {  
39     cliente = new PessoaJuridica();  
40     ((PessoaJuridica)cliente).setRazaoSocial(JOptionPane.showInputDialog(null, "Razão Social: "));  
41     ((PessoaJuridica)cliente).setCnpj(JOptionPane.showInputDialog(null, "CNPJ do Cliente: "));  
42 } else {  
43     JOptionPane.showMessageDialog(null, "OPÇÃO INVÁLIDA! Encerrando o programa...");  
44     return;  
45 }  
46  
47 cliente.setCidade(JOptionPane.showInputDialog(null, "Cidade do Cliente: "));  
48 cliente.setEstado(JOptionPane.showInputDialog(null, "Estado do Cliente: "));  
49  
50 JOptionPane.showMessageDialog(null, "DADOS DO CLIENTE\n\n" +  
51     cliente.listarDados());  
52  
53 Conta conta;  
54 String tipoConta = JOptionPane.showInputDialog(null, "Tipo de conta a ser criada:\n" +  
     "C - Conta Corrente\nP - Conta Poupança");
```

O programa prossegue como mostra a listagem dos quadros 53 e 54.

**Quadro 53** – Classe Principal: linhas 57 a 83

```
57 if (tipoConta.equals("P")) {
58     conta = new ContaPoupanca(cliente, 10, 5);
59 } else {
60     conta = new ContaCorrente(cliente);
61 }
62
63 agencia.incluirConta(conta);
64
65 JOptionPane.showMessageDialog(null, "DADOS DA CONTA\n\n" +
66         conta.listarDados());
67
68 int opcao = 0;
69 String ret;
70
71 do {
72     String mensagem = "SALDO EM CONTA: " + conta.getSaldoFormatado() + "\n\n" +
73         "OPÇÕES: \n1 - Depositar valor\n2 - Sacar valor\n3 - Finalizar";
74     try {
75         opcao = Integer.parseInt(JOptionPane.showInputDialog(null, mensagem));
76         switch (opcao) {
77             case 1:
78                 ret = JOptionPane.showInputDialog(null, "Valor do depósito:");
79                 conta.depositar(Double.parseDouble(ret));
80                 JOptionPane.showMessageDialog(null, "Depósito realizado!");
81                 break;
82             case 2:
83                 ret = JOptionPane.showInputDialog(null, "Valor do saque:");
84                 conta.sacar(Double.parseDouble(ret));
85                 JOptionPane.showMessageDialog(null, "Saque realizado!");
86                 break;
87         }
88     } catch (Exception e) {
89         JOptionPane.showMessageDialog(null, "Opção inválida!");
90     }
91 }
92
93 JOptionPane.showMessageDialog(null, "Operação finalizada!");
94
95 conta.exibirRelatorio();
96
97 conta.sair();
98
99 conta.close();
```

**Quadro 54** – Classe Principal: linhas 84 a 112

```

84     if (conta.sacar(Double.parseDouble(ret))) {
85         JOptionPane.showMessageDialog(null, "Saque realizado!");
86     } else {
87         JOptionPane.showMessageDialog(null, "FALHA NO SAQUE!");
88     }
89 }
90 } catch (NumberFormatException ex) {
91     JOptionPane.showMessageDialog(null, "VALOR INVÁLIDO!");
92 }
93 } while ((opcao == 1) || (opcao == 2));
94
95 } else if (opcaoPrincipal.equals("2")) {
96
97     if (agencia.getContas().size() == 0) {
98         JOptionPane.showMessageDialog(null, "NÃO HÁ CONTAS CADASTRADAS NO MOMENTO.");
99     } else {
100        JOptionPane.showMessageDialog(null, "A Agência " + agencia.getNumero() + " - "
101                                + agencia.getNome() + " possui " + agencia.getContas().size()
102                                + " conta(s).\n\nVeja quais são nas próximas telas");
103
104        for (Conta umaConta : agencia.getContas()) {
105            JOptionPane.showMessageDialog(null, umaConta.listarDados());
106        }
107    }
108 }
109 } while ((opcaoPrincipal.equals("1")) || (opcaoPrincipal.equals("2")));
110
111 }
112 }
```

A listagem mostrada no quadro 53 possui um único, e importantíssimo, acréscimo: a linha 63. Nela, adicionamos o objeto ContaCorrente ou ContaPoupança criado à lista de contas da agência. Note que, sendo uma lista de objetos da classe Conta, podemos acrescentar tanto uma conta corrente como uma conta poupança, já que ambas são subclasses de Conta.

Se a opção do usuário no menu principal, mostrado no início do laço mais externo, for a segunda, ele deseja visualizar uma listagem das contas cadastradas. São então executados os comandos contidos entre as linhas 95 e 108.

Na linha 97, usamos o método `size()` do `ArrayList` `contas` para saber se existem contas incluídas nele. Caso ainda não existam, avisamos o usuário do fato na linha 98.

Se contas possuir objetos, mostramos ao usuário os dados da agência e quantas contas ela possui (linhas 100 a 102). Em seguida, usando a versão do laço `for` que conhecemos na seção anterior desta aula, percorremos todas as contas existentes na lista, mostrando seus dados ao usuário (linhas 104 a 106).



## O que praticamos com este exemplo

Com essa última adição, implementamos boa parte do projeto de sistema bancário, incluindo os seguintes conceitos:

- *Classes e objetos básicos*, com seus atributos e ações.
- *Encapsulamento*, ocultando dados das classes que não devem ser acessados ou modificados e controlando o uso deles, quando necessário, com *getters* e *setters*.
- *Generalização e herança*, na implementação de Conta, ContaCorrente e ContaPoupanca, assim como Cliente, PessoaFisica e PessoaJuridica.
- *Polimorfismo*, na reescrita de métodos herdados, como listarDados() em Conta e ContaCorrente
- *Sobrecarga de métodos*, nos construtores de PessoaFisica, por exemplo
- *Composição*, usando ArrayList para fazer com que um objeto Agencia possua uma coleção de objetos Conta.

## Exercício do Módulo 4

Crie um aplicativo Java para armazenar uma lista de contatos.

Para isso, escreva o código necessário para uma classe denominada Contato, contendo campos para nome, telefone e e-mail. Não se esqueça de criar um construtor, assim como getters e setters para os campos.

Na classe Principal, possibilite ao usuário cadastrar um conjunto de contatos e, antes de finalizar, mostre os dados de todos a ele.

Para exibição dos dados de um contato, crie um método equivalente ao listarDados que usamos para Clientes, mas chame-o agora de *toString()*.

Fazendo a criação do *toString()* em Contatos, poderemos exibir os dados de um contato usando simplesmente o objeto como parâmetro da impressão. Ao invés de usar:

```
JOptionPane.showMessageDialog(null, objetoContato.listarDados() );
```

Usamos somente:

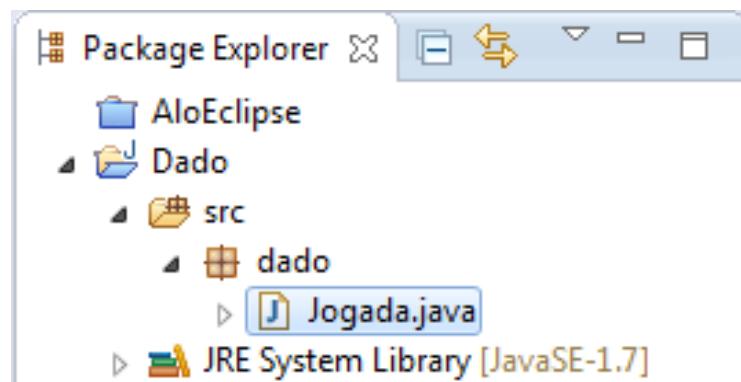
```
JOptionPane.showMessageDialog(null, objetoContato);
```

# Respostas Comentadas dos Exercícios

## Módulo 1

### Dado Virtual

O primeiro cuidado que você deve ter é com relação à padronização do nome dos pacotes em Java (sempre com todas as letras minúsculas) e das classes (primeira letra em maiúscula). A estrutura do seu projeto Dado deve ter ficado da seguinte maneira:



Para realizar o sorteio de um número aleatório de 1 a 6, você pode utilizar a classe **Random**, que você conheceu na disciplina de Lógica de Programação. Após instanciá-la em um objeto, basta chamar o método **nextInt()** para sortear o número, que deverá ser exibido na janela do **JOptionPane**.

Esta é uma das possíveis soluções para este exercício:

```

1 package dado;
2
3 import java.util.Random;
4
5 import javax.swing.JOptionPane;
6
7 public class Jogada {
8
9     public static void main(String[] args) {
10
11         Random gerador = new Random();
12         JOptionPane.showMessageDialog(null, gerador.nextInt(6) + 1);
13     }
14
15 }
```

Comentários sobre o código:

- **Linha 3:** como vamos utilizar a classe **Random**, precisamos colocar um **import** para ela, de forma que seja reconhecida por nossa classe **Jogada**.
- **Linha 11:** nesta linha instanciamos um objeto da classe **Random**, denominado **gerador**, e que será usado para sorteio do número aleatório.
- **Linha 12:** fazemos aqui um “dois-em-um” sorteamos e mostramos o número usando apenas uma linha de código. O método **nextInt(valor)**, existente nos objetos da classe **Random**, sorteia um número aleatório de 0 (inclusive) a valor - 1 (ou seja, o número anterior ao passado como parâmetro ao comando `nextInt`, incluindo este como possibilidade).
- Isso significa que **nextInt(6)** poderia trazer como resultados possíveis os valores 0, 1, 2, 3, 4 e 5. Como não podemos ter 0 como resultado, e precisamos que o número seis também possa ser dado como sorteado, simplesmente somamos 1 ao valor obtido. Assim, poderemos ter efetivamente qualquer um dos seguintes resultados: 1, 2, 3, 4, 5 ou 6.
- Note que o resultado de **nextInt** é somado em 1, para que depois seja mostrado por **showMessageDialog**. Isso significa que o método **showMessageDialog** pode ser usado para exibir diretamente valores inteiros, sem necessidade de conversão para **String**.

## Potência

A implementação desse exercício é uma versão simplificada do que fizemos na Calculadora. Veja a listagem.

```

1 package potencia;
2
3 import javax.swing.JOptionPane;
4
5 public class Potencia {
6
7     public static void main(String[] args) {
8
9         double base, expoente, resultado;
10        String valorStr;
11
12        try {
13            valorStr = JOptionPane.showInputDialog(null, "Informe a base:");
14            base = Double.parseDouble(valorStr);
15            valorStr = JOptionPane.showInputDialog(null, "Informe o expoente:");
16            expoente = Double.parseDouble(valorStr);
17        } catch (NumberFormatException ex) {
18            JOptionPane.showMessageDialog(null, "Valor inválido. Fechando o programa...");
19            return;
20        }
21
22        resultado = Math.pow(base, expoente);
23        JOptionPane.showMessageDialog(null, base + " elevado a " + expoente +
24            " é igual a " + resultado);
25    }
26 }
```

Para poder realizar o que foi pedido no enunciado (fechar o programa caso seja fornecido valor inválido para base ou expoente), agrupamos as duas entradas de dados e tentativa de conversão no mesmo bloco **try** (**linhas 13 a 16**).

Caso ocorra um erro de conversão, a **linha 18** no bloco **catch** mostra uma mensagem de erro, seguida, na **linha 19**, por uma simples instrução **return**. Ela ocasiona o encerramento do método **main**, fazendo com que o programa seja finalizado.

O método que faz o cálculo de potência é **Math.pow()**. Ele recebe dois parâmetros do tipo **double**: primeiro a base e depois o expoente. Assim, fazemos o cálculo na linha 22 e mostramos o resultado usando **showMessageDialog()**, na **linha 23**.

**OBSERVAÇÃO:** As linhas 23 e 24 correspondem a um único comando. Quebramos a linha no meio dele para facilitar a visualização na listagem

## Conversor Celsius para Fahrenheit

A implementação do conversor envolve basicamente a implementação correta da fórmula de conversão fornecida no enunciado. Uma possível solução é apresentada na listagem a seguir.

```

1 package temperatura;
2
3 import javax.swing.JOptionPane;
4
5 public class Temperatura {
6
7     public static void main(String[] args) {
8
9         double celsius = 0;
10        double fahr;
11
12        try {
13            String resp = JOptionPane.showInputDialog(null, "Informe os graus em celsius:");
14            celsius = Double.parseDouble(resp);
15        } catch (NumberFormatException ex) {
16            JOptionPane.showMessageDialog(null, "Valor inválido informado. Finalizando.");
17            return;
18        }
19
20        fahr = ((celsius * 9) / 5) + 32;
21        JOptionPane.showMessageDialog(null, celsius + " Celsius = " + fahr + " Fahrenheit");
22    }
23 }
```

Como você pode perceber pela listagem, a implementação é bastante simples e envolve basicamente solicitar o valor em graus Celsius (**linha 13**), convertê-lo para **double** (**linha 14**) não se esquecendo de tratar a possibilidade de erro de conversão (**catch** na **linha 15**) e, finalmente, na **linha 20** calcular o equivalente em Fahrenheit pela aplicação da fórmula.

Rpare que, como os graus e suas conversões podem ser obtidas em precisão decimal, declaramos as variáveis auxiliares como **double** e usamos **Double.parseDouble** para converter o valor informado.

Encerramos exibindo o resultado ao usuário na **linha 21**.

Adivinhe o número

```

1 package adivinhe;
2
3 import java.util.Random;
4
5 import javax.swing.JOptionPane;
6
7 public class Adivinhe {
8
9     public static void main(String[] args) {
10
11         int tentativas = 0;
12         int numeroPalpite = 0;
13         int numeroSorteado;
14         String mensagem;
15
16         Random gerador = new Random();
17         numeroSorteado = gerador.nextInt(50) + 1;
18
19         mensagem = "Pensei em um número de 1 a 50. "
20             + "Qual é? (digite 0 para desistir) ";
21
22     do {
23         String numeroString = JOptionPane.showInputDialog(null, mensagem);
24         try {
25             numeroPalpite = Integer.parseInt(numeroString);
26             if (numeroPalpite > 0) {
27                 if (numeroSorteado < numeroPalpite) {
28                     mensagem = "Meu número é MENOR que este...Tente de novo.";
29                 } else if (numeroSorteado > numeroPalpite) {
30                     mensagem = "Meu número é MAIOR que este...Tente de novo.";
31                 }
32                 tentativas++;
33             }
34         } catch (NumberFormatException e) {
35             JOptionPane.showMessageDialog(null, "Digite apenas números!");
36         }
37     } while (numeroPalpite != 0 && numeroPalpite != numeroSorteado);
38
39     String resultado = (numeroPalpite == numeroSorteado ? "acertou" :
40         "desistiu de acertar");
41
42     JOptionPane.showMessageDialog(null, "Você " + resultado + " o "
43         + "número " + numeroSorteado + " depois de "
44         + tentativas + " tentativa(s).");
45 }
46
47 }
```

- **Linhas 11 a 14:** aqui declaramos as variáveis que vamos precisar no nosso programa. Temos três variáveis int: uma para armazenar a quantidade de **tentativas** do usuário (inicializada com 0); uma para o número informado pelo usuário (**numeroPalpite**); e outra para guardar o número sorteado (**numeroSorteado**). Temos também uma String, chamada **mensagem**, que vai conter o texto das mensagens que exibiremos.
- **Linhas 16 e 17:** nestas linhas instanciamos um objeto da classe **Random**, chamado **gerador**, que será usado para o sorteio do número. Na linha 17, obtemos um valor aleatório de 0 a 49 (passando o valor 50 como parâmetro de **gerador.nextInt**) e somamos 1 a esse resultado, para assim obter efetivamente um número de 1 a 50, que é guardado na variável **numeroSorteado**.
- **Linha 19:** armazenamos a primeira mensagem que mostraremos ao usuário na variável **String** com este mesmo nome, e que declaramos na linha 14.
- **Linha 22:** não sabemos quantos palpites o usuário dará, mas ao menos um será feito (ou pelo menos precisamos dar uma chance para ele digitar 0 e encerrar o programa). Por isso, um laço do tipo **do...while** é ideal para este caso, já que garante que pelo menos uma janela de entrada de dados será apresentada ao usuário. Repare que este laço se encerra na **linha 37** e que sua condição de continuidade é que o palpite seja diferente de zero **e também** (por isso o && - operador booleano AND) diferente do número sorteado. Essa condição será quebrada quando o usuário digitar 0 ou acertar o número.
- **Linha 23:** exibimos a **mensagem** ao usuário e armazenamos o valor informado por ele em **numeroString**.
- **Linha 24:** iniciamos um bloco **try...catch** para converter o valor informado como **String** para int.
- **Linha 25:** fazemos a conversão de **String** para int usando o método **parseInt** da classe Integer. No caso de falha na conversão (uma **exceção**), o fluxo de execução do programa será desviado para a cláusula **catch**, na linha 34.
- **Linhas 26 a 33:** o primeiro **if** é usado para confirmar se o usuário deu um palpite ou digitou zero para finalizar. Dentro dele, caso o usuário tenha realmente dado um palpite, temos um outro **if** para testar se o número sorteado é inferior (linha 27) ou superior (linha 29) ao sorteado, atribuindo à **String mensagem** a informação que será passada. Na linha 32, incrementamos o contador de tentativas.
- **Linhas 34 a 36:** este bloco **catch** exibe uma mensagem de aviso caso tenha ocorrido um erro de conversão (reconhecido como uma exceção denominada **NumberFormatException** em Java) na linha 25.
- **Linha 37:** como mencionado anteriormente, o laço do...while será quebrado quando o usuário acertar digitar 0 ou acertar o número.
- **Linha 39:** após sair do laço, precisamos avisar o usuário sobre o que aconteceu. Das duas, uma: ou ele desistiu, ou acertou o número. Em ambos, precisamos mostrar também quantas tentativas foram feitas. Nesta linha, declaramos uma variável **String** chamada **resultado**. Usaremos ela posteriormente para montar a mensagem que será passada ao usuário. Sua única função é conter a razão do término do laço: se o usuário acertou o número, armazenamos nela o texto "acertou"; se o usuário desistiu, guardamos nela a mensagem "desistiu de acertar". Poderíamos fazer isso com um **if...else** simples, mas também podemos simplificar ainda mais usando o **operador ternário**, que conhecemos na seção 3.2, na Aula 3. Ficou curtinho, não é mesmo? E funciona mesmo!

- **Linha 42:** encerramos o programa mostrando ao usuário a mensagem final. Note que, usando a variável **resultado** criada na linha 39 e o operador de concatenação de **Strings** (+), a mensagem será criada adequadamente tanto no caso acerto quanto no de desistência.

## Histograma

Veja na listagem a seguir e nos comentários uma das soluções para o exercício do Histograma.

Se você realizou a contagem de ocorrências de sorteio para cada número em variáveis individuais (usando, por exemplo, **contador1**, **contador2**, ..., **contador10**), repare na listagem como o uso de um vetor torna o código mais compacto e eficiente.

```

1 package histograma;
2
3 import java.util.Random;
4 import javax.swing.JOptionPane;
5
6 public class GeradorDeHistograma {
7
8     public static void main(String[] args) {
9
10        Random random = new Random();
11        int histograma[] = new int[10];
12        StringBuilder construtorDeString = new StringBuilder();
13
14        construtorDeString.append("HISTOGrama\n");
15
16        for (int i = 0; i < 100; i++) {
17            histograma[random.nextInt(10)]++;
18        }
19
20        for (int i = 0; i < 10; i++) {
21            construtorDeString.append((i + 1) + ":" );
22            for (int j = 0; j < histograma[i]; j++)
23                construtorDeString.append("#");
24            construtorDeString.append("\n");
25        }
26
27        JOptionPane.showMessageDialog(null, construtorDeString);
28    }
29 }
```

- **Linha 10:** começamos declarando e instanciando nosso objeto gerador de números aleatórios, proveniente da classe **Random**. Neste caso, ele se chamará **random** (em letras minúsculas).
- **Linha 11:** declaramos e inicializamos um vetor de inteiros com dez posições, chamado **histograma**. Na linguagem Java os índices de um vetor começam por zero. Dessa forma, seu primeiro elemento será **histograma[0]** e seu último (o décimo) será **histograma[9]**. Em Java, os elementos de vetores **int** são inicializados contendo 0.
- **Linha 12:** como a **String** com o histograma será gerada por meio de várias concatenações, o ideal é utilizar para tanto um objeto da classe **StringBuilder**. Declaramos aqui o objeto **construtorDeString** para essa finalidade.
- **Linha 14:** concatenamos ao **construtorDeString**, usando o método **append**, o título informativo "HISTOGRAMA", junto de uma quebra de linha ("\\n").
- **Linhas 16 a 18:** por incrível que pareça, dentro deste pequeno laço fazemos o sorteio dos 100 números de 1 a 10 e também guardamos no vetor **histograma** a quantidade sorteada de cada um. Como? Primeiramente, note na linha 16 que nosso laço é um **for** com um contador **i**, que inicia em 0 vai até 99 (obedecendo a condição **i < 100**). Isso nos garante cem sorteios. O que precisamos é incrementar em uma unidade a posição do vetor **histograma** relativa ao número sorteado. É justamente isso que fazemos: veja que realizamos um pós-incremento (operador **++**) no vetor **histograma**, exatamente no índice correspondente ao número sorteado. E para o código não ficar redundante, é o próprio sorteio (**random.nextInt(10)**) que serve como índice para a posição adequada no vetor a ser incrementada. Não entendeu? Vamos usar um exemplo passo-a-passo:
  - o A linha diz: **histograma[random.nextInt(10)]++;**
  - o A primeira coisa que acontece na linha é a execução do comando mais interno, ou seja, a chamada ao método **nextInt()** dentro dos colchetes do índice do vetor. Passamos como parâmetro o número 10 a ele, o que significa o retorno de um valor de 0 a 9. "Mas não tinha que ser de 1 a 10?", você pergunta. Sim, mas você deve concordar que 0 a 9 correspondem efetivamente a 10 números possíveis: podemos interpretar o 0 como 1 e o 9 como 10. Logicamente, o resultado é o mesmo, e dessa forma também facilitamos o acesso ao respectivo índice no vetor;
  - o Suponha que **random.nextInt(10)** sorteie o número 5. Isso implica que a linha 17 equivalerá a: **histograma[5]++;**
  - o Concluímos que será incrementado o valor existente na posição 5 do vetor, que para nós equivale ao número 6 (assim como o elemento 0 equivale ao 1, e o 9 ao 10).
  - o Essa abordagem evita o (desnecessário) uso de **if** ou **switch...case** para determinar qual posição do vetor será incrementada, deixando o código mais elegante e eficiente.
- **Linhas 20 a 25:** feito os cem sorteios, temos no nosso vetor **histograma** a quantidade de vezes que cada número foi sorteado. Esse laço é usado justamente para montar a **String** que funcionará como nosso "gráfico em modo texto". Começamos com um laço que vai de 0 a 9, para que possamos percorrer todo o vetor **histograma**. Na linha 21, concatenamos ao **construtorDeString** o número da posição atual do contador somado de 1, seguido de ":". Isso faz com que, no primeiro elemento do vetor seja impresso "1: " (nós estamos guardando os números como 0 a 9, mas o usuário espera ver o resultado como 1 a 10). Precisamos agora imprimir uma quantidade de cerquilhas relativas ao número de vezes que 1 (0, nos "bastidores" do programa) foi sorteado. Isso está guardado em **histograma[0]**. Na linha 22, usamos um segundo laço for, que utilizará um contador do tipo **int** chamado **j**, o qual vai de 0 até o valor anterior ao existente em **histograma[i]** (condição **j**

< **histograma[i]**). Assim, se na posição 0 do vetor histograma estiver o número 5 (indicando que o primeiro número foi sorteado cinco vezes), nosso laço interno repetirá também cinco vezes (de 0 a 4). Dentro dele, na linha 23, concatenamos uma cerquilha à **construtorDeString** para cada iteração. Nesse exemplo, então, serão concatenadas cinco cerquilhas. Quando todas as cerquilhas necessárias forem impressas, o laço interno termina. Antes de passar à próxima posição do vetor, adicionamos à **construtorDeString** uma quebra de linha, para que possamos obter a visualização adequada do histograma. Assim acontecerá sucessivamente, até terminarmos de percorrer **histograma**.

- **Linha 27:** finalizado o laço principal, nosso “histograma TXT” está pronto. Mostramos ele ao usuário com **JOptionPane.showMessageDialog**, finalizando nosso programa.

## Módulo 2

### Forca

Há várias formas de se implementar corretamente o exercício da Forca. A maneira apresentada aqui é apenas uma delas, e leva em consideração a aplicação de boas práticas de programação Java que você está conhecendo no curso, como o encapsulamento.

Vamos começar com o código para a classe Forca.

```

56
57     forcas[5] = " ----- \n"
58     + " _ | | \n"
59     + " _ 0 | \n"
60     + "/ \\| | \n"
61     + "/ \\| | \n"
62     + "     ===\n";
63
64     forcas[6] = " ----- \n"
65     + " _ | | \n"
66     + " _ 0 | \n"
67     + "/ \\| | \n"
68     + "/ \\| | \n"
69     + "     ===\n";
70
71 }
72
73     public String getEstagioEnforcamento() {
74         return forcas[erros];
75     }
76
77     public StringBuilder getPalavraExibida() {
78         return palavraExibida;
79     }
80

```

- **Linha 7:** começamos declarando um vetor de **Strings** chamado **palavras**. Nele, colocamos um conjunto de palavras que podem ser usadas no jogo. Seguindo o princípio do encapsulamento, esta variável é declara-

da como privada, pois não deverá ser acessada externamente à classe.

- **Linha 10:** declaramos um novo vetor de **Strings**, chamado **forcas**, com sete posições. Ele será usado para armazenar os “desenhos” ASCII para os sete estágios de “enforcamento”.
- **Linhas 12 e 13:** na linha 12 é declarada uma variável privada int para a contagem de erros. Já na **linha 13**, declaramos uma variável contendo o índice da palavra sorteada para o jogo, dentro do vetor **palavras** declarado na **linha 7**.
- **Linha 14:** à medida que o usuário vai fazendo seus acertos, exibiremos a ele uma versão parcial da palavra sorteada, a **palavraExibida**. Esta palavra começará apenas com traços no lugar das letras, que serão trocadas a medida que usuário acerta uma letra existente. Como teremos que manipular o seu conteúdo, usamos aqui classe **StringBuilder** ao invés de simplesmente usar uma **String** padrão, cujo conteúdo é imutável.

- Linha 16 e 17:** iniciamos aqui o construtor para a classe **Forca** e inicializamos a contagem de erros com zero.
- Linhas 18 e 19:** instanciamos um objeto da classe **Random** e usamos o mesmo para sortear um número entre zero e o tamanho do vetor de palavras. Guardamos o resultado do sorteio em **indiceDaPalavraSorteada**, para que saibamos qual será a palavra que o jogador deverá adivinhar.
- Linha 20:** no início do jogo, a palavra estará totalmente escondida do usuário – serão exibidos apenas traços no lugar das letras em **palavraExibida**. Fazemos isso criando uma cópia da palavra sorteada, onde todos seus caracteres são trocados por traços usando o método **replace** das **Strings**.
- Linhas 22 a 27:** guardamos na primeira posição do vetor **forcas** o “desenho”, em formato de caracteres ASCII, para a força vazia. Para que ele seja impresso corretamente, note que ao final de cada linha colocamos uma quebra (**\n**).

```

28     forcas[1] = " ----- \n"
29     + " |   | \n"
30     + " 0   | \n"
31     + "    | \n"
32     + "    | \n"
33     + "    | \n";
34
35
36     forcas[2] = " ----- \n"
37     + " |   | \n"
38     + " 0   | \n"
39     + " /|   | \n"
40     + "    | \n"
41     + "    | \n";
42
43     forcas[3] = " ----- \n"
44     + " |   | \n"
45     + " 0   | \n"
46     + "/|\  | \n"
47     + "    | \n"
48     + "    | \n";
49
50     forcas[4] = " ----- \n"
51     + " |   | \n"
52     + " 0   | \n"
53     + "/|\\" | \n"
54     + "    | \n"
55     + "    | \n";

```

No trecho que comprehende as linhas entre 28 a 55, temos mais quatro estágios do “enforcamento”, cada um na sua respectiva posição do vetor **forcas**.

Veja um detalhe importante na **linha 53**: como colocamos uma barra invertida (**\**) para representar o braço esquerdo do enforcado, precisamos repeti-la para que o Java não a interprete como o início de um caractere de controle (como o **\n**, por exemplo). Por ter adicionado a segunda barra, também colocamos um espaço a mais nesta linha, de forma que ela fique com tamanho proporcional às demais.

```

56
57     forcas[5] = " ----- \n"
58     + " |   | \n"
59     + " 0   | \n"
60     + "/|\\" | \n"
61     + "/     | \n"
62     + "     ===\n";
63
64     forcas[6] = " ----- \n"
65     + " |   | \n"
66     + " 0   | \n"
67     + "/|\\" | \n"
68     + "/ \\\" | \n"
69     + "     ===\n";
70
71 }
72
73 public String getEstagioEnforcamento() {
74     return forcas[erros];
75 }
76
77 public StringBuilder getPalavraExibida() {
78     return palavraExibida;
79 }
80

```

Nas **linhas 57 a 69**, temos a definição dos dois últimos estágios da força. Na **linha 71**, fechamos o construtor para **Forca**, iniciado na **linha 16**.

Para poder exibir o estágio atual de enforcamento ao jogador, criamos um *getter* nas **linhas 73 a 75**. Ele retorna a posição do vetor **forcas** - que é inicializado no construtor da classe - na posição relativa à quantidade de erros.

Nas **linhas 77 a 79**, criamos um *getter* para exibir ao usuário aquilo que ele já descobriu da palavra secreta (e que está em **palavraExibida**).

```

81 public void testarLetra(String letra) {
82     if (palavras[indicePalavraSorteada].contains(letra)) {
83         for (int j = 0; j < palavras[indicePalavraSorteada].length(); j++) {
84             if (palavras[indicePalavraSorteada].charAt(j) == letra.charAt(0)) {
85                 palavraExibida.replace(j, j + 1, letra);
86             }
87         }
88     } else {
89         erros++;
90     }
91 }
92
93 public int getErros() {
94     return erros;
95 }
96
97 public String getPalavraSecreta() {
98     return palavras[indicePalavraSorteada];
99 }
100
101 }

```

Nas **linhas 81 a 91**, temos o método vital para o jogo: **testarLetra**. Sua função é receber como parâmetro a letra que o usuário “chutou”, verificar se a mesma existe na palavra secreta e, em caso positivo, atualizar a palavra exibida para mostrar onde a letra informada se encontra.

Primeiro, na **linha 82**, usamos o método **contains()** da **String** para verificar se a palavra sorteada possui a letra informada.

Em caso positivo, o bloco entre as **linhas 83 e 87** é executado. Dentro dele, percorremos cada um dos caracteres existentes na palavra sorteada, comparando-os com o que foi digitado pelo usuário. Se coincidir, substituímos na **palavraExibida** o traço pelo caractere, na sua posição correspondente (**linha 85**), usando o método **replace()** existente em objetos do tipo **StringBuilder**.

No caso da letra informada não existir na palavra sorteada, incrementamos o contador de erros na **linha 89**.

Nas **linhas 93 a 95**, temos um *getter* para retornar a quantidade de erros do usuário. Ele será necessário para futuramente testarmos o fim do jogo na classe **Principal**.

Caso o usuário não acerte a palavra, o *getter* definido entre as **linhas 97 e 99** será usado na classe **Principal** para exibir a palavra secreta ao jogador.

Você deve ter notado que nossa implementação da **Forca** traz apenas os *getters* necessários, e nenhum *setter*. Os valores das variáveis internas são modificados pelo método **testarLetra**. Dessa forma, cumprimos corretamente a prática do encapsulamento, escondendo do usuário da classe aquilo que não precisará ser utilizado.

Podemos passar agora à implementação da interface do jogo, na classe **Principal**.

```

1 package forca;
2
3 import java.awt.Font;
4
5 import javax.swing.JTextArea;
6 import javax.swing.JOptionPane;
7
8 public class Principal {
9
10    public static void main(String[] args) {
11
12        JTextArea textoForca = new JTextArea();
13        textoForca.setFont(new Font("Monospaced", Font.BOLD, 20));
14
15        Forca forca = new Forca();
16
17        do {
18            textoForca.setText(forca.getEstagioEnforcamento() + forca.getPalavraExibida());
19            String letra = JOptionPane.showInputDialog(null, textoForca);
20            if (letra.equals("")) {
21                letra = "#";
22            }
23            letra = letra.substring(0, 1).toUpperCase();
24            forca.testarLetra(letra);
25        } while ((forca.getErros() < 6) && (forca.getPalavraExibida().toString().contains("-")));
26

```

Começamos declarando na **linha 12** um objeto **JTextArea**, chamado **textoForca**. Trata-se de uma “área de texto” usada no framework de janelas do Java, o Swing. Como dissemos na apresentação do exercício, vamos usá-lo para que o “desenho de traços” da forca seja mostrado corretamente ao usuário, usando uma fonte de largura fixa (e que é definida na **linha 13**).

Na **linha 15**, declaramos e inicializamos uma instância da classe **Forca**, no objeto **forca**. O laço iniciado na **linha 17**, e que vai até a **linha 25**, controla a execução do jogo. Você pode perceber pela condição de continuidade do laço, na linha 25, que o mesmo será encerrado somente se a quantidade de erros do usuário chegar a 6 (neste caso, o jogador não conseguiu acertar a letra) ou se a palavra exibida não contiver mais traços (o que ocorrerá quando o jogador acertar a palavra).

Dentro do laço, repetimos a exibição do estágio de enforcamento e da palavra sendo adivinhada (**linha 18**), assim como fazemos a solicitação de uma letra. A letra informada pelo usuário é então testada pelo método **testarLetra**, no objeto **forca** (**linha 24**). Fazemos também uma pequena validação: caso o usuário não forneça uma letra, consideramos que uma cerquilha foi digitada, resultando em erro na adivinhação e aumentando o “enforcamento”.

```

27     String resultado;
28
29     if (forca.getErros() < 6) {
30         resultado = "\nVOCÊ DESCOBRIU A PALAVRA!";
31     } else {
32         resultado = "\nVOCÊ FOI ENFORCADO...";
33     }
34
35     textoForca.setText(forca.getEstagioEnforcamento() + forca.getPalavraSecreta() + resultado);
36     JOptionPane.showMessageDialog(null, textoForca);
37 }
38 }
39 }
40 }
```

O encerramento do laço implica em vitória ou derrota do jogador. Para exibir uma mensagem adequada a ele, testamos a quantidade de erros na **linha 29** e criamos uma mensagem correspondente para cada caso.

Finalmente, nas **linhas 35 e 36**, mostramos o resultado, junto da palavra secreta e estágio final da forca.

## Módulo 3

### Complementando o sistema bancário

Começamos implementando a classe **Conta**, seguindo as especificações do exercício:

```

1 package banco.modelo;
2
3 import java.text.DecimalFormat;
4
5 public class Conta {
6
7     public int numero;
8     public Cliente cliente;
9     public double saldo;
10
11    static int contador;
12
13    public Conta(Cliente cliente) {
14        contador++;
15        numero = contador;
16        saldo = 0;
17        this.cliente = cliente;
18    }
19
20    public String listarDados() {
21        return "NÚMERO: " + numero + "\n" +
22               "CORRENTISTA: " + cliente.nome + "\n" +
23               "SALDO: " + DecimalFormat.getCurrencyInstance().format(saldo);
24    }
25
26 }
```

Você pode notar a semelhança entre a implementação do número sequencial da conta e do código do Cliente - o conceito fundamental é o mesmo: o uso de um campo estático.

Um grande diferencial neste código está no uso de um campo referenciando um objeto de outra classe. Essa é uma das maneiras que implementamos **associações** entre classes em Java. O campo cliente, declarado na **linha 8**, receberá um objeto Cliente, sendo usado para identificar o correntista.

Note no método **listarDados (linha 20 a 24)** que retornamos o nome do correntista usando o campo cliente e, dentro deste, seu campo nome. Isso é perfeitamente possível, em virtude do campo nome possuir a visibilidade **public** dentro da classe **Cliente**.

Uma outra novidade é o uso da classe **DecimalFormat** do Java, na **linha 23**. Usamos ela para que o valor double do saldo seja exibido em formato financeiro, usando as configurações regionais do sistema.

Agora, para testar o uso da classe, você pode comentar as linhas existentes no método main da classe principal (para poder recuperá-las depois) e acrescentar as seguintes:

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

    //Cliente cliente1 = new Cliente("Anita", "Marília", "São Paulo");
    //Cliente cliente2 = new Cliente("Marcos", "Garça", "São Paulo");

    //JOptionPane.showMessageDialog(null, cliente1.listarDados());
    //JOptionPane.showMessageDialog(null, cliente2.listarDados());

    //JOptionPane.showMessageDialog(null, "Possuímos " +
    //    Cliente.qtdClientes() + " cliente(s) cadastrados");

    Cliente cliente = new Cliente();
    cliente.nome = JOptionPane.showInputDialog(null, "Nome do Cliente: ");
    cliente.cidade = JOptionPane.showInputDialog(null, "Cidade do Cliente: ");
    cliente.estado = JOptionPane.showInputDialog(null, "Estado do Cliente: ");

    JOptionPane.showMessageDialog(null, "DADOS DO CLIENTE\n\n" +
        cliente.listarDados());

    Conta conta = new Conta(cliente);

    JOptionPane.showMessageDialog(null, "DADOS DA CONTA\n\n" +
        conta.listarDados());
}
```

Começamos instanciando um objeto **cliente** e definindo para seus campos os valores de retorno obtidos com as janelas de entrada.

Em seguida, conforme o enunciado do exercício, mostramos a “ficha do cliente”, com uma chamada ao método **listarDados()**.

Instanciamos então um objeto da classe **Conta**, passando ao seu construtor o objeto **cliente** referente ao correntista.

Por fim, mostramos os dados da conta, usando o método **conta.listarDados()**.

## Funcionarios

A implementação deste exercício consiste na escrita da superclasse **Funcionario** e suas subclasses **Horista** e **Vendedor**, utilizando o conceito do polimorfismo ao escrever seus métodos **calcularSalarioDoMes()** e **listarDados()**.

A classe **Funcionario** poderia ser implementada como a listagem a seguir.

```

1 package funcionarios;
2
3 public class Funcionario {
4
5     protected String nome;
6     protected String setor;
7     protected double salarioBase;
8
9     public Funcionario(String nome, String setor, double salarioBase) {
10         super();
11         this.nome = nome;
12         this.setor = setor;
13         this.salarioBase = salarioBase;
14     }
15
16     public String getNome() {
17         return nome;
18     }
19
20     public void setNome(String nome) {
21         this.nome = nome;
22     }
23
24     public String getSetor() {
25         return setor;
26     }
27
28     public void setSetor(String setor) {
29         this.setor = setor;
30     }
31
32     public double getSalarioBase() {
33         return salarioBase;
34     }
35
36     public void setSalarioBase(double salarioBase) {
37         this.salarioBase = salarioBase;
38     }
39
40     public double calcularSalarioDoMes() {
41         return salarioBase;
42     }
43
44     public String listarDados() {
45         return "NOME: " + nome + "\n" +
46                 "SETOR: " + setor + "\n";
47     }
48 }

```

Uma vez que a classe **Funcionario** possuirá duas subclasses, declaramos seus campos **nome**, **setor** e **salarioBase** como **protected**, para que os mesmos possam ser acessíveis pelas subclasses (**linhas 5 a 7**).

Nas **linhas 9 a 14** temos um construtor para **Funcionario** que recebe como parâmetros os atributos básicos da classe e os armazena nos respectivos campos. Os *getters* e *setters* para os campos da classe são definidos nas **linhas 16 a 38**.

O método **calcularSalarioDoMes()**, escrito entre as **linhas 40 e 42**, é uma versão simplificada para o salário do funcionário – ele apenas retorna seu salário base. Nas subclasses **Horista** e **Vendedor**, reescreveremos este método para considerar as diferenças do salário destes dois tipos especializados de **Funcionario**.

A seguir, a implementação da classe **Horista**, subclass de **Funcionario**.

```

1 package funcionarios;
2
3 public class Horista extends Funcionario {
4
5     private int horasTrabalhadasNoMes;
6     private double valorPorHora;
7
8     public Horista(String nome, String setor, double salarioBase,
9                     int horasTrabalhadasNoMes, double valorPorHora) {
10        super(nome, setor, salarioBase);
11        this.horasTrabalhadasNoMes = horasTrabalhadasNoMes;
12        this.valorPorHora = valorPorHora;
13    }
14
15    public int getHorasTrabalhadasNoMes() {
16        return horasTrabalhadasNoMes;
17    }
18
19    public void setHorasTrabalhadasNoMes(int horasTrabalhadasNoMes) {
20        this.horasTrabalhadasNoMes = horasTrabalhadasNoMes;
21    }
22
23    public double getValorPorHora() {
24        return valorPorHora;
25    }
26

```

A classe **Horista** herda a classe **Funcionario**, como podemos ver pelo uso de **extends**, na **linha 3**. Dessa forma, “recebe automaticamente” seus campos e métodos.

Complementando a classe, acrescentamos nas **linhas 5 e 6** dois campos específicos para um funcionário **Horista**, **horasTrabalhadasNoMes** e **valorPorHora**.

Nas **linhas 8 a 13**, temos o método construtor para um **Horista**. Ele recebe todos os parâmetros necessário (inclusive aqueles recebidos de **Funcionario**). Para definição dos atributos genéricos (nome, setor e salário-base), invocamos o construtor da superclasse, usando **super()**. Já a definição dos atributos específicos (**horasTrabalhadasNoMes** e **valorHora**) é feita diretamente nos campos locais da classe.

Entre as linhas 15 e 25 temos alguns dos *getters* e *setters* para os campos locais de **Horista**.

```

27 public void setValorPorHora(double valorPorHora) {
28     this.valorPorHora = valorPorHora;
29 }
30
31 @Override
32 public double calcularSalarioDoMes() {
33     return horasTrabalhadasNoMes * valorPorHora;
34 }
35
36 @Override
37 public String listarDados() {
38     return super.listarDados() +
39         "HORAS NO MÊS: " + horasTrabalhadasNoMes + "\n" +
40         "VALOR/HORA: " + valorPorHora;
41 }
42
43 }
```

Nas **linhas 27 a 29** é definido o *setter* para o campo **valorPorHora**.

A primeira “novidade” desta classe é apresentada nas **linhas 31 a 34**, onde reescrevemos (ou como preferem alguns, sobrescrevemos) o método **calcularSalarioDoMes**, herdado de **Funcionario**. A anotação **@Override**, existente na **linha 31**, indica ao compilador Java que o método a seguir trata-se de uma reimplementação. Dentro do método, calculamos e retornamos o salário do funcionário **Horista**, obtido por meio da multiplicação da quantidade de horas trabalhadas pelo valor da hora.

A classe é encerrada com outra reimplementação: a do método **listarDados (linhas 46 a 41)**. Basicamente, ele agrupa ao retorno do método existente na superclasses as informações relativas ao **Horista**.

Passamos agora à classe **Vendedor**, uma outra variação de **Funcionario**.

```

1 package funcionarios;
2
3 public class Vendedor extends Funcionario {
4
5     private double totalDeVendasNoMes;
6     private double comissao;
7
8     public Vendedor(String nome, String setor, double salarioBase,
9                     double totalDeVendasNoMes, double comissao) {
10        super(nome, setor, salarioBase);
11        this.totalDeVendasNoMes = totalDeVendasNoMes;
12        this.comissao = comissao;
13    }
14
15    public double getTotalDeVendasNoMes() {
16        return totalDeVendasNoMes;
17    }
18
19    public void setTotalDeVendasNoMes(double totalDeVendasNoMes) {
20        this.totalDeVendasNoMes = totalDeVendasNoMes;
21    }
22
23    public double getComissao() {
24        return comissao;
25    }
26

```

A primeira parte de **Vendedor** segue os mesmos conceitos usados em **Horista**: herança de **Funcionario**, campos privados locais, seus respectivos *getters* e *setters* e um construtor.

```

27    public void setComissao(double comissao) {
28        this.comissao = comissao;
29    }
30
31    @Override
32    public double calcularSalarioDoMes() {
33        return salarioBase + totalDeVendasNoMes * comissao;
34    }
35
36    @Override
37    public String listarDados() {
38        return super.listarDados() +
39                "VENDAS NO MÊS: " + totalDeVendasNoMes + "\n" +
40                "COMISSÃO: " + comissao * 100 + "%";
41    }
42
43
44 }

```

A implementação de **calcularSalarioDoMes** de **Vendedor**, definida nas **linhas 31 a 34**, retorna o salário usando uma regra específica para vendedores: o salário total é composto pelo salário base acrescido da comissão em função das vendas realizadas.

Nas **linhas 36 a 41** temos a reimplementação de **listarDados**. Nela, agregamos ao retorno da versão existente na superclasse (**Funcionario**) os dados relativos ao Vendedor.

Veja na próxima listagem um exemplo de utilização destas classes.

```

1 package funcionarios;
2
3 import javax.swing.JOptionPane;
4
5 public class Principal {
6
7     public static void main(String[] args) {
8
9         Funcionario generico = new Funcionario("Genérico", "Setor", 750);
10        Funcionario horista = new Horista("Maria", "Educação", 0, 40, 45);
11        Funcionario vendedor = new Vendedor("Marcos", "Vendas", 750, 5000, 0.05);
12
13        JOptionPane.showMessageDialog(null, generico.listarDados() + "\n" + generico.calcularSalarioDoMes());
14        JOptionPane.showMessageDialog(null, horista.listarDados() + "\n" + horista.calcularSalarioDoMes());
15        JOptionPane.showMessageDialog(null, vendedor.listarDados() + "\n" + vendedor.calcularSalarioDoMes());
16    }
17
18 }
```

Repare que no método **main** da classe **Principal** declaramos três objetos da classe **Funcionario** (**generico**, **horista** e **vendedor**), mas para cada um deles usamos um construtor diferente!

Todos são funcionários, mas apenas **generico** é um **Funcionario** comum, já que instanciamos este objeto usando o construtor da superclasse; **horista**, apesar de declarado como **Funcionario**, é instaciado usando o construtor da classe **Horista**; e para vendedor, também declarado como **Funcionario**, usamos o construtor de **Vendedor**. Como um **Horista** e um **Vendedor** também são **Funcionários**, as declarações e inicializações realizadas nas **linhas 9 a 11** são perfeitamente válidas!

Agora repare no que será exibido nos **JOptionPane** existentes nas **linhas 13 a 15**. Apesar de declarados como objetos da classe **Funcionario**, as versões de **listarDados** e **calcularSalarioDoMes** para **horista** e **vendedor** serão aquelas das classes pelas quais foram instanciados, **Horista** e **Vendedor**!

Essa é mais uma amostra da flexibilidade que temos em uma linguagem orientada a objetos como Java, graças à herança e ao polimorfismo.

## Módulo 4

### Contatos

Este exercício serve como prática do uso de coleções de objetos em Java, por meio da classe **ArrayList**. Começamos criando um classe **Contato**, contendo os dados que serão guardados sobre uma pessoa na “agenda virtual” do usuário.

```

1 package contatos;
2
3 public class Contato {
4
5     private String nome;
6     private String telefone;
7     private String email;
8
9     public Contato(String nome, String telefone, String email) {
10         this.nome = nome;
11         this.telefone = telefone;
12         this.email = email;
13     }
14
15     public String getNome() {
16         return nome;
17     }
18
19     public void setNome(String nome) {
20         this.nome = nome;
21     }
22
23     public String getTelefone() {
24         return telefone;
25     }
26

```

Temos neste trecho a declaração dos campos privados **nome**, **telefone** e **email** (**linhas 5 a 7**), a definição do construtor (**linhas 9 a 13**) e de *getters* e *setters* para nome e telefone, seguindo os padrões do encapsulamento.

```

27     public void setTelefone(String telefone) {
28         this.telefone = telefone;
29     }
30
31     public String getEmail() {
32         return email;
33     }
34
35     public void setEmail(String email) {
36         this.email = email;
37     }
38
39     @Override
40     public String toString() {
41         return "NOME: " + nome + "\n" +
42                 "TELEFONE: " + telefone + "\n" +
43                 "E-MAIL: " + email;
44     }
45
46 }

```

Neste último trecho, são criados os *getters* e *setters* restantes (entre as **linhas 27 e 37**) e uma novidade: o método **toString** (**linhas 39 a 44**).

O método **toString** é padronizado em Java, e serve para trazer uma “representação **String**” do que o objeto contém.

Toda classe criada em Java descende de uma superclasse comum, denominada **Object**. Nesta classe, há uma implementação padrão de **toString**, que traz apenas um identificador para o objeto. Se o desenvolvedor desejar, pode reimplementar **toString** em sua classe, retornando uma **String** com informações sobre o objeto da maneira que achar mais conveniente. Fazer esse procedimento tem uma grande vantagem: quando passarmos um objeto com uma implementação de **toString** como parâmetro do tipo **String** de algum método (como por exemplo o método de impressão no terminal **System.out.print** ou um **JOptionPane.showMessageDialog**), **toString** será invocado automaticamente, retornando o valor definido na sua implementação.

Vamos ver na prática o resultado de usar **toString** na utilização de **Contato**, como a exemplificada na classe **Principal** a seguir.

```

1 package contatos;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import javax.swing.JOptionPane;
6
7 public class Principal {
8
9     public static void main(String[] args) {
10
11         List<Contato> contatos = new ArrayList<>();
12         String nome, telefone, email, opcao = null;
13
14         do {
15             nome = JOptionPane.showInputDialog(null, "Informe o nome do contato:");
16             telefone = JOptionPane.showInputDialog(null, "Informe o telefone do contato:");
17             email = JOptionPane.showInputDialog(null, "Informe o email do contato:");
18             Contato contato = new Contato(nome, telefone, email);
19             contatos.add(contato);
20             opcao = JOptionPane.showInputDialog(null, "Digite N para criar um novo contato"
21                 + " ou qualquer outra letra para terminar.");
22         } while (opcao.toUpperCase().equals("N"));
23
24         for (Contato umContato : contatos) {
25             JOptionPane.showMessageDialog(null, umContato);
26         }
27     }
28 }
```

Na **linha 11**, declaramos um **ArrayList** de objetos da classe **Contato**, denominado **contatos**, e inicializamos o mesmo usando seu construtor padrão.

Após solicitar os dados básicos de um contato nas **linhas 15 a 17**, instanciamos um objeto **Contato** na **linha 18** e adicionamos o mesmo ao **ArrayList contatos**, na **linha 19**. Tudo isso será feito dentro de

um laço **do...while** (**linhas 14 a 22**), que se repetirá enquanto o usuário optar em continuar adicionando contatos. Cada novo objeto **contato** será guardado dentro do **ArrayList** **contatos** por meio do método **add**.

Finalizado o laço, mostraremos os dados de todos os contatos criados ao usuário. Para isso, usaremos um “**for** especial” existente em Java, que percorre automaticamente elementos em uma lista ou vetor (**linhas 24 a 26**).

Na declaração do **for** (**linha 24**), especificamos que vamos percorrer o **ArrayList** **contatos**, e que cada um de seus elementos (do primeiro ao último) será referenciado pela variável **umContato** – um por vez, para cada iteração do laço.

Dentro do laço, usamos o **JOptionPane.showMessageDialog** para exibir os dados do contato. Repare no segundo parâmetro deste método – ao invés de usar uma **String** ou um método que retorne uma **String** (como fizemos até agora), colocamos diretamente o objeto **umContato**! Como definimos na classe **Contato** um método **toString**, e como este segundo parâmetro “pede uma **String**”, o **toString** do objeto será invocado automaticamente. Assim, conseguimos o mesmo efeito que antes tínhamos com o método **listarDados**!

Como prática adicional, procure agora abrir os exercícios anteriores e trocar o **listarDados** ou por um **toString**, deixando seu código ainda mais “enxuto”!

## Considerações Finais

Chegamos ao final de mais uma etapa deste curso. Parabéns!

Nesta disciplina, pudemos “entrar de cabeça na linguagem Java”, conhecendo inclusive o ambiente de desenvolvimento integrado mais usado no mundo junto com ela: o Eclipse.

Você pôde descobrir como fazer o download, instalar e criar projetos Java no Eclipse, bem como localizar e corrigir erros, além de testar seus aplicativos.

Fizemos uma revisão rápida de estruturas de controle e decisão, levando em conta as peculiaridades da linguagem Java e usando uma abordagem de entrada e saída de dados mais visual.

O ponto principal da linguagem Java, uso de classes e objetos, foi explorado implementando classes do sistema bancário. Com essa abordagem, você pôde ficar conhecendo mais sobre as classes já existentes na API básica da linguagem Java (como **String**, **Random**, **Integer**, **Double** e **StringBuilder**), assim como pôde descobrir como criar suas próprias classes. É assim que seus programas orientados a objetos serão compostos: uma mescla de classes preexistentes com aquelas que você mesmo cria, com base na engenharia de requisitos e a consequente modelagem do sistema usando UML.

Futuramente, você conhecerá outros *frameworks* e tecnologias utilizados para criação dos mais diversos tipos de aplicativos em Java. Em todos eles, você perceberá que as classes essenciais do Java SE e os conceitos fundamentais de engenharia de *software*, orientação a objetos e UML serão utilizados intensamente.

O campo de desenvolvimento Java é imenso e cheio de empolgantes oportunidades - não deixe de aproveitá-las!

**Prof. Querino**

## Referências Bibliográficas

---

ARNOLD, K.; GOSLING, J.; HOLMES, D. **A linguagem de programação Java.** Porto Alegre: Bookman, 2007.

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java.** São Paulo: Pearson Prentice-Hall, 2004.

BLOCH, J. **Effective Java.** Upper Saddle River: Addison-Wesley, 2012.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML: guia do usuário.** Rio de Janeiro: Elsevier, 2012.

DEITEL, H.; DEITEL, P. **Java: como programar.** São Paulo: Prentice-Hall, 2010.

ECLIPSE FOUNDATION. **Eclipse.** [website]. 2014. Disponível em: <http://www.eclipse.org/>, acesso em: 13.abr.2014.

FLANAGAN, D. **Java: o guia essencial.** Porto Alegre: Bookman, 2006.

GOSLING, J.; MCGILTON, H. **The Java language environment.** 1997. Disponível em: <http://www.oracle.com/technetwork/java/intro-141325.html>, acesso em: 10.abr.2014.

HIRONDELLE SYSTEMS. **Collected Java practices.** 2014. Disponível em: <http://www.javapractices.com/>, acesso em: 12.abr.2014.

ORACLE. **Java 7 API.** 2013. Disponível em: <http://docs.oracle.com/javase/7/docs/api/>, acesso em: 10.abr.2014.

\_\_\_\_\_. **Java tutorial.** 2014. Disponível em: <http://docs.oracle.com/javase/tutorial/>, acesso em: 10.abr.2014.

PRESSMAN, R. S. **Engenharia de software.** São Paulo: McGraw-Hill, 2010.

QUERINO FILHO, L. C. **Desenvolvendo seu primeiro aplicativo Android.** São Paulo: Novatec, 2013.

SOMMERRVILLE, I. **Engenharia de software.** São Paulo: Addison-Wesley, 2007.