



PROGRAMAÇÃO I

Professor Dr. Edson A. Oliveira Junior

UNICESUMAR

Av. Guedner, 1610 - Jardim Aclimação
Cep 87050-900 - MARINGÁ - PARANÁ
unicesumar.edu.br
44 3027.6360

UNICESUMAR EDUCAÇÃO A DISTÂNCIA

NEAD - Núcleo de Educação a Distância
Bloco 4 - MARINGÁ - PARANÁ
unicesumar.edu.br
0800 600 6360

as imagens utilizadas neste livro foram obtidas a partir dos sites PHOTOS.COM e SHUTTERSTOCK.COM

FICHA CATALOGRÁFICA

C397 CENTRO UNIVERSITÁRIO DE MARINGÁ. Núcleo de Educação a Distância; **JUNIOR**, Edson A. Oliveira

Programação I. Edson A. Oliveira Junior.

(Reimpressão revista e atualizada)

Maringá-Pr.: UniCesumar, 2016.

167 p.

"Graduação - EaD".

1. Programação I. 2. JAVA . EaD. I. Título.

ISBN 978-85-8084-608-9

CDD - 22 ed. 005.1
CIP - NBR 12899 - AACR/2

Ficha catalográfica elaborada pelo bibliotecário
João Vivaldo de Souza - CRB-8 - 6828

Reitor

Wilson de Matos Silva

Vice-Reitor

Wilson de Matos Silva Filho

Pró-Reitor de Administração

Wilson de Matos Silva Filho

Pró-Reitor de EAD

Willian Victor Kendrick de Matos Silva

Presidente da Mantenedora

Cláudio Ferdinandi

NEAD - Núcleo de Educação a Distância**Direção Operacional de Ensino**

Kátia Coelho

Direção de Planejamento de Ensino

Fabrício Lazilha

Direção de Operações

Chrystiano Mincoff

Direção de Mercado

Hilton Pereira

Direção de Polos Próprios

James Prestes

Direção de Desenvolvimento

Dayane Almeida

Direção de Relacionamento

Alessandra Baron

Head de Produção de Conteúdos

Rodolfo Encinas de Encarnação Pinelli

Gerência de Produção de Conteúdos

Gabriel Araújo

Supervisão do Núcleo de Produção de Materiais

Nádila de Almeida Toledo

Supervisão de Projetos Especiais

Daniel F. Hey

Coordenador de Conteúdo

Danillo Xavier Saes

Design Educacional

Camila Zagüini Silva, Fernando Henrique

Mendes, Nádila de Almeida Toledo

Rossana Costa Giani

Iconografia

Isabela Soares Silva

Projeto Gráfico

Jaime de Marchi Junior

José Jhonne Coelho

Arte Capa

André Morais de Freitas

Editoração

Robson Yuiti Saito

Qualidade Textual

Hellyery Agda, Ana Paula da Silva, Jaquelina Kutsunugi, Keren Pardini, Maria Fernanda Canova Vasconcelos, Nayara Valenciano, Rhaysa Ricci Correa

Ilustração

Robson Yuiti Saito



Professor
Wilson de Matos Silva
Reitor

Viver e trabalhar em uma sociedade global é um grande desafio para todos os cidadãos. A busca por tecnologia, informação, conhecimento de qualidade, novas habilidades para liderança e solução de problemas com eficiência tornou-se uma questão de sobrevivência no mundo do trabalho.

Cada um de nós tem uma grande responsabilidade: as escolhas que fizermos por nós e pelos nossos farão grande diferença no futuro.

Com essa visão, o Centro Universitário Cesumar – assume o compromisso de democratizar o conhecimento por meio de alta tecnologia e contribuir para o futuro dos brasileiros.

No cumprimento de sua missão – “promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária” –, o Centro Universitário Cesumar busca a integração do ensino-pesquisa-extensão com as demandas institucionais e sociais; a realização de uma prática acadêmica que contribua para o desenvolvimento da consciência social e política e, por fim, a democratização do conhecimento acadêmico com a articulação e a integração com a sociedade.

Diante disso, o Centro Universitário Cesumar almeja ser reconhecido como uma instituição universitária de referência regional e nacional pela qualidade e compromisso do corpo docente; aquisição de competências institucionais para o desenvolvimento de linhas de pesquisa; consolidação da extensão universitária; qualidade da oferta dos ensinos presencial e a distância; bem-estar e satisfação da comunidade interna; qualidade da gestão acadêmica e administrativa; compromisso social de inclusão; processos de cooperação e parceria com o mundo do trabalho, como também pelo compromisso e relacionamento permanente com os egressos, incentivando a educação continuada.



Professor
Fabrício Lazilha
Diretoria de
Planejamento de Ensino

Professora
Kátia Solange Coelho
Diretoria Operacional
de Ensino

Seja bem-vindo(a), caro(a) acadêmico(a)! Você está iniciando um processo de transformação, pois quando investimos em nossa formação, seja ela pessoal ou profissional, nos transformamos e, consequentemente, transformamos também a sociedade na qual estamos inseridos. De que forma o fazemos? Criando oportunidades e/ou estabelecendo mudanças capazes de alcançar um nível de desenvolvimento compatível com os desafios que surgem no mundo contemporâneo.

O Centro Universitário Cesumar mediante o Núcleo de Educação a Distância, o(a) acompanhará durante todo este processo, pois conforme Freire (1996): “Os homens se educam juntos, na transformação do mundo”.

Os materiais produzidos oferecem linguagem dialógica e encontram-se integrados à proposta pedagógica, contribuindo no processo educacional, complementando sua formação profissional, desenvolvendo competências e habilidades, e aplicando conceitos teóricos em situação de realidade, de maneira a inseri-lo no mercado de trabalho. Ou seja, estes materiais têm como principal objetivo “provocar uma aproximação entre você e o conteúdo”, desta forma possibilita o desenvolvimento da autonomia em busca dos conhecimentos necessários para a sua formação pessoal e profissional.

Portanto, nossa distância nesse processo de crescimento e construção do conhecimento deve ser apenas geográfica. Utilize os diversos recursos pedagógicos que o Centro Universitário Cesumar lhe possibilita. Ou seja, acesse regularmente o AVA – Ambiente Virtual de Aprendizagem, interaja nos fóruns e enquetes, assista às aulas ao vivo e participe das discussões. Além disso, lembre-se que existe uma equipe de professores e tutores que se encontra disponível para sanar suas dúvidas e auxiliá-lo(a) em seu processo de aprendizagem, possibilitando-lhe trilhar com tranquilidade e segurança sua trajetória acadêmica.

Professor Dr. Edson A. Oliveira Junior

Bacharel em Informática pela Universidade Estadual de Maringá (UEM), Mestre em Ciência da Computação pela Universidade Estadual de Maringá (UEM) e Doutor em Ciências de Computação e Matemática Computacional pelo Instituto de Ciências Matemáticas e de Computação (ICMC) da Universidade de São Paulo (USP). Todos esses títulos foram obtidos na área de concentração de Engenharia de Software, em que minha especialidade é Linha de Produto de Software e Gerenciamento de Variabilidade, além de Arquitetura de Software, Desenvolvimento Baseado em Componentes (DBC), Frameworks, Metamodelagem e Modelagem UML. DBC e Frameworks são temas de pesquisa que envolvem não só a modelagem de sistemas, mas a implementação de sistemas e suas arquiteturas.

Participo de vários cursos de Pós-Graduação em diversas instituições de ensino superior como a própria UEM, UNIPAR, Faculdade Integrado, Unicesumar, Faculdade Alfa, Instituto Paranaense de Ensino e Faculdade Cidade Verde. Possuo as seguintes certificações da Tecnologia Java: Sun Certified Java Associate, Sun Certified Java Programmer, Sun Certified Java Developer, Sun Certified Web Component Developer e Sun Certified Business Component Developer, todas elas certificadas pela Sun Microsystems entre os anos de 2003 e 2007. Ministro esta disciplina em cursos de treinamento técnico, graduação e pós-graduação desde o ano de 2000.

PROGRAMAÇÃO I

SEJA BEM-VINDO(A)!

Prezado(a) acadêmico(a), é com grande satisfação que apresento a você o livro de Programação I (Java). Este material foi elaborado com o objetivo de contribuir em sua formação, especialmente a de desenvolvedor(a) de software. Sou o professor Edson A. de Oliveira Junior, autor deste livro. Você pode ter certeza que este foi preparado com carinho especial para que você possa entender o que essa disciplina pode te trazer de benefício ao longo de sua vida como desenvolvedor(a) e/ou analista de sistemas.

Inicialmente, o livro abordará conceitos básicos da linguagem Java, desde a sua criação até o processo de compilação e execução e, em seguida, serão apresentados os demais conceitos para que você possa ter uma noção de programação orientada a objetos com a linguagem Java. Escrever este material foi um desafio, pois é necessário expressar os conceitos de uma forma clara para que você, caro(a) aluno(a), possa entender a lógica dos exemplos. Assim, espero que você goste do que foi colocado aqui.

Na Unidade I, teremos a apresentação da história e evolução da linguagem Java, desde as suas primeiras versões, em que só existia a linguagem Java puramente e algumas poucas bibliotecas disponíveis. Além disso, será apresentada a tecnologia Java onde a linguagem Java está inserida atualmente e as principais bibliotecas disponíveis para cada plataforma Java de desenvolvimento. O processo de compilação e interpretação de programas Java é apresentado, destacando cada fase e os elementos necessários para que um programa Java possa ser executado.

A Unidade II abordará as estruturas básicas de um programa em Java, como: tipos primitivos, declaração e inicialização de variáveis e escopo de variáveis. Estruturas de controle, repetição e seleção serão apresentadas para permitir que programas Java possam ser executados de forma a terem fluxos de informações diferentes em momentos diferentes de sua execução. Para tanto, estruturas como if, do-while e switch podem ser usadas.

Na Unidade III você entenderá a diferença entre classes e objetos Java. Você verá que uma classe Java serve como um modelo para que os objetos (instâncias) daquela classe possam ser criados e permitir que o programa Java tenha "vida". Mas para entender esses conceitos, serão apresentados os principais elementos que compõem uma classe Java: atributos, métodos, construtores, e o método main.

A Unidade IV abordará um dos conceitos mais importantes em programação orientada a objetos, o encapsulamento. Este conceito serve para proteger os dados de uma classe de acessos indevidos por elementos de outras classes ou até mesmo de outros pacotes. Porém, para entender tal conceito, é necessário aprender os chamados modificadores de acesso da linguagem Java: private, public, protected e default.

APRESENTAÇÃO

Por fim, na Unidade V serão apresentados outros dois conceitos fundamentais em programação orientada a objetos: herança e polimorfismo. Herança permite criar uma hierarquia de classes que possuem um conjunto de atributos e métodos comuns, mas com seus tipos próprios. Já polimorfismo permite criar tal estrutura, porém exigindo que cada tipo especializado tenha as suas próprias versões de implementação de seus métodos, que são diferentes dos métodos herdados.

Lembre-se sempre que programar é uma “arte moderna” em que aquele que detém o maior poder de abstração possível é aquele que melhor saberá desenvolver os seus programas.

Ótima leitura e sucesso em sua vida de desenvolvedor(a) de software.

Professor Dr. Edson A. Oliveira Junior

SUMÁRIO

UNIDADE I

INTRODUÇÃO À LINGUAGEM JAVA

-
- 15 Introdução
 - 15 Histórico e Evolução da Linguagem Java
 - 17 A Tecnologia Java
 - 21 O Processo de Compilação e Interpretação de Programas Java
 - 26 Considerações Finais
-

UNIDADE II

ESTRUTURAS BÁSICAS DA LINGUAGEM JAVA

-
- 33 Introdução
 - 33 O Que são Tipos Primitivos?
 - 36 Declaração e Inicialização de Variáveis
 - 38 Escopo de Variável
 - 42 Estruturas de Controle
 - 44 Estruturas de Seleção
 - 51 Estruturas de Repetição
 - 63 Considerações Finais
-



SUMÁRIO

UNIDADE III

CLASSES x OBJETOS JAVA

-
- 69 Introdução
 - 85 Objetos Java
 - 90 Javabeans ou Pojos
 - 96 Estado e Comportamento
 - 99 Considerações Finais
-

UNIDADE IV

MODIFICADORES JAVA E ENCAPSULAMENTO

-
- 105 Introdução
 - 112 Utilizando Modificadores de Acesso
 - 117 O Modificador **Static**
 - 119 O Modificador **Final**
 - 121 O Modificador **Abstract**
 - 123 Construtores Java
 - 127 Considerações Finais
-



SUMÁRIO

UNIDADE V

HERANÇA E POLIMORFISMO EM JAVA

133 Introdução

133 Herança

140 Polimorfismo

152 Considerações Finais

155 CONCLUSÃO

157 GABARITO

167 REFERÊNCIAS



INTRODUÇÃO À LINGUAGEM JAVA

UNIDADE

I

Objetivos de Aprendizagem

- Entender o que é a linguagem Java e a sua importância.
- Estudar o histórico e a evolução da linguagem Java, além das plataformas que englobam a tecnologia Java.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Histórico e Evolução da Linguagem Java
- A Tecnologia Java
- O Processo de Compilação e Interpretação de Programas em Java

INTRODUÇÃO

Caro(a) aluno(a), nesta primeira unidade trataremos alguns conceitos relacionados à linguagem Java que serão fundamentais para o entendimento das demais unidades.

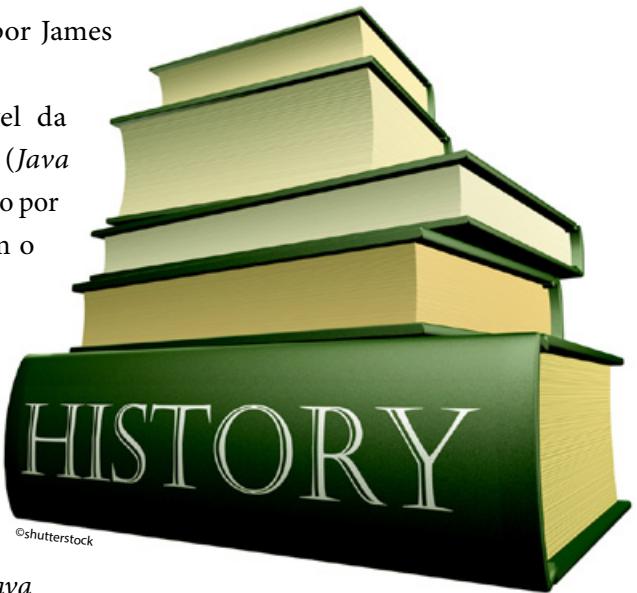
A tecnologia Java, formada pela linguagem Java mais todas as bibliotecas disponíveis para download e uso, tem sido responsável por grande parte do mercado de desenvolvimento de software comercial e acadêmico nas últimas duas décadas. Empresas de médio e grande porte vêm adotando tal tecnologia com base em seus poderosos recursos e características. Java é uma linguagem baseada no paradigma de orientação a objetos, o que facilita a manutenção de sistemas Java quando os princípios básicos de tal paradigma são seguidos.

HISTÓRICO E EVOLUÇÃO DA LINGUAGEM JAVA

A linguagem Java foi criada por James Gosling em 1992.

A primeira versão estável da linguagem Java foi o JDK (*Java Development Kit*) 1.0.2, conhecido por Java 1, em janeiro de 1996 com o codinome Oak.

Em fevereiro de 1997, foram adicionadas algumas bibliotecas, como: eventos com base na biblioteca *Abstract Window Toolkit* (AWT), classes internas, conexão com banco de dados via *Java Database Connectivity* (JDBC) e invocação remota de métodos.



Em dezembro de 1998, foi lançada a J2SE 1.2 (*Java Standard Edition*), codinome *playground* e conhecida simplesmente como Java 2. Esta foi a evolução mais significativa da linguagem Java, já que a tecnologia foi dividida em três principais plataformas: J2SE (*Java 2 Standard Edition*), J2EE (*Java 2 Enterprise Edition*) e J2ME (*Java 2 Micro Edition*).

Ao J2SE foram adicionadas as seguintes bibliotecas: *Swing* para interface gráfica com o cliente, coleções (*List*, *Set* e *Map*), e a possibilidade de criação de pontos flutuantes de acordo com o padrão IEEE 754.

Em maio de 2000, foi lançada a versão 1.3, codinome Kestrel. As mudanças mais notáveis foram: invocação remota de método compatível com CORBA, introdução das bibliotecas JavaSound para tratamento de sons e JNDI (*Java Naming and Directory Interface*) para o compartilhamento de recursos.

Em fevereiro de 2002, foi lançada a versão 1.4 da J2SE, codinome Merlin, que foi a primeira versão desenvolvida sob a tutela da *Java Community Process* (JCP). A JCP¹ é uma comunidade extremamente importante formada por um consórcio de empresas que regulamenta a tecnologia Java de forma geral. Dentre as maiores melhorias, destacam-se: a inclusão de expressões regulares, capacidade de lidar com o protocolo IPv6, biblioteca para *Logging*, integração *parsers XML*, melhoria nas bibliotecas para criptografia e segurança.

Em setembro de 2004 foi lançada a versão J2SE 5.0, codinome Tiger, conhecida como Java 5. Várias mudanças significativas foram incorporadas nesta nova versão, dentre elas: biblioteca para *Generics*, eliminando a necessidade de conversões entre tipos similares, inclusão de uma biblioteca para metadados de uma aplicação, *autoboxing/unboxing*, que são conversões automáticas entre tipos primitivos, *enumerations*, que permitem criar listas de valores ordenados, e a melhoria da estrutura de repetição *for* para coleções e *arrays*.

Em dezembro de 2006, foi lançada a versão Java SE 6, codinome Mustang, conhecida como Java 6. A partir desta versão, as siglas J2SE, J2EE e J2ME foram substituídas pelas siglas Java SE, Java EE e Java ME, respectivamente. Dentre as melhorias, podemos citar: aumento de desempenho da plataforma básica, suporte ao JDBC 4.0, uma biblioteca dedicada somente ao compilador Java e melhorias no desempenho e segurança da máquina virtual Java.

¹Saiba mais sobre a JCP em <<http://www.jcp.org/en/home/index>>. Texto em inglês.

Em julho de 2011, foi lançada a versão Java SE 7, codinome Dolphin, conhecida como Java 7. Algumas características adicionadas a esta versão: estrutura de seleção *switch* aceitando *strings*, e não somente valores inteiros, nova biblioteca para tratar entrada e saída e melhorias nos *streams* para XML e Unicode.

Espera-se para setembro de 2013 a versão Java SE 8 com algumas melhorias adiadas da versão 7. Além disso, a versão Java SE 9 está em discussão e possivelmente oferecerá a possibilidade de programação paralela. Para a versão 10, existem especulações da possibilidade de remover tipos primitivos, tornando Java uma linguagem totalmente orientada a objetos.

A TECNOLOGIA JAVA

O Java é composto por uma série de vários produtos de software e especificações provenientes, originalmente, da *Sun Microsystems*, e hoje sobre responsabilidade da Oracle. Esses produtos juntos fornecem um sistema para o desenvolvimento e a implantação de softwares em ambientes computacionais multiplataforma.

O Java é usado em uma ampla variedade de plataformas computacionais, desde sistemas embarcados e telefones celulares até servidores empresariais e supercomputadores. Menos comum, porém ainda usados, estão os Java *applets* que de vez em quando são utilizados para melhorar a segurança de navegadores Web e computadores de mesa, conhecidos como *desktops*.



Escrever usando a linguagem de programação Java é a principal forma de produzir o código que será implantado na forma de *bytecodes* Java. Existem, ainda, os compiladores de *bytecodes* disponíveis para gerar código para outras linguagens como Ada, JavaScript, Python e Ruby. Várias novas linguagens de programação foram projetadas para serem executadas de forma nativa na máquina virtual Java (*Java Virtual Machine - JVM*), que veremos na seção “A Máquina Virtual Java”, como Scala, Clojure e Groovy. A sintaxe de Java é muito semelhante à de C e C++, porém mais recursos orientados a objetos podem ser modelados.

O Java elimina algumas construções de baixo nível, como ponteiros, além de possuir um modelo de memória muito simples, em que cada objeto é aloocado em uma pilha e todas as variáveis de tipos de objeto são referências. O seu gerenciamento de memória é feito por meio da coleta de lixo (*Garbage Collector*) automática realizada pela JVM.

Uma edição da “plataforma Java” é o nome de um pacote de programas relacionados que permite o desenvolvimento e a execução de programas escritos em Java. A plataforma não é específica para qualquer processador ou sistema operacional. Porém, a sua execução requer uma JVM e um compilador com um conjunto de bibliotecas que são implementadas para diversos hardwares e sistemas operacionais para que os programas em Java possam ser executados de forma idêntica em qualquer ambiente. As seguintes plataformas Java são as mais comuns:

- **Java Card** – uma tecnologia que permite pequenas aplicações baseadas em Java (applets) para serem executadas em cartões inteligentes com segurança e similares de memória em pequenos dispositivos.
- **Java ME (Micro Edition)** – especifica vários conjuntos diferentes de bibliotecas (conhecidos como perfis) para dispositivos com armazenamento, exibição e capacidade de energia limitados. Muitas vezes usado para desenvolver aplicativos para dispositivos móveis, PDAs, set-top boxes de TV e impressoras.
- **Java SE (Standard Edition)** – para uso geral em aplicações, desktops, servidores e dispositivos similares.
- **Java EE (Enterprise Edition)** – é a soma da plataforma Java SE com as mais diversas outras APIs úteis para aplicações multicamadas e cliente-servidor em empresas.

A plataforma Java consiste de vários programas. Cada programa fornece uma parcela de suas capacidades gerais. Por exemplo, o compilador Java, que converte código-fonte Java em *bytecode* Java (uma linguagem intermediária para a JVM), é fornecido como parte do *Java Development Kit* (JDK). O *Java Runtime Environment* (JRE) complementa a JVM com um compilador *just-in-time* (JIT), que converte *bytecodes* intermediários no código de máquina nativo da plataforma alvo. Um extenso conjunto de bibliotecas também forma a plataforma Java.

Assim, os componentes essenciais da plataforma Java são: o compilador Java, as bibliotecas e o ambiente de tempo de execução em que o *bytecode* Java intermediário “executa” de acordo com as regras estabelecidas na especificação da máquina virtual.

O coração da plataforma Java é a máquina virtual, que executa programas de *bytecode* Java. Este código é o mesmo, não importa em que sistema operacional ou hardware o programa está sendo executado. O compilador JIT traduz o *bytecode* Java em instruções do processador nativo em tempo de execução e armazena o código nativo em memória durante a execução.

O uso do *bytecode* como linguagem intermediária permite que os programas Java possam rodar em qualquer plataforma que tenha uma máquina virtual disponível. O uso de um compilador JIT permite que aplicações Java, depois de um pequeno atraso durante o carregamento e uma vez “prontas”, tendam a ser executadas tão rápido como os programas nativos. Desde a versão 1.2 da JRE, a implementação de Java incluiu um compilador *just-in-time*, em vez de um interpretador.

Embora os programas Java sejam multiplataforma ou independente de plataforma, o código da máquina virtual em que estes programas rodam não é. Cada plataforma operacional possui a sua própria JVM.

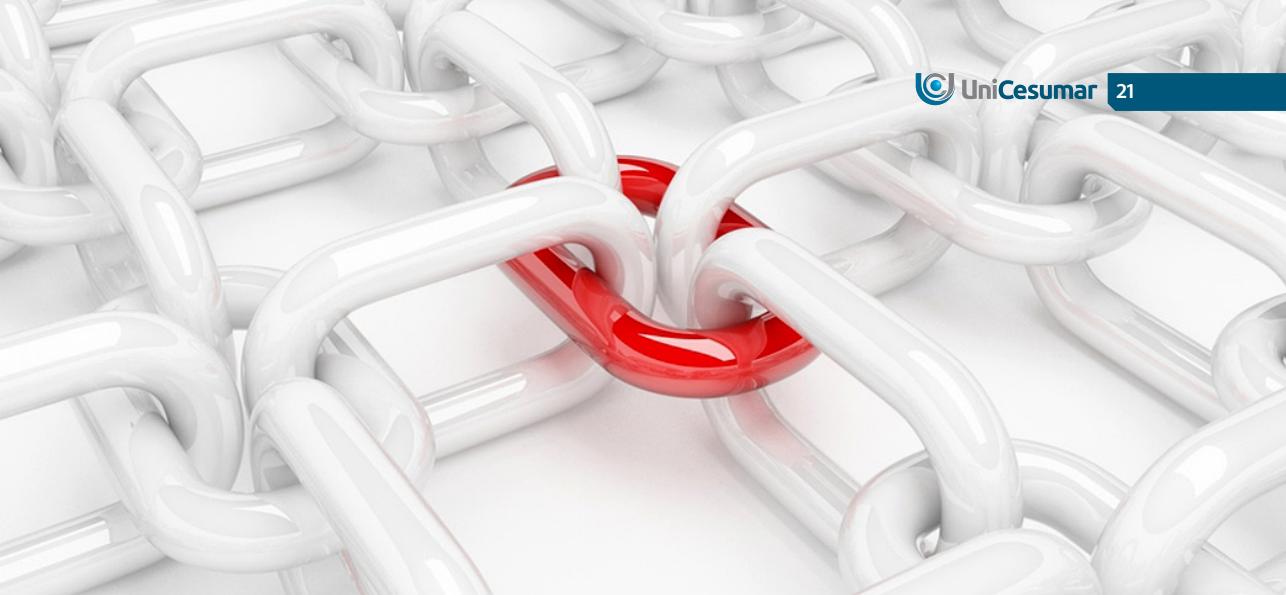
Em muitos sistemas operacionais modernos, um conjunto extenso de código reutilizável é fornecido para simplificar o trabalho do programador. Normalmente, esse código é fornecido como um conjunto de bibliotecas carregáveis dinamicamente, em tempo de execução. Como a Plataforma Java não é dependente de qualquer sistema operacional específico, os aplicativos não podem confiar em qualquer biblioteca de sistema operacional pré-existente. Em vez disso, a plataforma Java fornece um conjunto amplo de suas próprias bibliotecas-padrão

contendo, grande parte, as mesmas funções reutilizáveis comumente encontradas em sistemas operacionais modernos. A maior parte do sistema de bibliotecas também é escrita em Java. Por exemplo, biblioteca *Swing* desenha a interface do usuário e controla os eventos em si, eliminando muitas diferenças sutis entre diferentes plataformas, como lidar com componentes semelhantes.

As bibliotecas Java servem a dois propósitos principais. Primeiro, assim como outras bibliotecas de código-padrão, as bibliotecas Java fornecem ao programador um conjunto bem conhecido de funções para realizar tarefas comuns, como a manutenção de listas de itens ou a análise de uma *String* complexa. Segundo, as bibliotecas de classe fornecem uma interface abstrata para tarefas que normalmente dependem fortemente dos sistemas de hardware e operacional. Tarefas como acesso à rede e acesso a arquivos estão muitas vezes fortemente entrelaçadas com as implementações distintas de cada plataforma. As bibliotecas *java.net* e *java.io* implementam uma camada de abstração em código nativo do sistema operacional, então fornecido para a interface padrão das aplicações Java.

De acordo com a Oracle, o JRE é encontrado em mais de 850 milhões de computadores. A Microsoft não tem fornecido um JRE com seus sistemas operacionais desde que a *Sun Microsystems* processou a Microsoft por adicionar classes específicas do Windows ao pacote do ambiente de execução Java.

Alguns aplicativos Java estão em uso em desktop, incluindo ambientes de desenvolvimento integrados, como NetBeans e Eclipse, e os clientes de compartilhamento de arquivos como o Limewire e Vuze. O Java também é usado no ambiente de programação matemática MATLAB para tornar a interface do usuário mais amigável. O Java fornece interface de usuário multiplataforma para algumas aplicações, como o Lotus Notes.



©shutterstock

Reprodução proibida. Art. 184 do Código Penal e Lei 9.610 de 19 de fevereiro de 1998.

O PROCESSO DE COMPILAÇÃO E INTERPRETAÇÃO DE PROGRAMAS JAVA

O JDK inclui muitos utilitários para compilação, depuração e execução de aplicações Java.

O COMPILADOR JAVA

Uma aplicação Java deve ter pelo menos uma classe que contenha um método chamado main(), o qual contém o primeiro código a ser executado para iniciar a aplicação.

A classe, a seguir, é um bom exemplo, ela simplesmente mostra a frase “Olá, Mundo!” na saída padrão, que neste caso é o console:

```
public class OlaMundo {  
    public static void main(String[ ] args) {  
        System.out.println("Olá, Mundo!");  
    }  
}
```

O compilador Java simplesmente converte arquivos-fonte Java em *bytecodes*. O uso do compilador Java é o seguinte:

```
javac [opções] [arquivo fonte]
```

A forma mais direta é como se segue:

```
javac OlaMundo.java
```

Isso resultará em um arquivo *bytecode* com o mesmo nome do arquivo .java, mas com a extensão .class: OlaMundo.class. É possível, ainda, informar a pasta de destino do *bytecode* com a opção –d.

O CLASSPATH

O *Path* (caminho em inglês) é uma variável de ambiente de um sistema operacional que fornece a uma aplicação uma lista de pastas onde procurar por algum recurso específico.

O exemplo mais comum é o caminho para programas executáveis. A variável de ambiente CLASSPATH de Java é uma lista de locais que são visitados na procura por arquivos de classes. Tanto o interpretador Java como o compilador Java usa a CLASSPATH quando procura por pacotes e classes Java.

Um elemento de *classpath* pode ser uma pasta, um arquivo .JAR ou um arquivo .ZIP. A forma exata de configurar a *classpath* varia de sistema para sistema. Em um sistema baseado no Unix (incluindo o Mac OS X), você pode configurar a variável de ambiente CLASSPATH com uma lista de locais separados por dois pontos. Por exemplo:

```
% CLASSPATH=/home/ze/Java/classes:/home/maria/lib/foo.jar:  
% export CLASSPATH
```

Nesse exemplo, estamos especificando uma *classpath* com três locais: uma pasta chamada classes, um arquivo .JAR e a pasta atual que é especificada com um ponto.

No sistema Windows, a variável de ambiente CLASSPATH é configurada com uma lista de locais separada por ponto e vírgula. Por exemplo:

```
C:\> set CLASSPATH=D:\users\joao\Java\classes;E:\apps;
```

O inicializador Java e outras ferramentas de linha de comando sabem como encontrar as classes essenciais, as quais são incluídas na instalação Java. As classes nos pacotes java.lang, java.io, java.net, e java.swing, por exemplo, não precisam ser incluídas na *classpath*.

Para encontrar outras classes, o compilador e o interpretador Java procuram os elementos da *classpath* na ordem em que foram inseridos. A busca combina os locais e os nomes dos pacotes das classes. Por exemplo, imagine a estrutura de pacote e classe animais.passaros.Galinha. Procurar o local /usr/lib/java incluído na *classpath*, significa procurar por /usr/lib/java/animais/passaros/Galinha.class. Procurar pelo local /home/maria/classesuteis.jar incluído na *classpath*, significa procurar pelo arquivo /home/maria/classesuteis.jar e dentro dele procurar pelo arquivo compactado animais/passaros/Galinha.class.

Se você deseja compilar sua aplicação com pacotes e classes definidos por você mesmo(a), terá que dizer à JVM onde procurar os pacotes e classes na *classpath*. Essa inclusão da informação de localização da classe na *classpath* é feita dizendo ao compilador onde a classe ou pacote desejado estão com as opções de linha de comando *-cp* ou *-classpath*.

Por exemplo:

```
javac -cp terceiros\classes;\home\maria\classesuteis.jar;. AloMundo.java
```

Essa linha de comando diz que além das classes *core*, qualquer classe ou pacote que esteja nos locais citados na lista separada por ponto e vírgula pode ser utilizado na compilação de AloMundo.java.

Note que não é necessário incluir na *classpath* da compilação o local das classes *core*, como já foi dito, e também locais já definidos na variável de ambiente CLASSPATH.

Para conhecer outras opções do compilador javac, basta digitar javac –help ou simplesmente javac sem nenhum complemento. Uma lista com as opções será mostrada.

O INTERPRETADOR JAVA

A interpretação de arquivos *bytecode* Java é a base para a criação de aplicações Java.

A forma de utilizar o aplicativo java.exe para interpretar arquivos de *bytecode* (.class) é a seguinte:

```
java [-opções] nome_da_classe [argumentos]
```

Interpretação do *bytecode*

O interpretador Java é chamado com o aplicativo java.exe (no Windows). Ele é usado para interpretar o *bytecode* (arquivo .class) e executar o programa.

O nome da classe deve ser especificado de forma completa, incluindo pacotes, se houver. Alguns exemplos:

```
% java animais.passaros.Galinha  
% java AloMundo
```

O interpretador procura pela classe na *classpath*. Por outro lado, você pode querer definir onde se encontram certos pacotes ou classes que sejam importantes para a interpretação (como classes de apoio). Você pode encontrar pacotes ou classes em tempo de execução incluindo a opção –cp ou –classpath com o interpretador. A utilização é a mesma que no comando javac visto anteriormente.

O interpretador Java pode ser utilizado com a opção `-jar` para executar um arquivo container .JAR. Por exemplo:

```
% java -jar batalhanaval.jar
```

Nesse caso, o arquivo .JAR se torna o *classpath* e a classe que contém o método `main()` dentro dela é considerada o programa a ser executado.

A ASSINATURA DO MÉTODO MAIN()

O método `main()` deve possuir a assinatura de método correta. A assinatura de um método é um conjunto de informações que define o método. Ela inclui o nome do método, seus argumentos e o tipo de retorno, assim como o modificador de visibilidade e tipo. O método `main()` deve ser público (*public*), estático (*static*) e receber um *array* de objetos *Strings* (textos, nesse caso sem espaço) e não deve retornar valor indicando com a palavra reservada *void*. Assim:

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

O fato de `main()` ser público e estático simplesmente significa que ele é acessível globalmente e que ele pode ser chamado diretamente pelo nome. Quem o chama é o inicializador quando interpretamos o *bytecode*.

O único argumento do método `main()` é um *array* de objetos *Strings*, que serve para armazenar em cada entrada do *array* os parâmetros digitados pelo usuário após o nome da classe a ser interpretada. O nome do parâmetro pode ser escolhido pelo usuário (escolhemos “argumentos” no exemplo acima); mas o tipo deve ser sempre `String[]`, que significa *array* de objetos *String*.

Se, por exemplo, interpretarmos uma classe que aceita argumentos da seguinte forma:

```
java ClasseExemplo arroz feijão macarrão
```

Então teremos a primeira posição do *array*, argumentos[0] igual a “arroz”, a segunda posição, argumentos[1] igual a “feijão” e a terceira posição do *array*, argumentos[2] igual a “macarrão”. Repare que *arrays* têm sua indexação começada por zero.

CONSIDERAÇÕES FINAIS

Nesta primeira unidade, foi apresentada a história e a evolução da linguagem Java, os elementos que compõem a tecnologia Java e o processo de compilação, interpretação e execução de um arquivo Java.

Esses conceitos são importantes do ponto de vista geral, enquanto você, caro(a) aluno(a), deve dominá-los com o intuito de entender grande parte do que acontece desde o momento em que programamos usando Java até o momento em que o programa criado é executado.

O Java é uma linguagem e uma tecnologia muito poderosa, que pode trazer grandes benefícios ao desenvolvedor, desde que este tenha o domínio dos conhecimentos básicos. Assim, espero que esta primeira unidade tenha lhe fornecido algum conhecimento novo e que você possa desfrutar de todo o restante do livro.

ATIVIDADES



1. Por que o Java divulga tanto que os seus programas são “*write once, run everywhere*”?
2. Quais as principais características do Java 5? O que mudou nos programas já existentes até o momento do seu lançamento?
3. O que é JCP? Como você pode fazer para submeter uma mudança que você considera importante para futuras versões de Java?
4. Quais plataformas compõem a tecnologia Java? Forneça uma pequena descrição de cada uma delas.
5. Explique o que é *bytecode* e como ele deve ser “lido” para permitir que um programa Java seja executado.
6. Explique, com as suas palavras, o processo de compilação, interpretação e execução de programas Java. Dica: faça uma ilustração.
7. Por que o *classpath* é tão importante para o Java?

MATERIAL COMPLEMENTAR



NA WEB

Saiba mais sobre programação paralela

Por Domingos Manoel Oran Barros Coelho Júnior

Ultimamente os processadores de computadores e notebooks estão sendo fabricados com mais de um núcleo. Isso requer um olhar especial por quem desenvolve software e o deseja fazer com qualidade.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://www.devmedia.com.br/programacao-paralela/21405>>. Acesso em: 03 abr. 2013.



NA WEB

Programação Paralela Multicore

Disponível em: <<http://www.youtube.com/watch?v=UjgzYVEG9I0>>.



NA WEB

O que é a tecnologia Java e por que é necessária?

Por Oracle, Inc.

Java é uma linguagem de programação e uma plataforma de computação lançada pela primeira vez pela Sun Microsystems em 1995. É a tecnologia que capacita muitos programas da mais alta qualidade, como utilitários, jogos e aplicativos corporativos, entre muitos outros, por exemplo. O Java é executado em mais de 850 milhões de computadores pessoais e em bilhões de dispositivos em todo o mundo, inclusive telefones celulares e dispositivos de televisão.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <http://www.java.com/pt_BR/download/faq/whatis_java.xml>. Acesso em: 03 abr. 2013.



NA WEB

Java está em toda parte.

Disponível em: <<http://www.youtube.com/watch?v=sd4KPlF8eAo>>.



MATERIAL COMPLEMENTAR



NA WEB

O processo interpretação e compilação: Entendendo o Java de uma forma diferente

Por Oracle, Inc.

Entenda como o Java é compilado e interpretado neste artigo que ilustra todas as etapas com exemplos práticos.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://www.devmedia.com.br/processo-de-interpretacao-e-compilacao-entendendo-o-javade-uma-forma-diferente/24257>>.



NA WEB

Codificando, compilando e executando um programa Java

Disponível em: <<http://www.youtube.com/watch?v=M9LCDfRTyQ0>>.



NA WEB

Introdução à linguagem Java – Capítulo 1 do Livro “Programação Java com ênfase em orientação a objetos”

Por Douglas R. Mendes

Saiba mais sobre o assunto no capítulo completo.

Disponível em: <<http://www.novateceditora.com.br/livros/javaoo/capitulo9788575221761.pdf>>.

REFLITA

O Java é uma linguagem totalmente orientada a objetos!



ESTRUTURAS BÁSICAS DA LINGUAGEM JAVA

UNIDADE

II

Objetivos de Aprendizagem

- Estudar as estruturas Básicas da Linguagem Java.
- Estudar Tipos Primitivos.
- Estudar Declaração e Inicialização de Variáveis.
- Estudar Escopo de Variáveis.
- Estudar Estruturas de Controle.
- Estudar Estruturas de Repetição.
- Estudar Estruturas de Seleção.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- O que são tipos primitivos
- Declaração e inicialização de variáveis
- Escopo de variável
- Estrutura de controle
- Estrutura de repetição
- Estrutura de seleção

INTRODUÇÃO

Caro(a) aluno(a), nesta unidade iremos compreender alguns dos elementos básicos da linguagem Java. Abordaremos os aspectos fundamentais das palavras-chaves, tipos primitivos, variáveis e escopo de variáveis, que são responsáveis por informar como acessar uma variável de outra classe.

Também abordaremos estruturas de controle, repetição e seleção. Essas estruturas de controle são uma parte-chave de quase toda linguagem de programação e a linguagem Java oferece várias maneiras de executá-las. Algumas, como as instruções *if* e *for*, são comuns à maioria as linguagens.

Para a implementação de todos os exemplos, usaremos a plataforma de desenvolvimento NetBeans IDE e a ferramenta *Astah Community* para realizar todas as modelagens UML de nosso projeto. A UML (*Unified Modeling Language* - Linguagem de Modelagem Unificada) é uma linguagem visual utilizada para modelar sistemas computacionais orientados a objeto. Neste trabalho, faremos uso do diagrama de atividades que apresenta muita semelhança com fluxogramas utilizados para desenvolver a lógica de programação e determinar o fluxo de controle de um algoritmo.

O QUE SÃO TIPOS PRIMITIVOS?

Toda a linguagem de programação possui uma maneira de tratar os dados que serão manipulados em um sistema. Para tanto, esses dados devem ser armazenados em tipos de dados que correspondam ao seu valor.

Definir um tipo de dado mais adequado para ser armazenado em uma variável é uma questão de grande importância para garantir a resolução do problema. Ao desenvolver um algoritmo, é necessário que se tenha conhecimento prévio do tipo de informação (dado) que será utilizado para resolver o problema proposto. Daí escolhe-se o tipo adequado para a variável que representará esse valor.

As variáveis devem ser declaradas respeitando-se a sintaxe básica “tipo nome-Variavel;”. Há oito tipos de dados primitivos em Java e vamos compreender um pouco mais sobre cada um a seguir.

TIPOS INTEIROS

Os tipos inteiros são para números sem partes fracionárias. Valores negativos são permitidos. O Java fornece os quatro tipos de números inteiros mostrado na tabela abaixo:

byte	Representam números inteiros de 8 bits (1 byte). Podem assumir valores entre -128 a 127.
short	Representam números inteiros de 16 bits (2 bytes). Podem assumir valores entre -32.768 até 32.767.
int	Representam números inteiros de 32 bits (4 bytes). Podem assumir valores entre -2.147.483.648 até 2.147.483.647.
long	Representam números inteiros de 64 bits (8 bytes). Podem assumir valores entre -9.223.372.036.854.775.808 até 9.223.372.036.854.775.807.

Tabela 1 – Tipos inteiros

Fonte: o autor

Na maioria das situações, o tipo *int* é o mais prático. Os tipos *byte* e *short* são essencialmente utilizados para aplicativos especializados, como tratamento de arquivo de baixo nível, ou para grandes *arrays* quando há pouco espaço de armazenamento.

O tipo *long* deve ser identificado com a letra “L” para não ser “compactado” pela linguagem em um tipo inteiro. A compactação ocorre como uma maneira de reduzir o uso de memória.

Tipos de ponto flutuante

Os tipos de ponto flutuante denotam números com partes fracionárias. Os dois tipos de ponto flutuante são mostrados na tabela abaixo:

float	Representam números reais de 32 bits com precisão simples. Podem assumir valores de ponto flutuante com formato definido pela especificação IEEE 754.
double	Representam números reais de 64 bits com precisão dupla. Assim como o float, podem assumir valores de ponto flutuante com formato definido pela especificação IEEE 754.

Tabela 2 – Tipos flutuantes

Fonte: o autor

O nome *double* refere-se ao fato de que esses números são duas vezes mais precisos do que o tipo *float*. As únicas razões para utilizar *float* são aquelas raras situações em que o processamento é importante ou quando você precisa armazenar um grande número deles. Números do tipo *float* têm sufixo “F” (por exemplo, 3.454F). Números de ponto flutuante sem um sufixo F sempre são considerados como o tipo *double*. Opcionalmente, você pode fornecer o sufixo “D” (3.22D).

O tipo **char**

O tipo *char* é utilizado para descrever caracteres individuais. Por exemplo ‘A’ é um caractere de constante com um valor 65. Ele é diferente de “A”, uma string que contém um único caractere.

char	Representam notação de caracteres de 16 bits (2 bytes) para formato Unicode UTF-16. Podem assumir caracteres entre '\u0000' a '\uffff' e valores numéricos entre 0 a 65535.
------	---

Tabela 3 – Tipos char

Fonte: o autor

O tipo **boolean**

O tipo *boolean* tem dois valores, *false* e *true*. Ele é utilizado para avaliar condições lógicas. Você não pode converter entre números inteiros e valores *boolean*.

boolean	Representam apenas 1 bit de informação (0 ou 1). Podem assumir apenas os valores <i>true</i> e <i>false</i> .
---------	---

Tabela 4 – Tipos boolean

Fonte: o autor

DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

DECLARAÇÃO DE VARIÁVEIS

Java é uma linguagem de programação fortemente tipada. Isto significa que cada variável deve ter um tipo de dado associado a ela. Por exemplo, uma variável pode ser declarada para usar um dos oito tipos de dados primitivos: *byte*, *short*, *int*, *long*, *float*, *double*, *char* ou *boolean*, que são os tipos primitivos que vimos anteriormente.

O nome de uma variável pode começar com uma letra, um sublinhado “_”, ou cifrão \$. Se o nome da variável começar com um sublinhado, o segundo caractere deve ser uma letra alfabética. Veja, abaixo, alguns exemplos de declarações de variáveis.

```
11 public class TiposPrimitivos {  
12  
13     float pi;  
14     double tamanho;  
15     char estadoCivil;  
16     boolean aprovado;  
17     short s;  
18     byte b;  
19     int _variavel1;  
20     long $variavel2;
```

Figura 1 – Declaração de variáveis

O nome não pode começar com um dígito (0, 1, 2, 3, 4, 5, 6, 7, 8 ou 9). Veja, a seguir, um exemplo de tentativa de declaração de uma variável iniciando com dígito.

```
23 int 8teste;
```

Figura 2 – Declaração de uma variável incorretamente

Observe que o NetBeans IDE destacou o nome incorreto de variável.

Depois do primeiro caractere, o nome da variável pode incluir letras, dígitos (0, 1, 2, 3, 4, 5, 6, 7, 8 ou 9) ou sublinhados, em qualquer combinação. O nome de uma variável também não pode ser nenhuma das palavras-chaves da linguagem Java. As palavras-chaves são palavras especiais, reservadas em Java. Elas têm um significado para o compilador que as usa para determinar o que seu código-fonte está tentando fazer. Veja no quadro abaixo a lista das 49 palavras-chaves reservadas.

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

INICIALIZANDO VARIÁVEIS

Antes de uma variável ser utilizada, ela deve receber um valor inicial. Isto é chamado de inicialização de variável. Se tentar usar uma variável sem antes a inicializar, como no exemplo abaixo, ocorrerá um erro:

```
13     int numero;
14     // tente adicionar o valor 5 para a variavel numero
15     numero = numero + 5;
```

Figura 3 – Inicialização de variável incorretamente

O compilador irá lançar um erro informando que a variável número não pode ser inicializada.

Para inicializar uma variável, usamos uma instrução de atribuição. Uma instrução de atribuição segue o mesmo padrão de uma equação de matemática (por exemplo, a equação $2 + 2 = 4$). Para dar um valor à variável, o lado esquerdo tem de ser o nome da variável e o lado direito o valor atribuído, conforme figura abaixo:

```
14      int numero;
15      //iniciando a variável
16      numero = 0;
17
```

Figura 4 – Inicialização de variáveis

No exemplo acima, o número foi declarado com um tipo de dados *int* e foi inicializado com 0 (zero). Podemos agora adicionar 5 para a variável número, pois ela já foi inicializada:

```
14      int numero;
15      //iniciando a variável
16      numero = 0;
17      numero = numero + 5;
18
```

Figura 5 – Inicialização de variáveis

Tipicamente, a inicialização de uma variável é feita ao mesmo tempo que a sua declaração:

```
14      int numero = 0;
15
16      numero = numero + 5;
```

Figura 6 – Inicialização de variáveis

ESCOPO DE VARIÁVEL

Muitos erros de compilação são gerados porque os programadores não possuem uma imagem clara de quanto tempo as variáveis estão disponíveis e quando elas podem ser acessadas. O conceito escopo de variável descreve a vida de uma variável, ou seja, refere-se à acessibilidade de uma variável. É a parte do programa em que o nome da variável pode ser referenciado. Você pode declarar variável

em vários lugares diferentes do mesmo código. Abaixo serão listados vários escopos de variável para que você possa compreender mais sobre esse assunto tão importante.

VARIÁVEIS DE INSTÂNCIA (ATRIBUTOS)

As variáveis de instâncias são definidas dentro da classe, mas fora de qualquer método, e só são inicializadas quando a classe é instanciada. Elas são os campos que pertencem a cada objeto. Por exemplo, o código abaixo define campos (variáveis de instância) para a classe Livro.

```
11 public class Livro {  
12  
13     long preco;  
14     int quantidade;  
15     char tipo;  
16  
17 }
```

Figura 7 – Variáveis de instância

A classe Livro do código anterior informa que cada instância sua terá seu próprio preço, quantidade e tipo. Em outras palavras, cada instância poderá ter seus próprios valores exclusivos para esses três campos.

Variáveis Locais

As variáveis locais são as declaradas dentro de um método. Da mesma forma que a variável local inicia sua existência dentro do método, ela também é eliminada quando a execução do método é concluída. Não há palavra-chave especial que designa uma variável como local; a determinação vem inteiramente a partir do local em que a variável é declarada - que é entre as chaves de abertura e fechamento do método.

As variáveis locais só são visíveis para os métodos em que são declaradas, não são acessíveis a partir do resto da classe. Por isso elas não utilizam a maioria dos modificadores e acessos como *public*, *abstract*, *static* etc., porém ela pode

fazer uso do modificador final.

A variável local precisa ser inicializada para ser usada. A sintaxe para declarar uma variável local e inicializá-la é semelhante ao declarar um campo. Veja na imagem abaixo um exemplo de variável local.

```
11  public class Livro {  
12  
13      public void metodoContar() {  
14          int contador = 20;  
15          contador = contador + 1;  
16      }  
17  }
```

Figura 8 – Variável local

Observe que a variável contador foi declarada dentro do método metodoContar(). E ela foi inicializada na mesma linha onde foi declarada. O compilador rejeitará qualquer tentativa de utilização de uma variável a qual não tenha sido atribuído um valor, porque – diferente das variáveis de instância – as variáveis locais não recebem valores padrão.

A variável contador do exemplo acima não pode ser referenciada em nenhum código externo ao método onde foi declarada. Para compreendermos mais, vamos analisar a figura a seguir:

```
13  public void metodoContar() {  
14      int contador = 20;  
15      contador = contador + 1;  
16  }  
17  
18  //O código abaixo não será compilado.  
19  // o compilador não tem acesso a variável contador  
20  public void imprimir(){  
21      System.out.println(contador);  
22  }  
23 }
```

Figura 9 – Tentativa de acesso de variável local

Observe que no segundo método imprimir() a variável contador foi destacada pelo NetBeans IDE acusando um erro de acesso.

Variáveis de Classe (Estáticas)

Variáveis de classe, também conhecidas como variáveis estáticas, são declaradas com a palavra-chave *static* em uma classe, mas fora de um método construtor ou um bloco. O modificador *static* diz ao compilador que há apenas uma cópia de cada variável de classe por classe, independentemente de quantos objetos são criados a partir dele.

A ideia por trás do conceito de variável estática é a de que objetos de uma mesma classe podem e, em algumas situações, devem compartilhar valores em comum, caso contrário, teríamos de criar atributos que precisariam ser atualizados todos ao mesmo tempo, e em cada objeto criado.

Vejamos o seguinte exemplo: um sistema de venda de livros possui um desconto que é ajustado regularmente de acordo com a inflação e repassado a todas as vendas de livros do sistema, quando tivermos de fazer um ajuste ou atualizar o valor do desconto, teríamos de fazê-lo em todos os livros cadastrados. Porém, se essa variável não “pertencer” aos objetos, e sim à própria classe, todos os livros saberão do novo desconto, pois se baseiam no mesmo valor. Observe na figura a seguir o exemplo de um código com a variável *static*.

```

11  public class Livro {
12
13      //variavel estatica representando 20% desconto
14      public static int DESCONTO = 20;
15
16      public void vender(double valor){
17          double novoValor = valor - (valor * DESCONTO)/100;
18      }
19 }
```

Figura 10 – Variáveis estáticas

Neste caso, a variável quando é estática funciona mais ou menos como uma variável do tipo global em linguagens como Pascal ou Delphi. Embora no Java não exista um conceito de variável global.

Então quando a classe é criada, a variável DESCONTO é inicializada com 20 e na chamada do método *vender()* ela é calculada gerando assim um novo valor para o livro.

COMPARANDO OS ESCOPOS DE VARIÁVEL

A tabela abaixo apresenta uma comparação resumida entre os escopos de variável vistos anteriormente e os principais pontos que devem ser lembrados.

	Local	Instância (atributos)	Classe (estáticas)
Onde a Declarar	Dentro de um método	Dentro de uma classe, fora de todos os métodos, mas sem <i>static</i>	Dentro de uma classe, fora de todos os métodos, com <i>static</i>
Proprietário	Método em que é declarado	Cada objeto (instância da classe) é dono de uma cópia	Classe onde ele é declarado
Tempo de vida	Apenas enquanto o método estiver em execução	Enquanto o objeto que o conter existir	Enquanto a classe que o conter estiver carregada
Onde é utilizável	Apenas dentro do método em que é declarado	Em todos os métodos de instância da classe	Em todos os métodos da classe

Tabela 5 – Comparando o escopo de variáveis

Fonte: o autor

ESTRUTURAS DE CONTROLE

Quando você compila e executa um programa Java, a JVM executa o que está dentro do método principal. Estas instruções são realizadas sequencialmente. Um programa de computador (algoritmo) é como uma receita: uma sequência de instruções:

Leia um valor X;
Calcule Y como função de X: $Y = X * 2$;
Imprima o resultado.

Estruturas de controle são instruções que permitem que blocos específicos de código sejam escolhidos para serem executados, redirecionando determinadas partes do fluxo do programa. O Java contém três tipos de estruturas de controle: instruções de sequência; instrução de seleção (*if, if else, switch*) e instruções de repetição (*while, do-while, for*).

INSTRUÇÕES DE SEQUÊNCIA

Normalmente instruções em um programa são executadas uma após a outra na ordem em que são escritas. Somente programas muito simples são estritamente sequenciais. Vejamos o diagrama de atividades UML que modela um fluxo de sequência em Java com apenas 2 instruções.

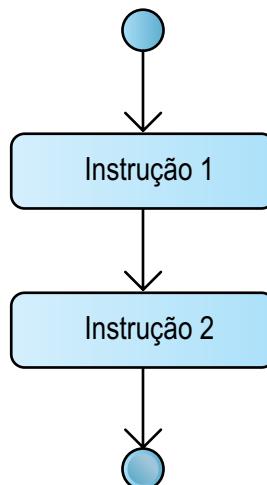


Figura 11 – Fluxograma da estrutura sequencial
Fonte: o autor

O círculo sólido na parte superior representa o estado inicial do nosso programa, os retângulos são as diretrizes (instruções), as setas representam as transições e o círculo sólido cercado por um círculo vazio representa o estado final do nosso programa. Esse padrão de modelagem será visto em outros exemplos desse material.

ESTRUTURAS DE SELEÇÃO

INSTRUÇÕES *IF* E *ELSE*

A linguagem Java disponibiliza os comandos *if* e *else* que, de forma seletiva, definem qual bloco de comandos deverá ser executado. Se a condição do comando *if* for avaliada como verdadeira será executado o bloco de comandos dentro do *if*. Abaixo temos a estrutura do comando *if*.

```
if (expressão){  
    diretiva  
}
```

Nesse código, todas as instruções colocadas entre chaves serão executadas se forem verdadeiras. Abaixo, podemos observar o fluxograma que representa o funcionamento do comando *if*.

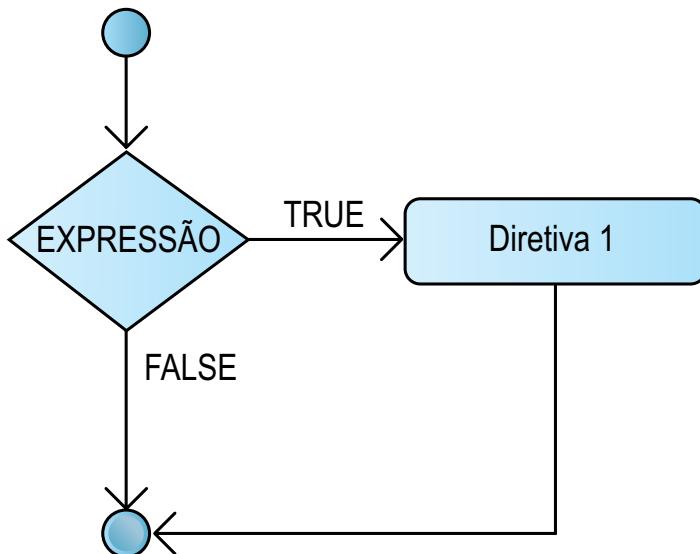


Figura 12 – Fluxograma do comando if

Fonte: o autor

Diretivas são as instruções que a linguagem Java fornece para a construção de programas. Elas são separadas por ; (ponto e vírgula).

Vejamos um exemplo de um código utilizando a instrução *if*.

```

11  public class EstruturaControle {
12      //exemplo comando if
13
14      public static void main(String args[]) {
15
16          int valor1 = 5;
17          int valor2 = 10;
18
19          if (valor1 > valor2) {
20              System.out.println("valor 1 é maior que valor 2");
21          }
22
23          System.out.println("Soma dos valores: " + (valor1 + valor2));
24      }
25  }
  
```

Figura 13 – Programa com instrução if

Observe que temos uma expressão comparando o *valor1* com o *valor2*, no caso, a expressão retornou um valor verdadeiro, o que fez com que a diretiva fosse executada. Agora vejamos a estrutura de condição *if else*.

```
if (expressão) {
    Diretiva 1
} else {
    Diretiva 2
}
```

Veja, a seguir a representação gráfica da estrutura de seleção *if* e *else*. Observe que cada diretiva é executada dependendo do resultado da expressão.

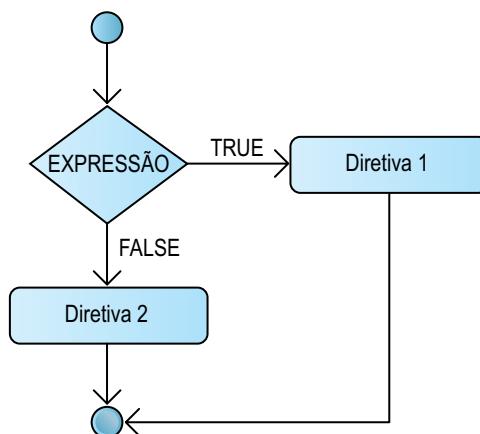


Figura 14 – Fluxograma do comando *if* e *else*

Fonte: o autor

Agora vejamos um programa utilizando *if* e *else*:

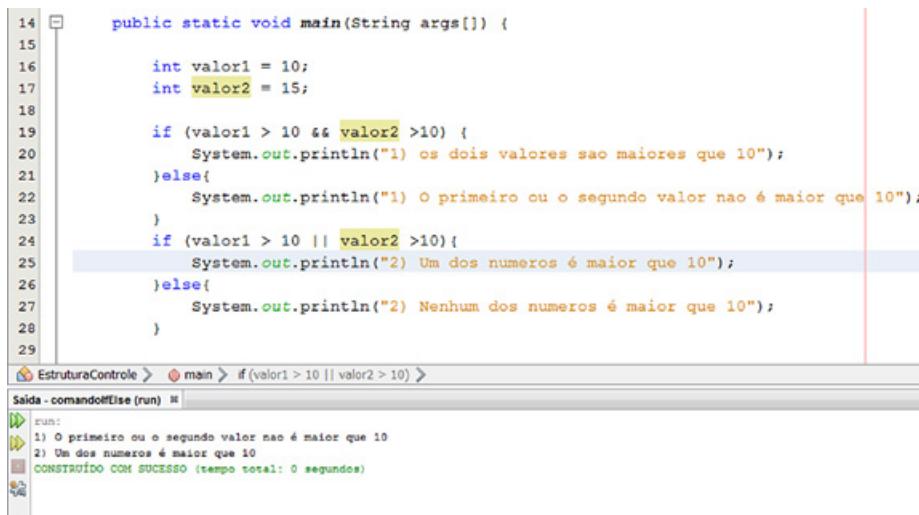
```
9  public class EstruturaControle {
10     //exemplo comando if e else
11
12     public static void main(String args[]) {
13         int valor1 = 5;
14         int valor2 = 10;
15         if (valor1 > valor2) {
16             System.out.println("valor 1 é maior que valor 2");
17         } else {
18             System.out.println("valor 2 é maior que valor 1");
19         }
20     }
21 }
22 }
```

Figura 15 – Programa com a instrução *if* e *else*

A expressão (valor1>valor2) sempre retornará um valor lógico, ou seja, verdadeiro ou falso. Caso verdadeiro, o bloco de comandos abaixo da condição será executado, caso contrário, o bloco *else* será executado. Você não é obrigado a implementar o bloco *else* caso seu programa não necessite.

ARGUMENTOS VÁLIDOS PARA INSTRUÇÕES IF

As instruções *if* só podem avaliar valores booleanos. Qualquer expressão que tenha como resultado um valor booleano será adequada. Podemos fazer uso de operadores lógicos "&&" e "||". Quando usamos o operador **&&**, caso a primeira condição avaliada em um comando *if* seja falsa, automaticamente a segunda expressão não será avaliada. Quando usamos **||**, caso a primeira condição avaliada em um comando *if* seja verdadeira, automaticamente a segunda expressão não será avaliada.



```

14 public static void main(String args[]) {
15
16     int valor1 = 10;
17     int valor2 = 15;
18
19     if (valor1 > 10 && valor2 >10) {
20         System.out.println("1) os dois valores sao maiores que 10");
21     }else{
22         System.out.println("1) O primeiro ou o segundo valor nao é maior que 10");
23     }
24     if (valor1 > 10 || valor2 >10){
25         System.out.println("2) Um dos numeros é maior que 10");
26     }else{
27         System.out.println("2) Nenhum dos numeros é maior que 10");
28     }
29 }
```

The screenshot shows a Java code editor with line numbers 14 to 29. Lines 14-28 contain the code above. Line 29 shows the closing brace for the else block. Below the code editor is a terminal window titled 'Saída - comandoIfElse (run)' showing the output of the program. The output consists of two lines of text: '1) O primeiro ou o segundo valor nao é maior que 10' and '2) Um dos numeros é maior que 10'. At the bottom of the terminal window, it says 'CONSTRUIDO COM SUCESSO (tempo total: 0 segundos)'.

Figura 16 – Programa com instrução if e else e operadores lógicos

Note no nosso código que só foram impressas as linhas que retornaram uma condição positiva. Já que os testes *if* requerem expressões booleanas, você precisa ter um conhecimento sólido tanto sobre os operadores lógicos quanto sobre a sintaxe e semântica do teste *if*.

Instruções Switch

Uma estrutura muito utilizada em programação é o *switch*. A instrução *switch* verifica uma variável e trabalha de acordo com seus *cases*. Os *cases* são as possibilidades de resultados que são obtidos por *switch*.

Basicamente, o *switch* serve para controlar várias ações diferentes de acordo com o *case* definido dentro dele. Segue abaixo a estrutura do *Switch*:

```
switch (expressão ordinal) {  
    case valor ordinal 1:  
        diretiva 1;  
        break;  
    case valor ordinal 2:  
        diretiva 2;  
        break;  
        ...  
    default:  
        diretiva N;  
}
```

A expressão ordinal é a expressão que retorna um valor de algum tipo discreto (inteiro, *char* etc.). Veja, a seguir, o fluxograma da instrução *switch*.

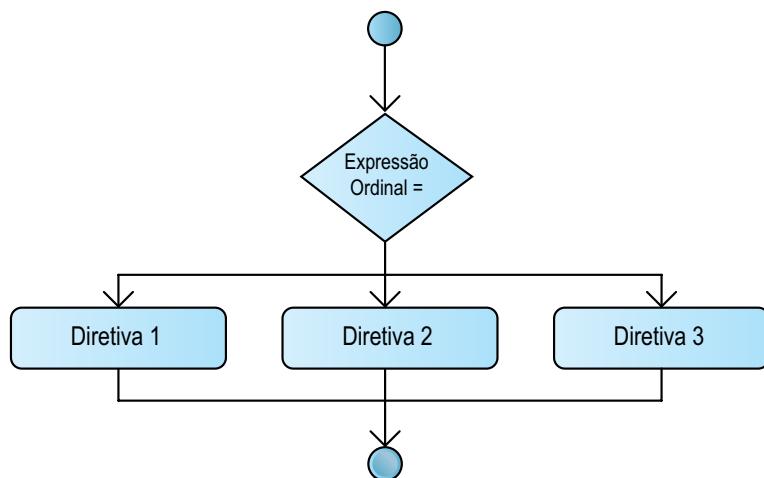
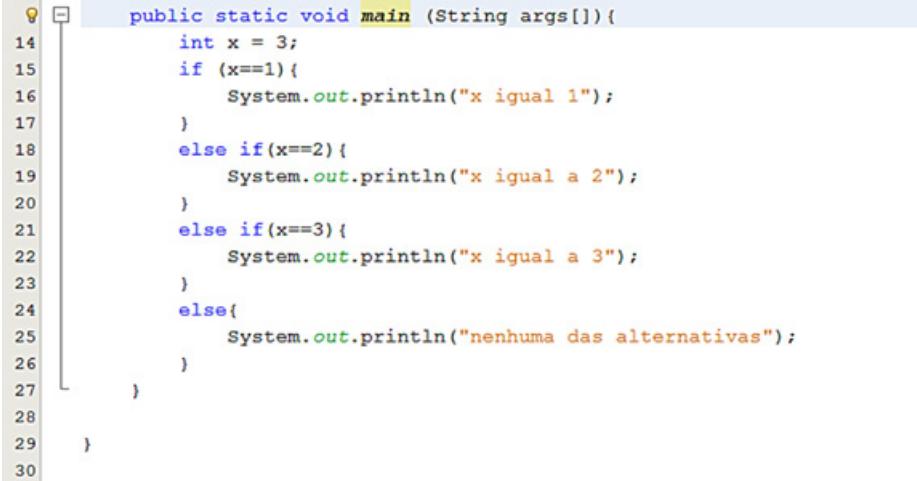


Figura 17 – Programa com instrução switch

Fonte: o autor

É possível construir um programa com *if* equivalente ao *switch*. Observe o código do bloco *else-if* da figura abaixo com *if* aninhados:



```
public static void main (String args[]){
    int x = 3;
    if (x==1){
        System.out.println("x igual 1");
    }
    else if(x==2){
        System.out.println("x igual a 2");
    }
    else if(x==3){
        System.out.println("x igual a 3");
    }
    else{
        System.out.println("nenhuma das alternativas");
    }
}
```

Figura 18 – Instruções if aninhados

Talvez você esteja se perguntando quando devo usar o *switch* e quando devo utilizar o comando *if*. Qual dos dois comandos será mais útil? O código da figura anterior é um bom motivo para se utilizar *switch*, pois o uso de cada um dos comandos depende da composição da estrutura de decisão. Então devemos escolher o *switch* quando, no teste realizado, usarmos uma mesma variável, igualando-a com vários valores diferentes. A estrutura do comando *switch* vem acompanhada com o comando *case*, que com base no valor da variável do comando *switch* define qual opção será executada. Para que somente um entre os vários comandos seja executado, devemos executar o comando *break* logo após a execução dos comandos contidos no bloco do comando *case* selecionado. Veja um exemplo na figura abaixo:

```
13  public class Menu {  
14      public static void main(String[] args) throws IOException {  
15          System.out.println("Digite um dos comandos abaixo do menu: ");  
16          System.out.println("1: Comprar; 2: Vender; 3: Listar; 4: Sair");  
17          // le do teclado um caracter  
18          Scanner scan = new Scanner(System.in);  
19          int opcao = scan.nextInt();  
20          switch (opcao) {  
21              case 1:  
22                  System.out.println("Você acessou COMPRAR");  
23                  break;  
24              case 2:  
25                  System.out.println("Você acessou VENDER");  
26                  break;  
27              case 3:  
28                  System.out.println("Você acessou LISTAR");  
29                  break;  
30              default:  
31                  System.out.println("Saindo do sistema");  
32          }  
33      }  
34  }
```

Figura 19 – Instrução switch

Nesse programa vamos ler do teclado um caractere numérico utilizando a classe *Scanner* do Java. A classe *Scanner* é uma das diversas classes do Java que permite a leitura de dados vindos do teclado. A utilização do método *nextInt()* da classe *Scanner* deixa no *buffer* a instrução da tecla pressionada.

Agora que entendemos um pouco sobre essa poderosa classe *Scanner* do Java, vamos nos ater ao que interessa, que é a instrução de controle *switch*.

Nesse programa leremos do teclado um caractere numérico, e caso seja igual a uma das opções do nosso menu, ele então apresentará a informação personalizada para cada opção escolhida. Somente executamos o comando *break* ao final dos comandos executados. É importante observar que a palavra-chave *break* interrompe a execução do *case*. Se a palavra-chave *break* for omitida, o programa simplesmente continuará a executar os outros blocos *case* até que uma palavra *break* seja encontrada na instrução *switch*, ou até que a instrução termine. Veja no exemplo abaixo como ficaria o nosso código anterior comentando todos os comandos *break*.

```

18     Scanner scan = new Scanner(System.in);
19     int opcao = scan.nextInt();
20     switch (opcao) {
21         case 1:
22             System.out.println("Você acessou COMPRAR");
23             // break;
24         case 2:
25             System.out.println("Você acessou VENDER");
26             //break;
27         case 3:
28             System.out.println("Você acessou LISTAR");
29             // break;
30     default:
31         System.out.println("Saindo do sistema");
32     }

```

... Saída - ComandoSwitch (run)

```

run:
Digite um dos comandos abaixo do menu:
1: Comprar; 2: Vender; 3: Listar; 4: Sair
1
Vocé acessou COMPRAR
Vocé acessou VENDER
Vocé acessou LISTAR
Saindo do sistema
CONSTRUÍDO COM SUCESSO (tempo total: 3 segundos)

```

Figura 20 – Instrução switch

Observe o resultado na saída de comando. Essa combinação aconteceu porque o código não chegou a uma instrução *break*; portanto, a execução simplesmente continuou a percorrer cada instrução *case* até o final. Esse processo é chamado de “passagem completa” por causa da maneira como a execução passa de uma instrução para a seguinte.

Há também o comando *default*, que representa uma exceção a todas as opções listadas nos comandos *case*. É importante observar que nos comandos *case* da instrução *switch* só são aceitas variáveis do tipo *int*, *char*, *byte*, *short*.

ESTRUTURAS DE REPETIÇÃO

Os *loops* da linguagem Java permitem que você repita a execução de um bloco de código até que uma determinada condição seja verdadeira ou durante uma quantidade específica de iterações.

USANDO LOOP FOR

O *loop for* será particularmente útil para o controle do fluxo quando você já souber quantas vezes terá de executar as instruções do bloco do *loop*. A figura abaixo apresenta a estrutura do comando *for*.

```
for ([início]; [condição]; [inc/dec]) {
    diretiva;
}
```

Note que *início* é a diretiva executada antes do laço começar; em seguida, temos a *condição* que é a expressão de condição de parada do laço (geralmente, comparação com o valor final); o *inc/dec* representa diretiva executada

antes de começar cada iteração do laço (geralmente usada para incrementar ou decrementar o contador). A seguir fluxograma de atividades do comando *for*.

O *loop for* é uma construção geral para suportar a interação que é encontrada por uma variável de contador ou semelhante, que é atualizada depois de cada iteração. Veja abaixo um exemplo de um *loop for*.

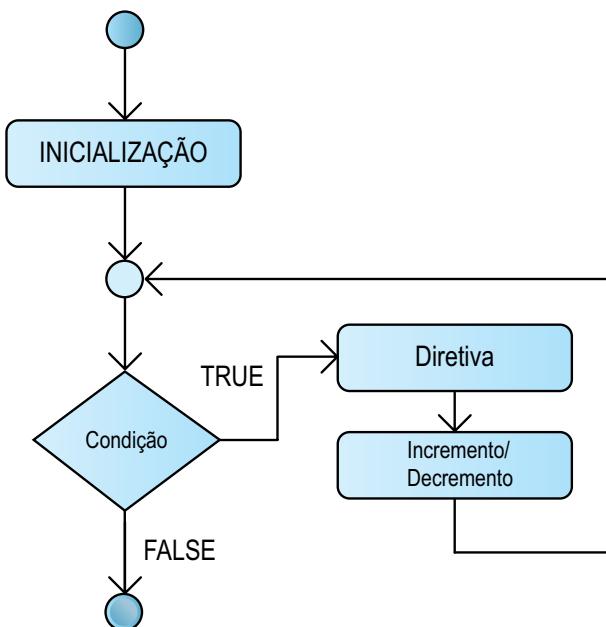


Figura 21 – Instrução for
Fonte: o autor

```

public class ComandoFor {
    public static void main (String args[]){
        for(int i =1; i<=5; i++)
            System.out.println(i);
    }
}

```

Saída - comandoFor (run) ➔

```

run:
1
2
3
4
5
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)

```

Figura 22 – Exemplo de programa com instrução for

Lembrando que devemos usar o comando *for* quando sabemos de antemão quantas vezes o *loop* deverá ser executado.

Em várias linguagens de programação, o *for* (ou similar) serve somente para repetição simples:

```

para i de 1 até 10 faça
    Escreva i
fim_para

```

Em Java, pode-se usar para fazer repetição condicional:

```
15     boolean achou = false;
16     for (int i = 0; (!achou); i++) {
17         /* ... */
18     }
```

Figura 23 – Declaração da instrução for

Podemos efetuar múltiplas diretivas na inicialização e no incremento, se necessário, separando com vírgulas:

```
21     for (int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
22         System.out.println("i= " + i + " j= " + j);
23     }
```

Figura 24 – Declaração da instrução for

USANDO LOOP WHILE

O *loop while* será adequado em cenários nos quais você não souber quantas vezes o bloco ou instrução terá que ser repetido. A instrução *while* tem o formato a seguir:

```
while (condição)
{
    Diretiva;
}
```

A condição é a expressão de condição de parada do laço (expressão lógica). A seguir, o diagrama da instrução *while*.

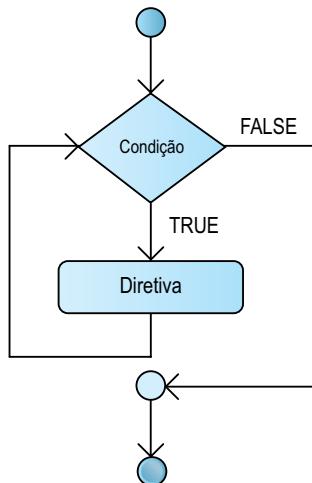


Figura 25 – Fluxograma da instrução while

Fonte: o autor

O problema com estruturas de repetição, principalmente com *while*, é o que chamamos de *looping infinito*. Damos esse nome ao fato de que o programa fica repetindo a mesma sequência de códigos esperando por um resultado que nunca irá acontecer. Portanto, é imprescindível que uma determinada variável seja modificada de acordo com cada *loop*.

Na figura abaixo temos um programa utilizando a instrução *while*.

O código Java exibe a seguinte saída:

```

19     int contador = 0;
20     while (contador < 5) {
21         System.out.println("Repetição Nro: " + contador);
22         contador++;
23     }
24 }
25
26
  
```

Saída - comandoFor (run) #

```

run:
Repetição Nro: 0
Repetição Nro: 1
Repetição Nro: 2
Repetição Nro: 3
Repetição Nro: 4
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
  
```

Figura 26 – Exemplo de programa com instrução while

Note que a variável contador é iniciada com 0, a cada *loop* executado é somado 1 ao contador. Perceba que o *while* irá manter a repetição enquanto a variável contador for menor que 5.

Outro ponto importante é que a variável contador é inicializada antes de chegar ao *while*, assim, o *while* comparará a sentença e só depois permitirá a execução do bloco. Se quisermos fazer todo o bloco primeiro e só depois fazer a comparação, devemos utilizar o comando *DO WHILE*.

Usando *loop do-while*

O *do while* difere do *while* no sentido de permitir que pelo menos uma execução do bloco de comandos seja executada antes de testar a condição. O bloco de comandos será executado enquanto a condição for verdadeira. A figura abaixo mostra um exemplo da sintaxe do comando *do while* e a representação gráfica do comando.

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

Abaixo o fluxograma da instrução *do while*.

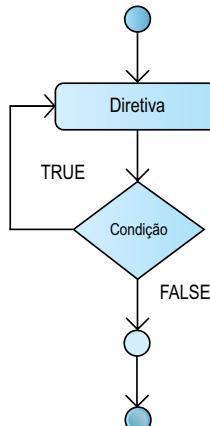


Figura 27 – Exemplo de programa com instrução do-while

Fonte: o autor

Note que *do while* só avalia depois, certamente executando a diretiva ao menos uma vez; neste caso, devemos ter as mesmas precauções quanto ao *while*, no que diz respeito ao *looping* infinito. Não é necessário comparar a variável de controle antes do bloco de código como acontece com *while*, pois a comparação só será feita após todo o código ter sido executado. Veja no exemplo abaixo:

```

30         // comando repeticao DO-WHILE
31         int valor = 1;
32         do{
33             System.out.println("O numero é: " + valor);
34             valor++;
35         }while (valor < 5);
36     }
37 }
38 }
39

```

ComandoFor > main > while (contador < 5) >

Saída - comandoRepeticao (run) :

```

run:
Digite f para terminar:
O numero é: 1
O numero é: 2
O numero é: 3
O numero é: 4
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)

```

Figura 28 – Exemplo de programa com instrução do-while

DESVIOS INCONDICIONAIS

O Java só tem dois casos específicos de desvios incondicionais: *break* e *continue*. *Break* e *continue* são dois comandos de controle de estruturas largamente utilizados em *loops* (repetições), como *for* e *while*.

Break

O comando *break* tem a função de interromper a execução de um *loop*. No comando *switch*, por exemplo, ele determina que não pode ser executado o *case* seguinte, e assim por diante.

A seguir, temos uma repetição que se inicia em 1 e deve terminar em mil (500), mas dentro desta estrutura há uma condição: se a variável for igual a 6, saia da estrutura de repetição. Vejamos:

```
11 public static void main (String args[]){
12     for (int cont=1; cont<=500; cont++){
13         System.out.println("nr: "+cont);
14         if (cont==6)
15             break;
16     }
17 }
18
19
```

ComandoContinue >

Saída - comandoContinue (run) #

```
run:
nr: 1
nr: 2
nr: 3
nr: 4
nr: 5
nr: 6
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 29 – Exemplo de programa com comando break

Note que mesmo que a condição *for* instruísse para que o programa imprimisse números até o 500, a condição *if* se tornou válida quando o *cont* chegou ao número 6, e sendo verdadeiro, executou seu bloco de instrução interrompendo o programa.

Continue

O comando *continue* tem a função de fazer com que a condição do comando de *loop* seja novamente testada, mesmo antes de alcançar o fim do comando. Veja, a seguir, um exemplo do comando *continue*.

The screenshot shows a Java development environment. In the code editor, lines 12 to 22 of a main method are visible:

```

12     public static void main(String args[]) {
13
14         for (int i = 0; i < 12; i++) {
15             if ((i > 4) && (i < 8)) {
16                 continue;
17             }
18             //apresenta na tela quando o i nao estiver entre 4 e 8
19             System.out.println("i = " + i);
20         }
21
22

```

The terminal window below shows the output of the program, which prints the values of 'i' from 0 to 11, except for 5, 6, and 7, due to the 'continue' statement.

```

Salida - comandoBreakContinue (run)
run:
i = 0
i = 1
i = 2
i = 3
i = 4
i = 6
i = 8
i = 9
i = 10
i = 11
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)

```

Figura 30 – Exemplo de programa com comando continue

Note que o programa não imprimiu os números 5, 6 e 7 devido à execução do comando *continue*.

Break e continue rotulados

Tanto a instrução *break* quanto *continue* podem ser não rotuladas ou rotuladas. As rotuladas serão necessárias somente nas situações em que você tiver um *loop* aninhado e precisar indicar qual quer encerrar ou a partir de qual deseja continuar a próxima interação. Uma instrução *break* sairá do *loop* rotulado e não do *loop* atual se a palavra-chave *break* for combinada com um rótulo. Um exemplo do formato de um rótulo se encontra no código a seguir:

The screenshot shows a Java code editor with the following code:

```
16 public static void main(String[] args) {  
17     int y = 7;  
18     externo:  
19         for (int i = 0; i < 15; i++) {  
20             while (y > 3) {  
21                 y++;  
22                 System.out.println("Dentro do Loop");  
23                 break externo;  
24             }  
25             System.out.println("Fora do loop");  
26         }  
27         System.out.println("Fora do Programa");  
28     }  
29 }  
30 }
```

Below the code is a toolbar with icons for Run, Stop, and Break. The status bar shows "Rotulo > main >". A "Saída - Rotulo (run)" tab is open, displaying the program's output:

```
run:  
Dentro do Loop  
Fora do Programa  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 31 – Exemplo de programa com comando break rotulado

Nesse exemplo, a sentença “Dentro do Loop” será exibida uma vez. Em seguida, a instrução *break* rotulada será executada e o fluxo sairá do *loop* rotulado com o rótulo que declaramos “externo”. Então, a próxima linha de código exibirá “Fora do Programa”. Vejamos o que acontecerá se a instrução *continue* for usada em vez de *break*:

The screenshot shows a Java code editor with the following code:

```
16 public static void main(String[] args) {  
17     int y = 1;  
18     externo:  
19         for (int i = 0; i < 15; i++) {  
20             while (y < 3) {  
21                 y++;  
22                 System.out.println("Dentro do Loop");  
23                 continue externo;  
24             }  
25             System.out.println("Fora do loop");  
26         }  
27         System.out.println("Fora do Programa");  
28     }  
29 }  
30 }
```

Figura 32 – Exemplo de programa com comando continue rotulado

```
run:
Dentro do Loop
Dentro do Loop
Fora do Programa
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

Figura 33 – Resultado do programa com comando continue rotulado

Nesse exemplo, a sentença “Dentro do Loop” foi executada 2 vezes atendendo a condição de $y < 3$. Depois de concluída essa instrução, o programa imprimiu 13 vezes a sentença “Fora do Loop”. Para concluir, quando a condição do *loop* externo for avaliada como falsa, o *loop* i será encerrado e a sentença “Fora do Programa” será exibida.

Break e continue não rotulados

As instruções *break* e *continue* não rotuladas saíram da estrutura do *loop* atual e prosseguirão na linha de código posterior ao bloco do *loop*. O exemplo abaixo demonstra uma instrução *break*.

The screenshot shows a Java code editor with the following code:

```
16 public static void main(String[] args) {  
17  
18     boolean pausa = true;  
19     while (true){  
20         if (pausa){  
21             System.out.println("Programa Pausado");  
22             break;  
23         }  
24     }  
25 }  
26 }  
27 }
```

The line `break;` at line 22 is highlighted in yellow. Below the code editor is a run configuration window titled "NaoRotulado (run)" with the following output:

```
run:  
Programa Pausado  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 34 – Resultado do programa com comando break não rotulado

No exemplo anterior, a instrução *break* não foi rotulada. A seguir, temos o exemplo com a instrução *continue* não rotulada.

The screenshot shows a Java code editor with the following code:

```
16 public static void main(String[] args) {  
17  
18  
19  
20     for (int cont = 1; cont <= 8; cont++) {  
21         if (cont == 3) {  
22             continue;  
23         }  
24         System.out.println("nr: " + cont);  
25     }  
26 }  
27 }
```

The line `continue;` at line 22 is highlighted in yellow. Below the code editor is a run configuration window titled "NaoRotulado (run)" with the following output:

```
run:  
nr: 1  
nr: 2  
nr: 4  
nr: 5  
nr: 6  
nr: 7  
nr: 8  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 35 – Resultado do programa com comando continue não rotulado

Observe que após executar o código anterior não foi impresso o número 3, pois ao chegar na condição, o programa encontrou um *continue* que mandou continuar a execução do programa.

CONSIDERAÇÕES FINAIS

Nesta Unidade abordamos aspectos fundamentais da linguagem Java. Como vimos, há vários tipos primitivos em Java, cada um com um consumo de memória determinado que afeta diretamente o seu alcance. Vimos também que uma variável referenciará sempre a um tipo primitivo de Java e que o escopo é a vida de uma variável em Java, tratando-se dos locais nos quais ela pode ser acessada. Em Java, o escopo de variáveis vai de acordo com o bloco onde elas foram declaradas. A variável é criada no primeiro acesso a ela e é destruída após o interpretador sair do bloco de execução ao qual ela pertence.

Também abordamos uma importante área envolvendo maneiras de controlar o fluxo de seu programa, com base em um teste condicional. Primeiro você conheceu as instruções *if* e *switch*. A instrução *if* avalia uma ou mais expressões com relação a um resultado booleano. A instrução *switch* é usada para substituir várias instruções *if-else*.

Você também conheceu três estruturas de *loop* disponíveis na linguagem Java. Essas estruturas são os *loops for, while* e *do-while*. De um modo geral, o *loop for* é usado quando sabemos quantas vezes será preciso percorrer o *loop*. O *loop while* é usado quando não sabemos quantas vezes precisamos percorrer o *loop*, enquanto o *do-while* é empregado quando temos que percorrer o *loop* pelo menos uma vez.

ATIVIDADES



1. Suponha que uma variável é declarada como int a = 2147483647. Qual será o resultado das impressões a seguir?
 - a) System.out.println(a);
 - b) System.out.println(a + 1);
 - c) System.out.println(2 - a);
 - d) System.out.println(-2 - a);
 - e) System.out.println(2 * a);
 - f) System.out.println(4 * a);
2. O que está errado com cada uma das seguintes afirmações?
 - a) if (a > b) then c = 0;
 - b) if a > b { c = 0; }
 - c) if (a > b) c = 0;
 - d) if (a > b) c = 0 else b = 0;
3. Escreva um programa para cada situação de instruções de controle abaixo:
 - a) If-else.
 - b) Switch.
 - c) For loop.
 - d) While.

MATERIAL COMPLEMENTAR



NA WEB

Tipos primitivos do Java

Por ActJava

Quando criamos um programa, devemos saber com que tipo de informação estamos lidando e para qual finalidade ela será útil. Afinal, devemos ter conhecimento disso para declararmos corretamente o tipo de dado que a variável irá suportar.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://www.actjava.com.br/2012/04/tipos-primitivos.html>>.



NA WEB

Certificação Java – Tipos Primitivos

Disponível em: <<http://www.youtube.com/watch?v=PN0avdHxRZ8>>.



NA WEB

Fundamentos da UML

Por KDE.org

A Unified Modelling Language (UML) é uma linguagem ou notação de diagramas para especificar, visualizar e documentar modelos de software orientados por objetos. A UML não é um método de desenvolvimento, o que significa que não lhe diz o que fazer primeiro ou o que fazer depois ou como desenhar o seu sistema, mas ajuda-o a visualizar o seu desenho e a comunicar com os outros. A UML é controlada pela Object Management Group (OMG) e é a norma da indústria para descrever graficamente o software.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <http://docs.kde.org/stable/pt_BR/kdesdk/umbrello/uml-basics.html>. Acesso em 14 maio 2013.



NA WEB

Introdução a UML

Disponível em: <<http://www.youtube.com/watch?v=hfN6n5fJfLc>>.

REFLITA



A eficiência de Java depende totalmente do tipo de hardware utilizado!

CLASSES X OBJETOS JAVA

UNIDADE



Objetivos de Aprendizagem

- Entender o que é uma classe Java.
- Entender o que é um objeto Java.
- Estudar o que é atributo, método, construtores, e o método main.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- Conceitos Básicos de Classes
- Conceitos Básicos de Objetos
- JavaBeans ou POJOs
- Estado e Comportamento de Objetos

INTRODUÇÃO

Conforme demonstrado em aula extra (vídeo de explicação da instalação da IDE Netbeans), vamos utilizar agora a IDE Netbeans para a criação de uma nova classe no Java. A IDE é um ambiente de desenvolvimento integrado que possibilita o desenvolvimento de aplicações Java de forma mais rápida e mais simples. O Netbeans facilita o processo de criação de projetos, compilação, depuração e escrita de código através de um editor de textos eficiente e inúmeras ferramentas de processamento e controle.

Conforme visto nas Unidades anteriores, a Classe será o modelo utilizado para criarmos o Objeto, e desta maneira resolver o problema. Mas como criar uma Classe em Java? E um Objeto? Será possível pegar e tocar neste Objeto que iremos criar? Vamos aprender como fazer estas e mais coisas neste momento.

Para os estudos a seguir, iremos criar um projeto na nossa IDE NetBeans chamado Agenda. Para isto, basta:

Clicar em “Arquivo” -> “Novo Projeto”

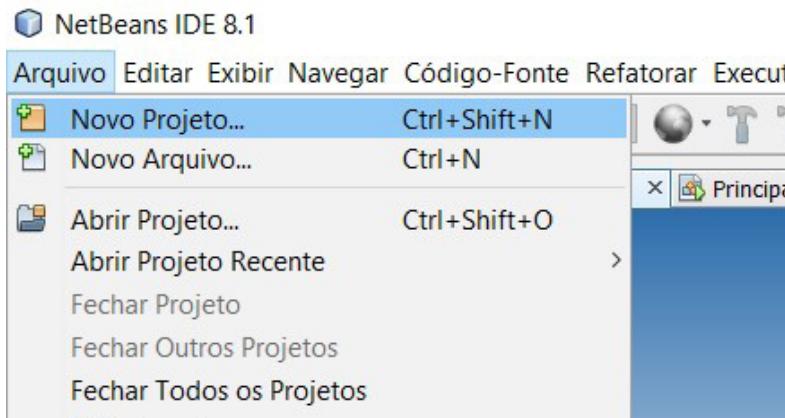


Figura 36 – Menu para se criar um Novo Projeto

Neste momento há uma infinidade de possibilidades para escolher que tipo será o seu projeto. Neste momento, iremos trabalhar apenas com aplicações para desktop. Logo usaremos o Java. Ao clicar em Java, algumas opções irão aparecer na parte direita da tela. Basta escolher, criar uma Aplicação Java e clicar em Próximo.

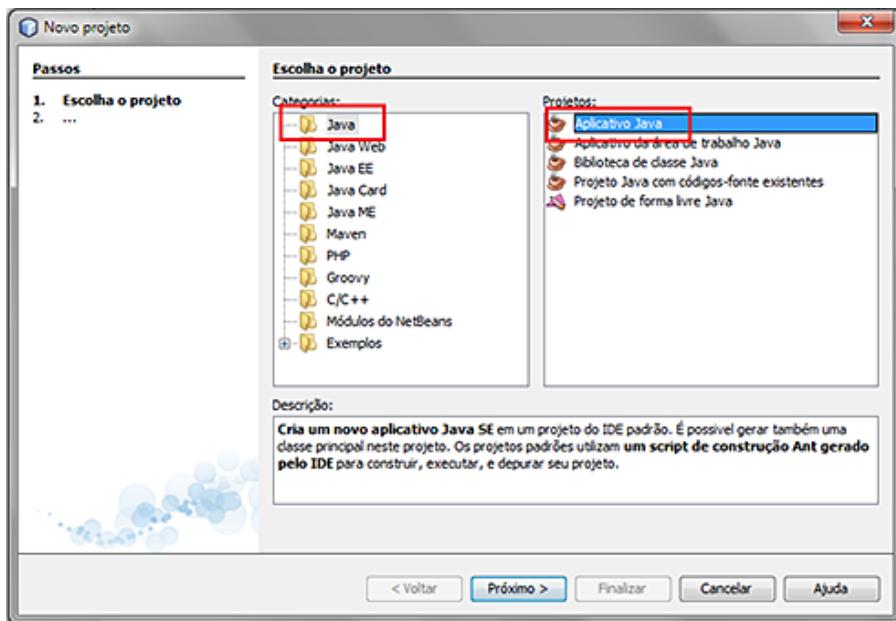


Figura 37 – Primeiro passo para se criar um projeto Java

Em seguida, o NetBeans nos dá a oportunidade de dar um nome para o projeto. Coloque o nome de Agenda no primeiro campo. Logo abaixo, ele mostra onde ficarão os arquivos dentro do sistema de pastas de seu sistema operacional. Grave bem o local, ou coloque um local de sua escolha, para depois você ter acesso ao programa pronto. Não se esqueça de desmarcar a opção de Criar Classe Principal, para que possamos nós mesmos criar a nossa classe.

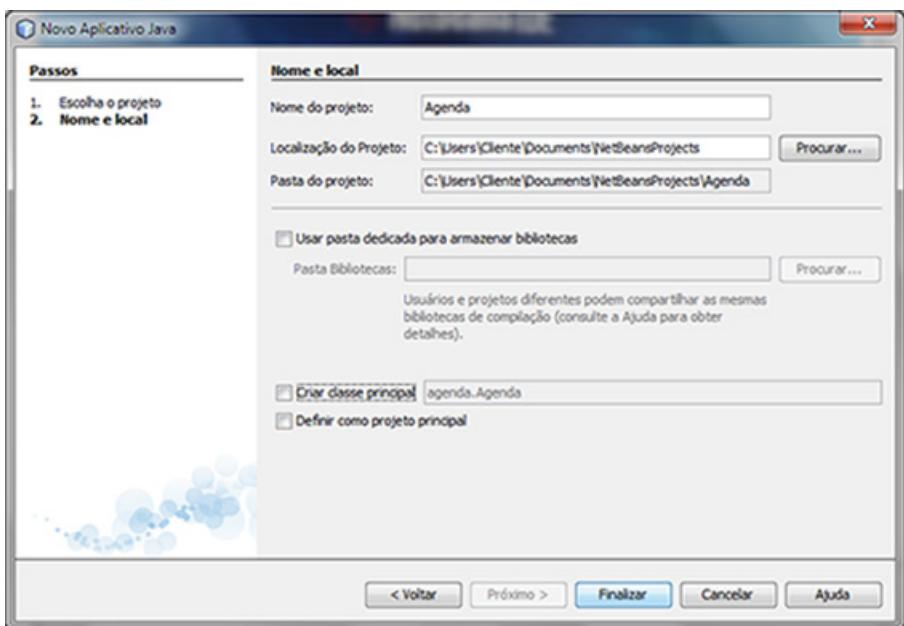


Figura 38 – Segundo passo para se criar um projeto Java

Ao se criar um projeto novo, é possível verificar 2 itens dentro do projeto. O “Pacotes de código-fonte”, que será o local onde guardaremos nossas classes, e Bibliotecas, que é o local onde guardaremos classes já existentes em forma de biblioteca. Por hora, vamos necessitar apenas da JDK do Java.

Para nos auxiliar na organização de nossas classes, podemos utilizar um organizador que chamamos de Pacote.

PACOTES

Os pacotes são agrupamentos de classes, que nos auxiliam a separá-las de acordo com uma determinada característica das classes. Esta organização se faz necessária, pois por meio dela fica mais fácil para que outras pessoas que olhem o seu código entendam o seu programa, além de auxiliar no trabalho em equipe e manutenção do seu software. Dificilmente um programador que faça um software mais complexo que uma calculadora não utilize pacotes.

Apesar de aparecer um “<pacote padrão>” dentro do projeto, este não é um pacote. Para se criar um pacote, basta clicar com o botão direito sobre a área onde irão ficar as nossas classes, clicar em “Novo” -> “Pacote Java...”.

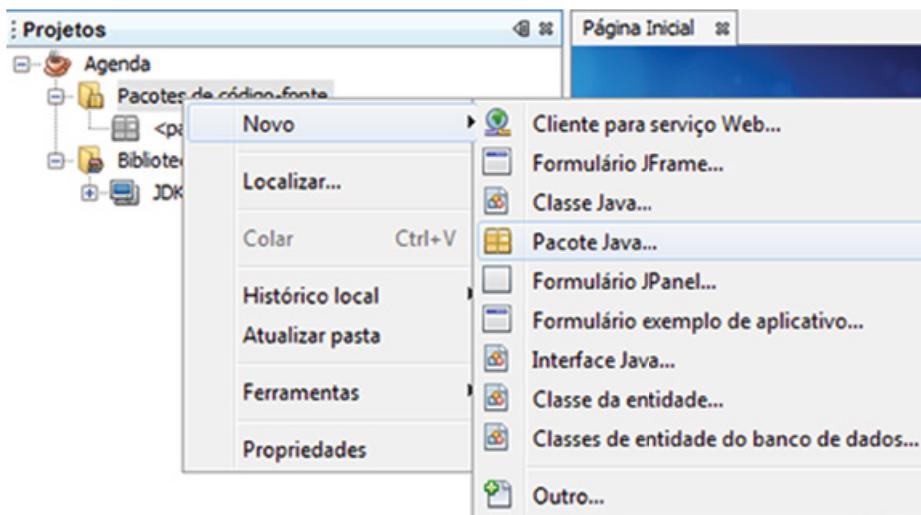


Figura 39 – Menu para se criar um Pacote novo

Caso a opção “Pacote Java...” não esteja disponível, selecione a opção “Outro...”. Esta opção irá abrir uma janela, onde você deverá procurar pela opção “Java” na janela da esquerda, e “Pacote Java” na janela da direita.

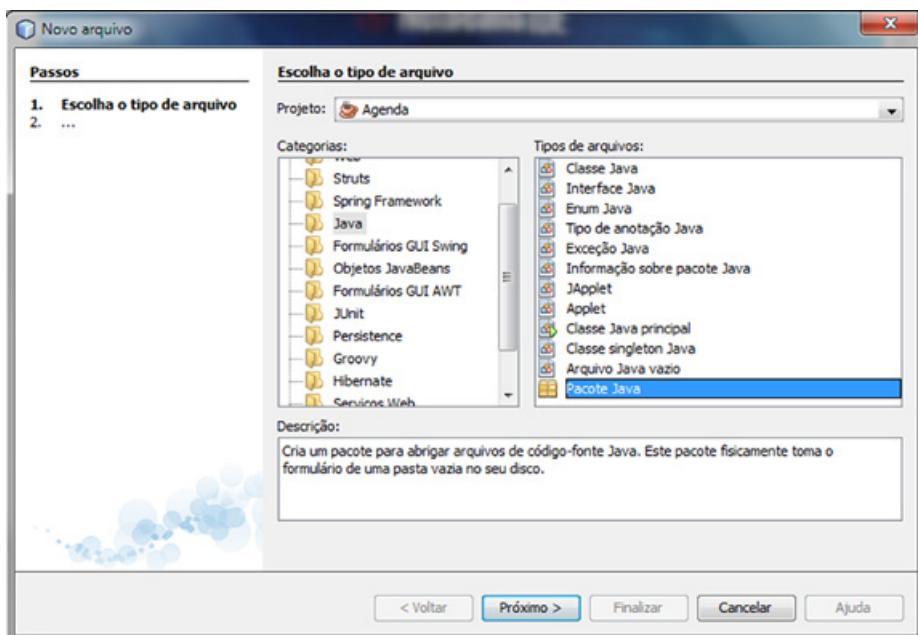


Figura 40 – Forma alternativa para se criar um Pacote novo

Ao se clicar em “Pacote Java...” uma nova janela irá se abrir, onde deverá ser colocado o nome do pacote. Vamos atribuir o nome de “Dados”.

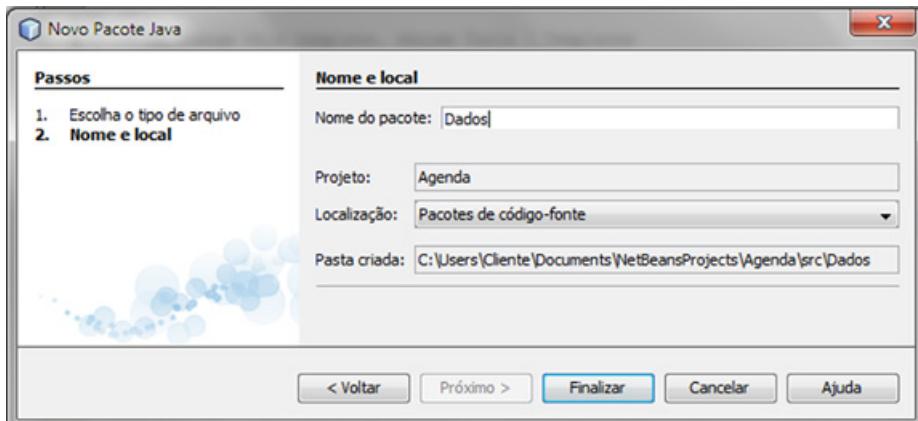


Figura 41 – Tela onde o nome do pacote será atribuído

Dentro deste pacote, vamos criar uma classe nova, chamada Pessoa. Para isto, clique com o botão direito sobre o pacote Dados, selecione “Novo” -> “Classe Java...”.

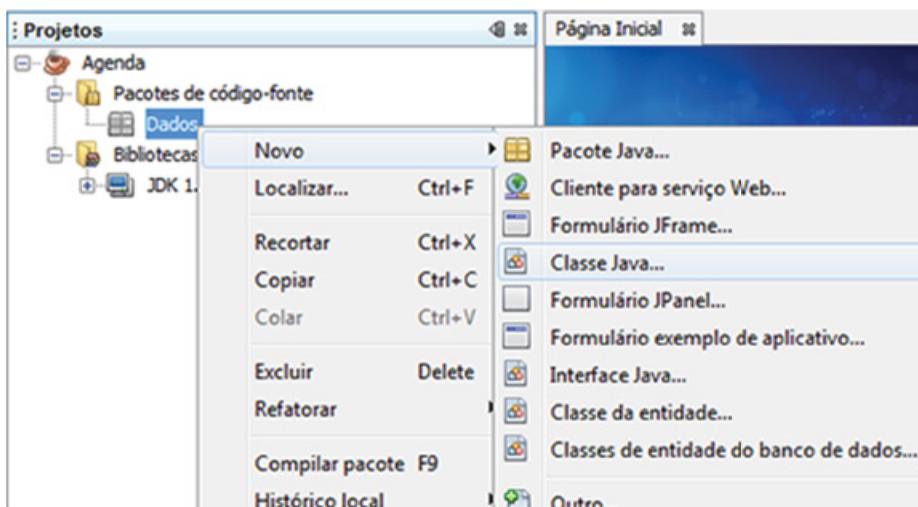


Figura 42 – Menu para se criar uma nova Classe Java

Em seguida, basta colocar o nome Pessoa e a classe será criada. Ao abrir o arquivo Pessoa.java, podemos verificar que ele se encontra dentro do pacote Dados pela primeira linha de comando, que informa em qual pacote ele está. Também o NetBeans oferece um padrão ao se criar as classes. Algumas informações adicionais podem ser vistas, mas estão comentadas, ou seja, não farão diferença na nossa programação.

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package Dados;
6
7  public class Pessoa {
8
9  }
10 }
11 }
```

Figura 43 – Código gerado pelo NetBeans ao se criar uma classe com o nome de Pessoa

CONCEITOS BÁSICOS DE CLASSES

Com a classe Pessoa aberta, pode-se verificar alguns detalhes. Primeiro que o nome da classe se encontra no fim do comando “public class Pessoa”. Este *public class* indica que iremos criar uma classe que poderá ser utilizada por quem quiser usá-la. Depois deste comando, dá-se o nome da classe, que não é por acaso o mesmo nome do arquivo, para depois abrir e fechar os colchetes. Tudo que se encontrará dentro destes colchetes fará parte da classe Pessoa, por isso tome cuidado para quando for implementar seus atributos e métodos, crie eles dentro destas chaves.

Mas o que deveremos colocar dentro desta classe? Acho que chegou a hora de verificarmos qual é o problema que deverá ser resolvido com o nosso programa. Nossa usuário gostaria de ter uma agenda eletrônica, onde ele possa gerenciar o nome, idade e telefone de uma única pessoa. Opa, então o que fizemos até aqui foi o certo. Criamos uma classe Pessoa, que irá conter os dados que queremos guardar. Estes dados vão acabar virando Atributos de nossa classe.

Atributos

Os atributos de uma classe são, basicamente, variáveis que se encontram dentro da classe, e que a ela pertencem. Como no exemplo a seguir, podemos ver 3 variáveis criadas dentro da classe Pessoa. O tipo de variável de cada atributo deverá ser analisado e estipulado antes de se começar a programar. Para nosso problema, precisamos de uma variável para guardar o nome, uma para a idade e outra para o telefone. Ficou definido que, para o nome e para o telefone, será utilizado o tipo *String*, enquanto que para a idade, será usado o tipo *int*.

```
11  public class Pessoa {  
12      String nome;  
13      String telefone;  
14      int idade;  
15  }  
16 }
```

Figura 44 – Classe Pessoa depois de criar 3 atributos

Agora, quando criarmos um objeto do tipo Pessoa, estas variáveis serão criadas em forma de atributos, reservando um espaço na memória para elas. Logo, se criar 10 objetos do tipo pessoa, serão reservados 20 espaços para se guardar *String* e 10 espaços para se guardar *int*. Apesar disso, cada objeto terá as suas variáveis e não influenciarão uns nos outros.

Mas ao criar-se um objeto do tipo Pessoa, as variáveis terão algum valor padrão? É possível mudar este valor padrão? Podemos definir o valor padrão por meio dos Construtores das classes.

Construtores

Este artifício que podemos criar dentro de uma classe nos auxilia na hora definir valores padrão para os atributos que serão criados ao se montar um objeto. Os construtores são métodos, ou seja, poderão conter códigos de programação dos mais diversos. A principal diferença entre um construtor e um método qualquer, é que o construtor não pode conter nenhum tipo de retorno, deverá ter exatamente o mesmo nome da classe e ele sempre será chamado primeiro quando um objeto for construído.

Neste momento, vamos criar um construtor que apenas irá atribuir valores padrão para os atributos criados anteriormente. Primeiramente precisa-se criar o cabeçalho do construtor, que contém se ele é acessível de fora da classe, o nome dele e se ele precisa de alguma variável para começar a trabalhar. Para a classe Pessoa, precisaremos criar um construtor da seguinte forma:

```
public Pessoa() { }
```

O *public* indica que ele será acessível de fora da classe. Conforme citado anteriormente, o nome do construtor é exatamente o nome da classe (cuidado nesta parte, pois o java considera diferente as letras maiúsculas e minúsculas). Por fim, temos que abrir e fechar os parênteses. Se houver necessidade de alguma variável para o construtor funcionar, basta colocar entre estes parênteses. Como neste caso não há necessidade de nenhuma variável a mais, basta colocar os parênteses sem nada. Eles são obrigatórios e esquecê-los faz com que o java se engane, e

pense que você está tentando criar uma variável comum, e não um método. Em seguida ao cabeçalho, você deve abrir e fechar as chaves, assim como foi feito na hora de criar a classe. Todos os códigos referentes ao construtor devem estar dentro destas chaves.

Como neste construtor iremos apenas iniciar os atributos com um valor, teremos o seguinte código:

```

16  public Pessoa() {
17      idade = 0;
18      nome = "";
19      telefone = "";
20  }

```

Figura 45 – Construtor Pessoa(), inicializando os atributos vazios

É possível criar diversos construtores para uma mesma classe. Como todos têm que ter o mesmo nome da classe, o que irá diferenciar um construtor do outro são as variáveis criadas nos parênteses. Para o nosso exemplo, vamos criar um segundo construtor, porém os valores que irão para os atributos serão passados por parâmetro. A passagem de parâmetro se dá quando recebemos estas variáveis de fora, ou seja, pelos parênteses. A seguir podemos ver como ficará o segundo construtor.

```

22  public Pessoa(String nom, String tel, int id)
23  {
24      nome = nom;
25      telefone = tel;
26      idade = id;
27  }

```

Figura 46 – Novo construtor, recebendo 3 parâmetros, um para cada atributo

Podemos reparar que a principal diferença entre o primeiro e o segundo construtor se dá nos parênteses. Neste segundo pode-se ver 3 variáveis sendo criadas como parâmetros. Foi tomado o cuidado de não se criar variáveis com os mesmos nomes dos atributos, mas veremos mais adiante o que fazer quando isso ocorrer. Porém, os tipos das variáveis são os mesmos. Também é muito importante notar como foi separada uma variável da outra. Com o uso da vírgula. Como neste caso temos estas variáveis, em vez de atribuir valores fixos para os atributos, as variáveis foram atribuídas.

Vale lembrar que todos os códigos, desde a criação dos atributos e dos construtores, devem ficar dentro das chaves da classe.

O construtor é um método especial. Mas o que é um método? E se eu quiser criar um método normal? Teremos esta resposta na próxima seção.

MÉTODOS

Enquanto os atributos definem as características que um objeto terá quando criado, os métodos irão indicar quais funcionalidades que este objeto terá. Os métodos nada mais são do que pedaços de código que recebem um determinado nome, e que podem ser chamados quando criamos um objeto.

A estrutura de criação de um método é um pouco diferente da de um construtor. Este possui 4 partes. Primeiro, é identificado se ele é acessível para outras classes. Em seguida, temos que indicar se ao fim do método, este terá que retornar algum valor ou não. Em seguida, temos o nome e, por fim, se ele irá receber algum tipo de parâmetro ou não. Supondo que vamos criar um método que recebe uma idade, e este método deverá verificar se é uma idade válida, ou seja, se é maior que 0 anos de idade. Se for, deverá retornar verdadeiro, e caso não for, deverá retornar falso.

Mesmo não sabendo como iremos realizar esta comparação, já podemos pensar em como ficará o cabeçalho deste método. Primeiro, ele deverá ser acessível de outras classes. Depois, deverá retornar verdadeiro ou falso, dependendo da idade. Para este tipo de valor, podemos utilizar uma variável do tipo *boolean*. Depois temos o nome, que iremos colocar como *testeIdade*. Por fim, este método deverá receber a idade que irá testar para poder avaliá-la. O cabeçalho da função ficará desta forma:

```
public boolean testeIdade(int idade) { }
```

Com a variável em mãos, fica fácil montar uma lógica de programação que testa se a idade é maior ou menor que 0. O código para esta função pode ser escrito com um simples *if*:

```

if (idade >=0)
    return true;
else
    return false;

```

O código final fica:

```

29     public boolean testeIdade(int idade)
30     {
31         if (idade >= 0)
32             return true;
33         else
34             return false;
35     }

```

Figura 47 – Método testeIdade, que recebe um parâmetro e retorna true ou false

Mas como podemos ter certeza de que esta variável “idade” no *if* é a variável que está sendo passada por parâmetro, e não o atributo «idade» da classe? Existe uma regra que diz que uma variável criada dentro de um método (ou dentro das chaves, ou dentro dos parênteses) sempre leva vantagem sobre um atributo. Esta característica recebe o nome de sombreamento, onde a variável criada para o método deixa o atributo em sua sombra. Mas e se quisermos acessar o atributo de dentro do método? Veremos mais adiante como realizar esta façanha.

Outro método interessante de se criar é uma mensagem padrão para mostrar os atributos da classe de forma ordenada. Para isto, vamos criar um método chamado mostrarDados, onde este não precisa receber nenhum valor, mas ele retorna uma *String* preparada para ser usada. O código desta função ficará da seguinte forma:

```

37     public String mostrarDados()
38     {
39         String retorno = "Nome: " + nome + " Idade: " + idade + " Telefone: " + telefone;
40         return retorno;
41     }

```

Figura 48 – Método mostrarDados, que retorna uma frase contendo os atributos organizados

Repare que para criar os nomes dos métodos, algumas regras são adotadas para manter um padrão. Primeiro, é importante ressaltar que nomes de métodos não podem conter espaços, caracteres especiais ou iniciar com número. Além disso, adota-se a seguinte regra: o nome do método inicia-se com minúsculo, e a cada palavra nova, não dá espaço nem *underline*, mas deixa a primeira letra maiúscula. Esta última regra é um padrão adotado entre os programadores, com o intuito de uma maior legibilidade dos nomes dos métodos. Fica a seu cargo adotar esta regra ou não.

Nossa classe Pessoa está pronta. Ela contém 3 atributos, 2 construtores e 2 métodos que poderemos utilizar quando criarmos um objeto. Mas e o objeto dela, vamos criar onde? Para poder utilizar esta classe, precisaremos criar uma segunda classe, mas com uma diferença bem crucial. Esta segunda classe não servirá para se criar objetos, mas sim para conter o método *main*.

Normalmente, cada método possui um nome diferente em relação aos outros que estão na mesma classe, que é um fator determinante para diferenciá-lo dos demais. Porém, muitas vezes você terá a necessidade de que um método tenha a possibilidade de trabalhar com diferentes tipos de variáveis. Um exemplo claro é o método *println()*. Apesar de manter o mesmo nome, você pode passar qualquer tipo de variável que ele mostrará na tela. Nós podemos recriar esta característica também realizando a sobrecarga de um método.

Um método não é identificado pelo seu nome, e sim pela sua assinatura. A principal diferença é que na assinatura, além do nome, os parâmetros também ajudam a identificar. Vamos criar um método chamado teste:

```
public void teste() {...}
```

A assinatura deste método é *teste()*. Se quisermos criar outro método com o mesmo nome, devemos mudar a forma como os parâmetros estão chegando, por exemplo:

```
public void teste(String var1){...}
```

Porém, a assinatura de um método não leva em consideração o nome das variáveis, ou seja, não é possível criar outro método chamado teste, que receba uma variável do tipo *String*, mesmo que esta variável tenha outro nome. O que é levado em conta é o número de parâmetros e a ordem deles. Logo, podemos criar outro método:

```
public void teste(String var1, String var2, int var3){...}
```

e

```
public void teste(String var1, int var2, String var3){...}
```

O método *main*

Este método é bem especial, pois serve de ponto de partida para que o seu sistema comece a rodar. Todo programa em java começa de um método *main*, que chama outros métodos e cria objetos de outras classes. Enquanto as outras classes trazem os códigos necessários para resolver nosso problema, o *main* irá efetivamente resolver o problema com o auxílio das classes previamente criadas.

O método *main* possui uma forma exata de ser criado, que é a seguinte:

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

Para que ele seja o início de seu programa, ele deve conter todos estes detalhes. Primeiro, o *public* indica que ele será aberto para outras classes. O *static*, uma novidade para nós até o momento, indica que não precisaremos criar um objeto desta classe para usar o *main*. Desta forma, o próprio java chama este método por você. *Void* indica que o método não retorna nada e *main* é o seu nome. Como parâmetro, ele recebe um vetor de *Strings*, que pode ser passado ou não, quando se chama o *main*.

Vamos criar um pacote novo chamado Principal (ainda lembra como?) e, dentro deste pacote, criar uma classe chamada Início. Vamos criar um método *main* dentro desta classe. O código deverá ficar da seguinte forma:

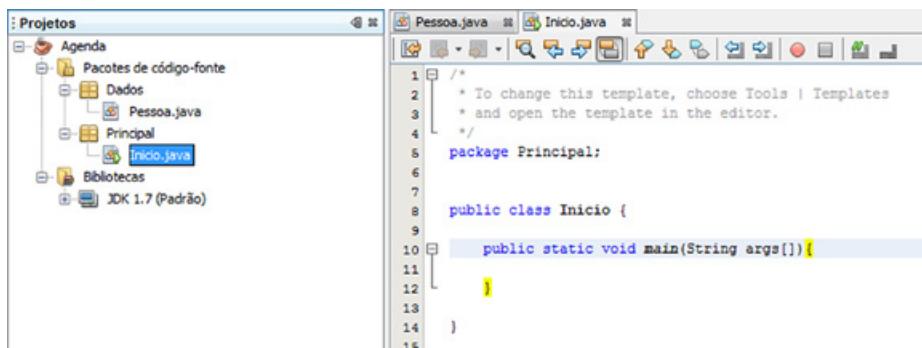


Figura 49 – Nova classe Início, dentro do pacote Principal; esta classe contém o método main

Para verificar se o seu código está funcionando, vamos mandar imprimir um “Olá Mundo!” na tela. Para isto, vamos usar o seguinte comando:

```
System.out.println("Olá Mundo!");
```

System.out indica a saída padrão do computador, que por hora é o monitor. Neste comando, utilizamos o método *println*, que imprime o que passamos por parâmetro e pula uma linha. Para imprimir na tela, também podemos utilizar métodos como *print* e *printf*, que possuem poucas diferenças, e ao longo de nossa experiência iremos verificar quais são elas.

Para executar o seu código, o Netbeans oferece uma série de possibilidades. Uma delas é clicar com o botão direito sobre o seu projeto e escolher a ação “Executar”.

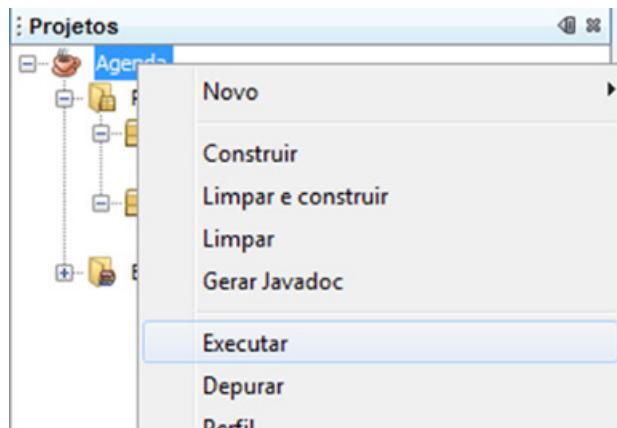


Figura 50 – Menu utilizado para executar o projeto

Todo projeto precisa indicar qual é a sua classe que contém o método *main* implementado. Caso não estiver configurado, ele irá solicitar quando for executar o seu programa. No nosso caso, basta selecionar a opção Principal.Inicio. É possível verificar que este nome está assim por causa da organização feita até o momento. Primeiro é indicar o caminho entre os pacotes e, por fim, o nome da classe.

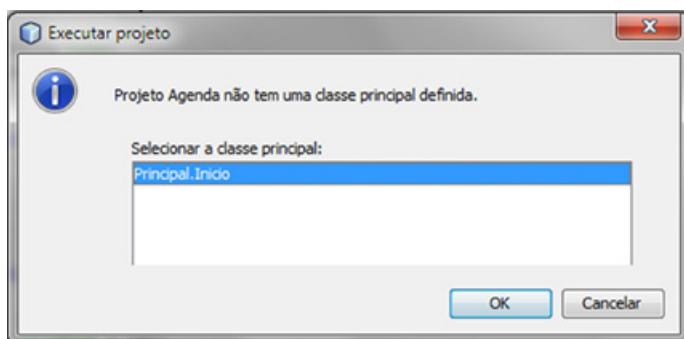


Figura 51 – Tela utilizada para indicar qual classe contém o método main

Para verificar se o seu projeto contém uma classe principal, ou se você quiser mudar esta classe em seu projeto, basta clicar com o botão direito sobre o seu projeto e escolher “Propriedades”.

Uma vez aberta a janela de propriedades, procure pela categoria “Executar”. Ao selecionar esta categoria, alguns campos irão aparecer na janela. Procure pelo campo “Classe Principal”. Este campo contém qual classe será inicializada primeiro. Você pode mudar a classe principal clicando no botão “Procurar” ao lado do campo.

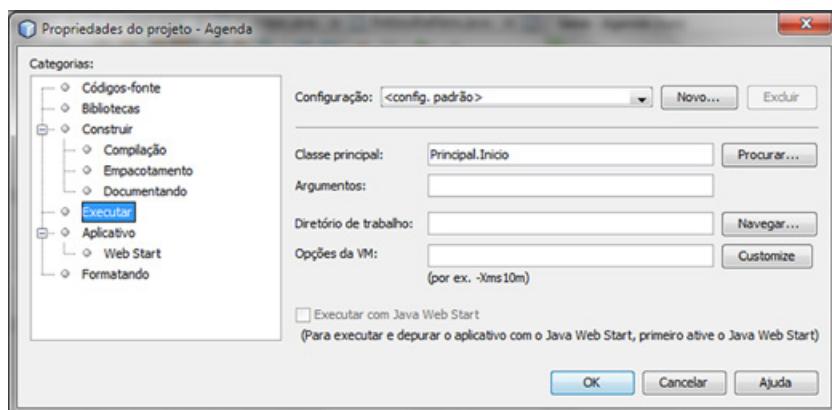


Figura 52 – Tela de Propriedades do Projeto

Se a classe principal estiver configurada corretamente, ao executar o seu projeto uma nova aba chamada “Saída” deverá aparecer na janela do NetBeans contendo o resultado de seu programa. Neste caso, apenas um “Olá Mundo!”.

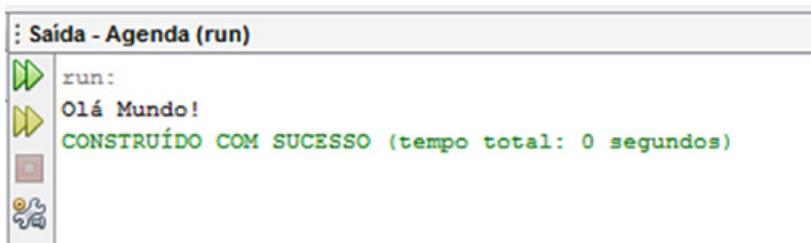


Figura 53 – Tela contendo o resultado da execução do código gerado

Mas onde estão os objetos? Como faço para criá-los e utilizá-los?

OBJETOS JAVA

Em nosso programa, até agora, apenas criamos classes. Uma ainda nem foi utilizada, enquanto que a outra age como a classe principal. A partir de agora, nossos esforços estarão voltados para criar o código dentro da classe Início, dentro do método *main*. Primeiro, vamos criar um objeto da classe Pessoa, que se encontra em outro pacote.

Depois, vamos adicionar uma linha de comando antes da classe Início, porém depois do comando que indica o pacote daquela classe.

```
import Dados.Pessoa;
```

Este comando importa para a classe uma outra classe, podendo esta ser de uma biblioteca ou de outra classe disponível na máquina. No nosso caso, vamos importar a classe Pessoa, que se encontra dentro do pacote Dados.

A partir de agora, podemos criar e usar a classe Pessoa. Vamos usá-la para criar um objeto. Dentro do método *main*, vamos inserir o seguinte comando:

```
Pessoa pes = new Pessoa();
```

O comando acima cria um objeto do tipo Pessoa, chamado pes. Este comando possui 2 etapas bem distintas. O “Pessoa pes” cria uma variável do tipo Pessoa. Seria o equivalente a colocarmos “int pes”, que seria uma variável do tipo inteiro. Porém, o objeto ainda não existe só nesta declaração. Neste momento, apenas a referência ao objeto foi criada, ficando a cargo do “new Pessoa()” a função de instanciar o objeto, ou seja, criar o objeto na memória. O comando “new” reserva o espaço na memória, que tem seu valor calculado quando informamos qual tipo de objeto será criado, indicado no “Pessoa()”.

Estas duas ações são tão distintas que podemos reescrever o código acima em 2 linhas separadas.

```
Pessoa pes;  
pes = new Pessoa();
```

Porém, existe uma regra de que você não pode utilizar o objeto enquanto ele não for criado na memória, ou seja, não for dado o comando “new” nele. Uma questão muito importante é que, ao se dar este comando, logo em seguida você determina qual construtor irá chamar. No código anterior, você chama o construtor que não recebe parâmetro algum. Logo, os atributos destes objetos estão zerados ou com *Strings* vazias. Como temos 2 construtores disponíveis para uso, para utilizar o construtor que recebe valores por parâmetros, basta mandar os valores no comando *new*. Exemplo:

```
Pessoa pes2 = new Pessoa("João", "(11) 0980-0980", 13);
```

Neste comando, estamos criando um segundo objeto chamado de pes2, onde, ao instanciar o objeto, os valores que estão sendo passados serão passados para os atributos.

Mas e o primeiro objeto, o “pes”? Agora que ele já foi criado e passou pelo construtor que inicia os atributos zerados, como podemos fazer para atribuir algum valor para ele? Podemos acessar seus atributos e métodos colocando o nome do objeto, um ponto e o atributo ou método desejado. Vamos atribuir alguns valores para seus atributos.

```
pes.nome = "Fulano";  
pes.idade = 22;  
pes.telefone = "(00) 7654-3210";
```

Porém, ao inserirmos este código acima, o NetBeans acusa que estas são linhas de programação que contêm erros. Para analisar o erro, coloque o cursor sobre o sinal de aviso no início da linha com erro.

```

11     public static void main(String args){}
12
13     nome is not public in Dados.Pessoa; cannot be accessed from outside package
14     ----
15     (Alt-Enter mostra dicas)
16     ● pes.nome = "Fulano";
17     ● pes.idade = 22;
18     ● pes.telefone = "(00) 7654-3210";

```

Figura 54 – Códigos dentro da classe Início, que contém erros

Ao ler o problema, ele está indicando que os atributos não são acessíveis de fora do pacote por não serem públicos. Para resolver este problema, podemos adotar duas medidas. A primeira e mais simples é inserir o modificador *public* para cada atributo na classe, conforme a imagem seguinte.

```

8   public class Pessoa {
9
10      public String nome;
11      public String telefone;
12      public int idade;

```

Figura 55 – Modificação nos atributos da classe Pessoa com o Public

A segunda medida que podemos tomar será discutida na próxima seção.

A partir de agora, o nosso objeto “pes” possui seus atributos com os valores determinados naquele código. Para confirmar, vamos utilizar o método *mostrarDados()* para mostrar os valores dos atributos. O comando ficará assim:

```
System.out.println(pes.mostrarDados());
```

Para verificar que cada objeto tem os seus dados, vamos também chamar o mesmo método, mas pelo objeto “pes2”.

```
System.out.println(pes2.mostrarDados());
```

O código e o resultado devem ficar desta maneira:

```
5 package Principal;
6
7 import Dados.Pessoa;
8
9 public class Inicio {
10
11     public static void main(String args[]){
12         Pessoa pes;
13         pes = new Pessoa();
14         Pessoa pes2 = new Pessoa("João", "(11)0980-0980", 13);
15         pes.nome = "Fulano";
16         pes.idade = 22;
17         pes.telefone = "(00) 7654-3210";
18         System.out.println(pes.mostrarDados());
19         System.out.println(pes2.mostrarDados());
20     }
21
22 }
```

: Saída - Agenda (run)

```
run:
Nome: Fulano Idade: 22 Telefone: (00) 7654-3210
Nome: João Idade: 13 Telefone: (11)0980-0980
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 56 – Código e resultado do código gerado e executado

Um erro muito comum que ocorre no momento que trabalhamos com mais de um objeto dentro de um método. O que acontece se colocarmos a seguinte operação:

```
pes = pes2;
```

Em um primeiro momento, parece que os atributos foram replicados e agora existem dois objetos contendo os mesmos valores. Porém, você irá perceber algo estranho quando você mudar o atributo de algum dos objetos. Se esta linha acima for executada, caso você altere o atributo nome de um objeto, o outro também será alterado. Isto porque os objetos não foram replicados, e sim apenas a variável que aponta para o objeto. Na verdade, existe um único objeto e 2

caminhos diferentes para se chegar nele. É extremamente importante fazer com que cada variável criada para servir de objeto tenha o seu próprio comando *new*. Atribuições desta forma podem levar a erros que não são facilmente identificados.

Existe um objeto especial, que podemos usar como quisermos e que nos ajuda ao criar os métodos de algumas classes. Este objeto especial é o *this*. Não há necessidade de criá-lo para utilizá-lo, porém ele tem um uso muito restrito. Ele representa o próprio objeto. Complicado? Bom, como podemos pegar o atributo nome do objeto criado pes? Basta colocar:

```
pes.nome;
```

Porém, se estamos dentro da classe “Pessoa”, como podemos acessar o próprio atributo, sendo que cada objeto tem o seu? Fácil, coloque:

```
this.nome;
```

Assim, quando chegar nesta linha, o Java saberá que é para você pegar o atributo do próprio objeto. Portanto, quando um objeto chamar uma função que contenha este código, ele vai pegar o próprio atributo e utilizá-lo. Anteriormente, ao criarmos o método de testar a idade, criamos uma variável dentro do método com o mesmo nome do atributo. Se deixarmos no *if* a condição:

```
if (idade > 0)
```

Será analisada a variável que está chegando pelo parâmetro. Se mudarmos para:

```
if (this.idade > 0)
```

O atributo que será utilizado. Vamos deixar sem o objeto *this*, pois montamos nosso método para trabalhar com o parâmetro mesmo. Com o uso do *this*, podemos até mesmo chamar construtores, mas veremos depois de aprender sobre JavaBeans.

Mas agora uma questão importante, podemos criar um construtor que recebe um objeto como parâmetro? A resposta é claro que sim, mas tem que ter certo cuidado na hora que for trabalhar com ele.

Este tipo de construtor é muito útil quando você possui objetos que você quer replicar e evitar o erro de 2 variáveis apontarem para o mesmo objeto. Basicamente, a gente recebe o objeto como parâmetro, mas dentro do construtor, cada atributo é tratado separadamente. O código desta função possui as seguintes características:

```
55     public Pessoa (Pessoa nova)
56     {
57         this.nome = nova.nome;
58         this.idade = nova.idade;
59         this.telefone = nova.telefone;
60     }
```

Figura 57 – Novo construtor, recebendo um Objeto do mesmo tipo como parâmetro

JAVABEANS OU POJOS

O que fizemos até agora foi criar duas classes, sendo que uma é a classe principal e contém o método *main*, enquanto que a outra foi criada para ser utilizada. Esta classe que foi criada com o intuito de se criar objetos dela foi feita da maneira mais fácil possível, porém fica fora das convenções adotadas para se criar classes deste tipo.

Estas convenções foram escolhidas pela sociedade de desenvolvedores para estabelecer um padrão entre as diferentes classes que são criadas ao redor do mundo, e assim, desta forma, facilitar o uso de objetos criados a partir desta

classe. Apesar deste programa funcionar sem adotar estas convenções, o ideal é segui-las para que sua classe possa ser facilmente reutilizável e para que você, acostumado com as convenções, possa facilmente utilizar outras classes.

A primeira convenção é que um construtor que não receba parâmetros seja feito para que valores vazios sejam atribuídos. Esta convenção já foi atendida pela nossa classe.

Um próximo item seria tratar todos os atributos como *private*, ao invés de *public*. Isso implica que não podemos acessar o atributo diretamente de outra classe, como fizemos no exemplo anterior. Só que se colocarmos as variáveis como privadas, nosso código ficará com problemas.

Ao colocar um atributo como *private*, faz-se necessário criar 2 métodos específicos para aquele atributo, voltados para desempenhar as 2 funções principais de um atributo, receber valor e recuperar o valor que possui.

Para isto, vamos criar os métodos para o atributo nome. O código, a seguir, cria um método para inserir um valor em nome.

```
public void setNome(String nome) {
    this.nome = nome;
}
```

Veja que foi adotada a nomenclatura `setNome` para seguir um padrão. O nome do método que irá inserir valores nos atributos sempre será `set + Nome do Atributo`. Este método normalmente não possui retorno, pois é uma operação de inserção de conteúdo. Por isso, temos um parâmetro do mesmo tipo do atributo, e este parâmetro está sendo atribuído para o atributo. Neste momento, podemos notar como o `this` consegue diferenciar o atributo da variável.

Já o método que retorna o conteúdo do atributo fica um pouco diferente:

```
public String getNome() {
    return this.nome;
}
```

É possível notar a diferença no nome, já que este faz a união do nome do atributo com a palavra *get*. Além disso, ele possui o retorno do mesmo tipo do atributo e não recebe nenhum parâmetro. Vamos ver como ficará nossa classe depois de criados todos os *gets* e *sets*.

```
10     private String nome;
11     private String telefone;
12     private int idade;
13
14     public int getIdade() {
15         return idade;
16     }
17
18     public void setIdade(int idade) {
19         this.idade = idade;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public void setNome(String nome) {
27         this.nome = nome;
28     }
29
30     public String getTelefone() {
31         return telefone;
32     }
33
34     public void setTelefone(String telefone) {
35         this.telefone = telefone;
36     }
```

Figura 58 – Criação dos métodos *get* e *set* para os atributos que ficaram privates

Uma vez os atributos privados e seus métodos de acesso criados, só nos falta alterar o código na classe “Início” para que utilize estes métodos de acesso. Vamos substituir as linhas voltadas à inserção de conteúdo pelos seus respectivos métodos *set*. Como vamos utilizar um método, não há necessidade do sinal de igual. Basta passar o valor desejado dentro dos parênteses como parâmetros.

```

15     pes.setNome("Fulano");
16     pes.setIdade(22);
17     pes.setTelefone("(00) 7654-3210");

```

Figura 59 – Forma de utilização dos métodos set e get na classe Início

Você pode estar se perguntando: mas por que toda esta volta? O código está fazendo a mesma coisa que antes, mas está maior e mais complexo. O que ganhamos com isso? Realmente o código fica maior e mais complexo, porém agora você, como dono da classe que criou, tem um controle maior em como os objetos criados a partir dela trabalham com seus atributos. Vamos supor que no seu programa é de extrema importância que não possa ter idade negativa. Antes ficava a cargo de quem utilizava seu objeto de cuidar que isto não ocorra, mas agora você pode controlar isto.

É bem simples, basta apenas lembrar que dentro dos métodos podemos adicionar qualquer comando de programação, e que estes *sets* e *gets* são métodos. Para resolver o problema da idade negativa, vamos adicionar um controle dentro do *setIdade*. Assim, o objeto pode tentar inserir uma idade negativa, mas não conseguirá. Dessa forma os atributos ficam mais protegidos e sua classe menos propensa a cometer erros.

Para este problema em específico, temos criado já um método *testIdade*, que verifica exatamente esta condição. Vamos fazer uso deste método dentro do método *setIdade* e caso seja negativo, atribuir o valor 0 à idade.

```

public void setIdade(int idade) {
    if (testeIdade(idade))
        this.idade = idade;
    else
        this.idade = 0;
}

```

Como o método está na mesma classe que o método *setIdade*, não há necessidade de se criar um objeto para chamá-lo. Assim, o parâmetro que está chegando é passado para o método *testIdade* que, por sua vez, retorna verdadeiro ou falso. Este retorno é utilizado pelo comando *if* para direcionar o código.

Com os métodos *get* e *set* criados, o ideal é que mesmo dentro da classe, eles sejam utilizados. Assim, não há qualquer meio de um atributo não passar pelo teste imposto no método. No nosso exemplo, vamos alterar tanto os construtores, quanto o método *mostrarDados()*. Verificamos na alteração também que os métodos *get* e *set* estão sendo chamados como o uso do *this*, porém não há necessidade deles. Como veremos mais adiante, os métodos fazem parte do comportamento dos objetos, o que é algo compartilhado com todos. Foi utilizado o *this* apenas para enfatizar como utilizá-lo na classe.

```
42     public Pessoa() {
43         this.setIdade(0);
44         this.setNome("");
45         this.setTelefone("");
46     }
47
48     public Pessoa(String nom, String tel, int id)
49     {
50         this.setIdade(id);
51         this.setTelefone(tel);
52         this.setNome(nom);
53     }
54
55     public Pessoa (Pessoa nova)
56     {
57         this.setNome(nova.getNome());
58         this.setTelefone(nova.getTelefone());
59         this.setIdade(nova.getIdade());
60     }
61
62     public String mostrarDados()
63     {
64         String retorno = "Nome: " + this.getNome() + " Idade: " +
65             this.getIdade() + " Telefone: " + this.getTelefone();
66         return retorno;
67     }
```

Figura 60 – Modificação nos métodos para todos usarem apenas *get* e *set*

Um código mostrado acima que vale a pena comentar é a atribuição de um objeto para o outro. Vamos pegar a atribuição do atributo nome:

```
this.setNome(nova.getNome());
```

Neste comando, dois métodos de dois objetos diferentes estão sendo chamados. Primeiro é chamado o método `getNome()` do objeto `nova`. Por ser um `get`, ele não recebe parâmetros e retorna uma `String`. Este retorno está sendo utilizado como forma de parâmetro para o `setNome` do objeto `this`, ou seja, dele próprio. Sempre que há vários métodos aninhados pela utilização dos mesmos como parâmetros para outros, sempre leve em conta que o método que está mais interno nos parênteses será executado primeiro.

Outra forma que podemos fazer com os construtores é direcionar os outros construtores para um único construtor. Complicado? Nesta classe nós temos 3 construtores, que basicamente fazem a mesma coisa, porém com atributos em quantidades e formatos diferentes. Porém, o código dentro deles é muito parecido. Por que, em vez de ficar replicando as chamadas dos métodos `set` em cada construtor, não elegemos um único construtor que terá estas chamadas dos métodos `set` e direcionamos os outros para o construtor eleito? É bem simples se utilizarmos o `this`.

Normalmente o construtor eleito para este trabalho é aquele que recebe os atributos separadamente por parâmetro. Começando pelo primeiro construtor, vamos chamar o construtor eleito da seguinte forma:

```
this("", "", 0);
```

O comando `this` é usado de uma forma diferente neste caso. Não há necessidade do uso de pontos para chamada de método, apenas o `this` e os parâmetros. Como o construtor eleito recebe nome, telefone e idade, nesta ordem, passamos por parâmetro os valores que ele deverá colocar em cada atributo. Já no construtor que recebe o objeto como parâmetro, o comando ficará da seguinte forma:

```
this(nova.getNome(), nova.getTelefone(), nova.getIdade());
```

Simplesmente estamos pegando cada atributo separadamente e mandando em forma de parâmetro para o construtor eleito. Os três construtores juntos ficam da seguinte forma:

```
42     public Pessoa(){  
43         this("", "", 0);  
44     }  
45  
46     public Pessoa(String nom, String tel, int id)  
47     {  
48         this.setIdade(id);  
49         this.setTelefone(tel);  
50         this.setNome(nom);  
51     }  
52  
53     public Pessoa (Pessoa nova)  
54     {  
55         this(nova.getNome(), nova.getTelefone(), nova.getIdade());  
56     }
```

Figura 61 – Modificação nos construtores, para que todos apontem para um único construtor

ESTADO E COMPORTAMENTO

A partir de uma única classe, podemos criar diversos objetos. Porém, existe alguma forma de distinguir um objeto do outro? O que eles possuem em comum? Vamos analisar o que é feito na classe. As classes são compostas por um conjunto de atributos e métodos. O que ficou claro nos exemplos anteriores é que cada objeto criado possui seus próprios atributos. Mesmo que for da mesma classe, cada um possui o mesmo número e tipo de variáveis, mas cada um com o seu valor. Neste caso, a classe só informa o modelo do que cada objeto irá conter, mas o valor é gerenciado pelo objeto.

Este conjunto de atributos de um objeto define o estado em que ele se encontra. Cada objeto, por ter os seus próprios atributos, pode estar em estados diferentes. Por exemplo, dois objetos criados a partir da classe *File*, que gerencia arquivos do sistema. Um objeto pode estar com um documento em aberto e pronto para alterar, enquanto que o outro pode já estar fechado, não podendo mais alterar o arquivo.

Já o comportamento que o objeto tem é algo descrito e compartilhado entre todos os objetos por meio dos métodos. Estes possuem um código criado pelo

desenvolvedor, que não se altera de objeto para objeto, logo, todos possuem as mesmas funções. Seus atributos podem acabar influenciando em um método, mas todos os objetos têm acesso a todas as funções que os outros objetos têm.

Podemos ver esta questão no nosso exemplo. Quando criamos dois objetos, ao chamar a função setIdade(), ela irá atribuir o valor passado ou 0, dependendo do valor. Porém, todos os objetos estão sujeitos a executar qualquer parte do código.

COMPARANDO OBJETOS

Se voltarmos algumas páginas, no momento em que é citado sobre a criação de um objeto, vemos que o objeto é criado no momento em que usamos o comando *new*. Antes disso, é criada uma variável (no nosso exemplo o “pes” e o “pes2”), que vai servir de ponte para o nosso objeto quando resolvemos criá-lo. Ou seja, podemos criar esta variável, e só depois de muitas linhas de código chamar o *new*.

Isso nos faz pensar, o que é esse pes se ele existe antes mesmo de criar o objeto? Podemos pensar nele como um ponteiro que aponta para o objeto quando criado. Ou seja, quando executamos o seguinte comando:

```
Pessoa pes;
```

Estamos criando um ponteiro que não aponta para lugar algum. A partir do momento que fazemos:

```
pes = new Pessoa();
```

O ponteiro pes recebe o endereço de onde o comando *new* criou o objeto. Para termos certeza disto, basta colocar o seguinte comando:

```
System.out.println(pes);
```

Irá aparecer um valor numérico precedido de um arroba e algumas letras. Isto na verdade é o valor que pes possui, o endereço de seu objeto. Por isso, normalmente não trabalhamos com a variável pes diretamente. Sempre colocamos um ponto para acessar o conteúdo do objeto apontado, como um método ou atributo.

Vamos efetuar uma alteração no nosso código. Quero que seja criado um código que analise quem é a pessoa que possui a maior idade e mostre o nome dela na tela.

Baseado no código produzido até aqui, vamos acrescentar o seguinte código dentro do método *main*, depois dos códigos já criados:

```
if (pes > pes2)
    System.out.println(pes.getNome());
else
    System.out.println(pes2.getNome());
```

Apesar do NetBeans não apontar erro na sintaxe, nosso código acima contém um erro grave. Ao compararmos os objetos por meio de seus ponteiros, na verdade estamos comparando qual endereço de memória possui o maior valor. Logo, nosso código vai mostrar o nome de “pes” se o endereço apontado em pes for maior. Para que seja efetuada a comparação correta, precisa-se especificar qual atributo está sendo testado.

```
if(pes.getIdade() > pes2.getIdade())
```

CONSIDERAÇÕES FINAIS

Vimos nesta Unidade como criar Projetos dentro do NetBeans, além da criação de classes e da organização das mesmas dentro de pacotes. Também vimos que as classes podem e devem ser trabalhadas para que os atributos sejam protegidos de alterações externas, deixando-os privados e acessíveis somente por meio de métodos *get* e *set*.

Também vimos que existem os construtores, métodos especiais que são chamados no momento que o objeto de uma determinada classe é criado. Os objetos foram criados a partir de outra classe, e chamando o construtor, o objeto passa a existir e assim podemos acessar seus métodos e atributos que estão disponíveis.

Agora você deverá ser capaz de criar classes simples, além de conseguir utilizá-las.

ATIVIDADES



1. Vamos utilizar nosso conhecimento para ver se conseguimos resolver um problema real. Nosso dinheiro é uma das coisas mais importantes que temos nos dias de hoje. Para isso, precisamos de um programa confiável que gerencia exatamente quanto temos. Crie uma classe chamada Conta que consiga gerenciar o quanto de dinheiro temos, onde as únicas operações que podemos fazer com o nosso dinheiro guardado é sacar e depositar. Crie uma classe nova chamada Início, que irá conter o método *Main*, e crie dois objetos da classe Conta, um chamado contaCorrente e outro Poupança.

MATERIAL COMPLEMENTAR



NA WEB

Java Classes e Objetos

Por Mic Duarte

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://forum.imasters.com.br/topic/336253-curso-java-classes-e-objetos/>>.



NA WEB

Diferenças entre classes e objetos em Java

Disponível em: <http://www.youtube.com/watch?v=biGRUS_3NbY>.

MODIFICADORES JAVA E ENCAPSULAMENTO

UNIDADE

IV

Objetivos de Aprendizagem

- Entender o conceito de encapsulamento.
- Estudar os modificadores de acesso public, protected e private.
- Entender a diferença dos modificadores static, abstract e final.
- Controlar o acesso a métodos, atributos e construtores por meio de modificadores.
- Entender o que são Construtores de Classes.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- O que é encapsulamento
- Modificadores de acesso public, private e protected
- Utilizando modificadores de acesso (getters e setters)
- Modificador static
- Modificador final
- Modificador abstract
- Construtores

INTRODUÇÃO

Olá, seja bem-vindo(a) ao módulo de Modificadores de Acesso em JAVA. Nesta Unidade, serão estudados conceitos importantes para a programação Orientada a Objetos, pois é a partir dos modificadores que poderemos tratar do encapsulamento das classes.

A ideia de encapsulamento remete ao fato de que é possível agrupar estados e comportamentos em um mesmo módulo e ainda controlar suas propriedades e seu acesso. Para isso, serão tratados ainda neste módulo como escrever métodos de acesso aos elementos de uma classe utilizando os chamados métodos *getters* e *setters*.

Além dos modificadores de acesso, existem outros modificadores que podem ser utilizados para atribuir novas funções aos métodos e atributos, e estes podem ser utilizados em conjunto com modificadores de acesso. São eles: *static*, *final* e *abstract*. E, por fim, será mostrado como é possível implementar Construtores para as classes e desta forma controlar como os objetos de uma classe serão inicializados.

ENCAPSULAMENTO

Quando se fala de encapsulamento, você provavelmente imagina um objeto fechado, dentro de uma cápsula, e isto não é muito diferente na programação Orientada a Objetos, pois a ideia principal do encapsulamento envolve omitir os membros de uma classe, além de esconder como funcionam as rotinas ou regras de negócio. Desta forma, o programador deve se atentar às funcionalidades da Classe, e não como ela foi implementada, e isto é bom uma vez que há aumento na modularidade do seu projeto.

Sendo assim, realizar o encapsulamento de um projeto é fundamental para que seja possível minimizar o impacto de problemas referentes a alterações do projeto, uma vez que não é preciso alterar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que tal regra está encapsulada.

Para exemplificar, vamos focar no problema de realização de operações bancárias de um caixa eletrônico. No caixa é possível: sacar dinheiro, depositar, verificar saldo, verificar extrato, pagar contas e fazer transferências. Para utilizar o caixa eletrônico, não é necessário conhecer como as operações foram implementadas em nível de programação, mas sim o que cada operação afeta sua conta.

Analise a Figura 62 logo abaixo e verifique a classe Conta, com atributos e métodos para uma Conta Corrente. Note, por exemplo, que o método transferir recebe como parâmetro o número da conta (numContaDestino) e um valor a ser transferido (valor). Para utilizar este método, o programador não precisa saber como o criador desta classe implementou o método transferir, basta saber que este método é responsável por enviar um valor a outra conta corrente e que esta transferência implica em subtração do saldo da conta remetente.

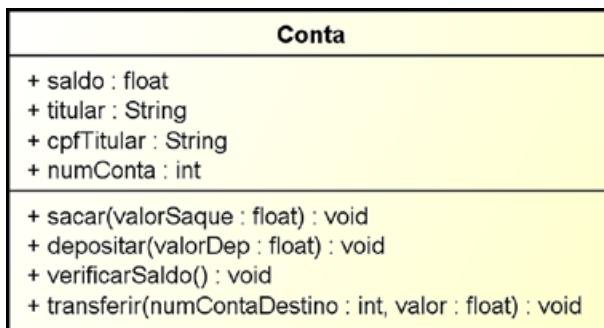


Figura 62 – Classe Conta com atributos e métodos públicos

Fonte: o autor

Ainda em relação à Figura 62, esta está ilustrada em um diagrama de Classe UML, e o símbolo de soma (“+”) indica que tanto os atributos quanto os métodos são públicos, isso significa que eles podem ser acessados e modificados de qualquer outra classe do projeto, e isto não é uma boa prática de programação, uma vez que seus atributos estão expostos e qualquer outro objeto pode alterar os valores dos elementos das classes, o que pode comprometer a coesão de seu projeto. Por exemplo:

```
//Criação da referência a Conta  
Conta c = new Conta();  
//Modificação do atributo saldo de forma direta  
c.saldo = 0;
```

Para solucionar este problema, faz-se necessária a utilização de modificadores de acesso, sendo que estes farão o papel de restringir ou autorizar o acesso a determinados atributos ou métodos das classes, e isto será visto a partir da próxima seção.

Modificadores de Acesso: *default, public, private e protected*

É a partir dos modificadores de acesso que poderemos restringir ou não o acesso aos atributos e métodos de uma classe. Na programação Orientada a Objetos, os modificadores de acesso mais utilizados são o *public* e o *private*.

Antes de abordar modificadores, é importante frisar alguns conceitos básicos sobre a programação em JAVA: Projeto e Pacotes do projeto.

Um projeto é a base para o desenvolvimento em Java, é nele que estarão todos os pacotes e classes da sua aplicação. Quando trabalhamos com o Netbeans, para a criação de um projeto, basta irmos ao menu “Arquivo” e, em seguida, selecionar a opção “Novo Projeto”, feito isso, aparecerá uma tela, como você pode verificar na Figura 63 a seguir:

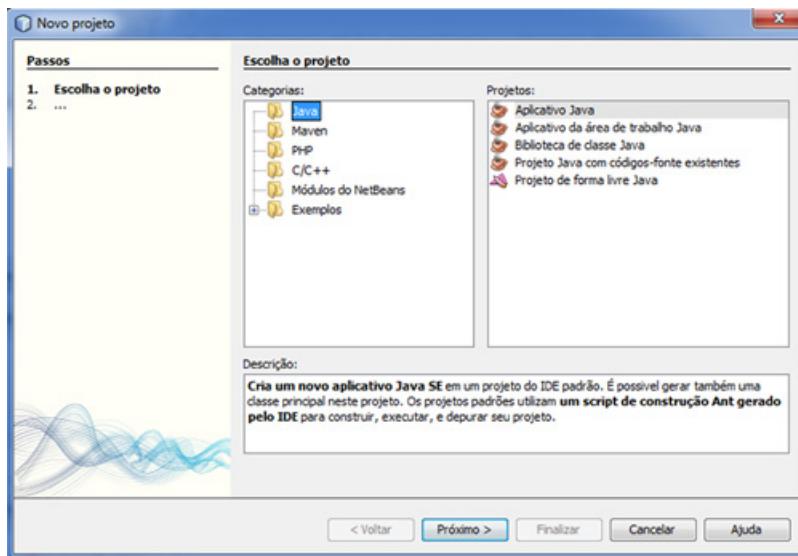


Figura 63 – Criando um projeto em JAVA no Netbeans

Como o Netbeans é uma ferramenta que suporta diversas linguagens de programação, ao criar o projeto, é necessário previamente selecionar a linguagem e, em seguida, o tipo de projeto, sendo assim, vamos criar um Aplicativo Java selecionando a opção “Aplicativo Java”. Feito isso, uma nova tela irá solicitar o preenchimento do nome do projeto, como você pode verificar na Figura 64:

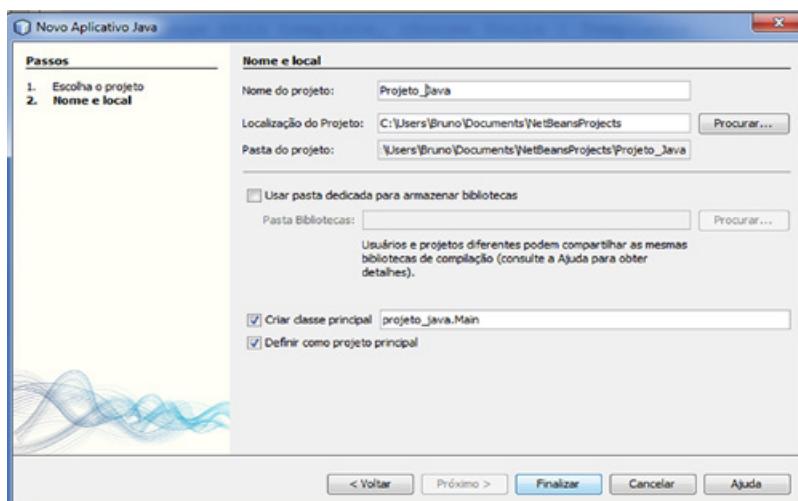


Figura 64 – Configurando o nome do projeto

Após estas etapas, seu projeto estará criado e poderá ser visualizado na Paleta de “Projetos” do Netbeans, como pode ser verificado na Figura 65. Como você pode notar, um projeto envolve Pacotes e Bibliotecas que, porventura, sua aplicação necessite utilizar.

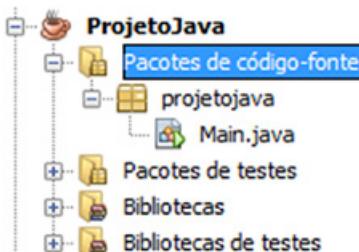


Figura 65 – Projeto em Java

Para criar um novo pacote na pasta “Pacotes de código-fonte”, por exemplo, é necessário clicar sobre a pasta com o botão direito e selecionar a opção: “Novo” -> “Pacote Java” e, em seguida, dar um nome ao seu novo pacote, como mostra a Figura 66.

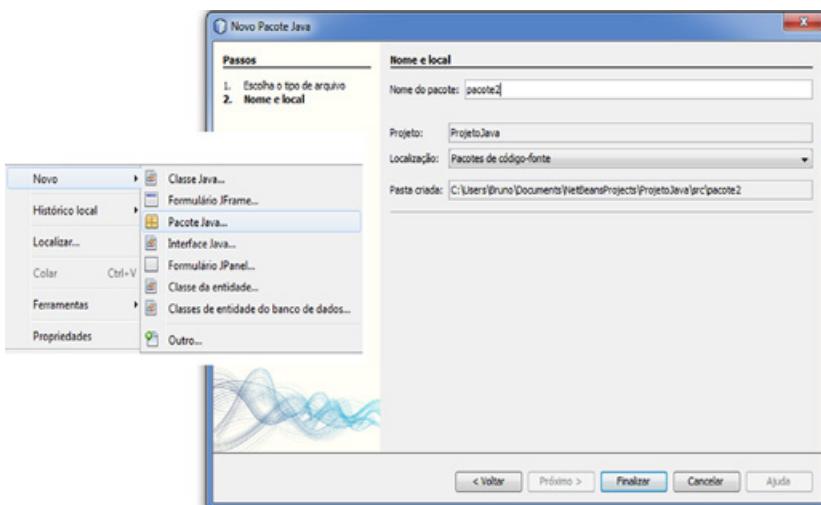


Figura 66 – Criando um novo pacote

Agora que você já entendeu o conceito de Pacotes, podemos iniciar a teoria sobre modificadores de acesso. Temos quatro modificadores de acesso: *public*, *private*, *protected* e *default*. O modificador *default* não precisa ser declarado, ou seja, atributos ou métodos que não têm a declaração de um modificador de acesso são considerados *default* (que do inglês significa “padrão”). Como exemplo, você pode notar no código abaixo a criação de um atributo com modificador de acesso *default*. No pacote em que está localizada a classe criada, o atributo com modificador *default* pode ser acessado de forma direta por qualquer classe, agora, se a classe que instanciar a classe estiver em outro pacote, este não será acessível de forma direta.

```
//exemplo de atributo default
String exemploDefault;
```

Atributos e métodos públicos (*public*) podem ser acessados e modificados de qualquer classe do projeto a partir da instância do objeto, ou seja, até de outros pacotes é possível ter acesso a atributos/métodos do tipo *public* de forma direta. Para declarar este tipo de modificador, é necessário utilizar a palavra reservada *public* antes de indicar o tipo de variável, isto pode ser verificado no exemplo de código-fonte abaixo.

```
//exemplo de atributo public
public int inteiroPublic;
//exemplo de método public
public int soma(int a, int b){
    return a+b;
}
```

Já atributos e métodos privados (*private*) são acessados somente pela própria classe, independente do pacote em que a classe esteja. O código-fonte abaixo mostra como se pode criar um atributo privado. Métodos privados só podem ser acessados pela própria classe, ou seja, eles não são visíveis e nem acessados por outros objetos.

```
//exemplo de atributo private
private String nomePrivate;
//exemplo de método private
private String[] splitNome(String n) {
    return n.split(" ");
}
```

Outro modificador de acesso não tão comumente utilizado é o modificador *protected* (protegido em português), este tipo de modificador funciona como um modificador público para elementos que estão no mesmo pacote, e em outros pacotes ele só é visível para classes que herdam a classe que possui o atributo protegido. A criação de atributos/métodos protegidos se faz por meio da palavra reservada *protected* antes da criação do atributo/método.

```
//exemplo de atributo protected
protected String nomeCliente;
//exemplo de método protected
protected String[] splitNome(String n) {
    return n.split(" ");
}
```

Baseado nas explicações dadas até aqui sobre os modificadores, é possível montar uma tabela que resume se o atributo/método é acessível ou não dependendo de sua localização no projeto.

MODIFICADOR	NA PRÓPRIA CLASSE	EM OUTRO PACOTE	NO MESMO PACOTE	HERANÇA (EM PACOTE DIFERENTE)
public	Acessível	Acessível	Acessível	Acessível
private	Acessível	Não	Não	Não
protected	Acessível	Não	Acessível	Acessível
default	Acessível	Não	Acessível	Não

Tabela 6 – Modificadores de acesso e suas visibilidades

É importante que você, como um futuro bom programador, entenda bem esta tabela, e a tenha sempre com você para eventuais consultas. Sendo assim, na próxima seção vamos mostrar na prática como utilizar os modificadores de acesso de modo a manter o encapsulamento de suas aplicações.

UTILIZANDO MODIFICADORES DE ACESSO

Tendo como base um projeto exemplo de um Caixa Eletrônico, onde o objetivo é realizar operações em contas bancárias, vamos elucidar a utilização dos modificadores de acesso, além disso, trazer o conceito dos *getters* e *setters*. Para isso, veja como foi projetada a aplicação na Figura 67.

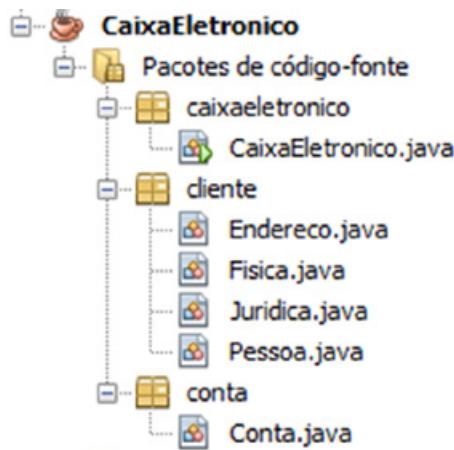


Figura 67 – Exemplo de projeto, seus pacotes e códigos-fonte

Como você pode notar, o projeto possui 3 pacotes: “caixaelectronico”, “cliente” e “conta”. Cada pacote tem suas classes que possuem suas particularidades e serão descritas uma a uma a seguir.

O pacote “caixaelectronico” possui a classe “CaixaEletronico” e esta é a classe inicial do projeto, é nesta classe que se inicia a execução da aplicação, pois esta contém o método principal (*main*) e ela será nosso gerenciador de contas.

O pacote “cliente” possui classes que estruturam os objetos Clientes que podem ser pessoa física (Fisica.java) ou jurídica (Juridica.java), além de estruturar como será o formato do endereço de cada pessoa (Endereco.java).

E, por fim, o pacote “conta” possui a classe Conta.java que armazena as informações de uma conta. No diagrama, a seguir, mostrado pela Figura 68 é possível ter uma visão geral de cada Classe do projeto.

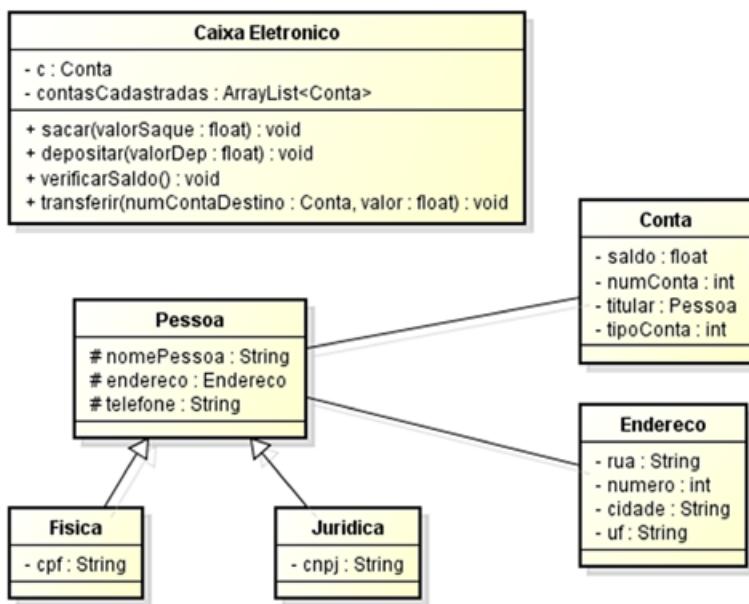


Figura 68 – Diagrama de Classe do projeto de caixa eletrônico

Fonte: o autor

Tendo como base os diagramas da Figura 68, vamos primeiramente entender o Diagrama de Classes em questão, note que antes dos nomes dos atributos e métodos das classes existe um sinal (“-”, “+” ou “#”), em UML, o sinal de “-” significa que trata-se de um atributo/método que utiliza modificador de acesso *private*, o sinal “+” indica o modificador *public* e o sinal “#” indica o sinal de modificador *protected*. Note ainda que as classes Física e Jurídica se ligam à classe Pessoa com uma seta grande e vazia, e esta ligação significa que Física e Jurídica são “filhas” de Pessoa, ou seja, herdam todas as propriedades da classe Pessoa (o conceito de herança será abordado na próxima Unidade e lá retomaremos este projeto).

Agora você deve estar se perguntando: “se houve restrição do acesso a alguns atributos das Classes apresentadas, como será realizada a alteração das informações quando forem necessárias?”. A resposta é relativamente simples, basta criar métodos públicos que controlarão os acessos a estas variáveis, ou seja, será criada uma interface de acesso para cada atributo privado (*private*) da classe em questão, é aí que entra o conceito de *getters* e *setters* na programação Orientada a Objetos e este é um mecanismo de acesso a dados encapsulados.

Sendo assim, vamos aprender a criar métodos que alteram os valores dos atributos privados; a estes métodos damos o nome de *setters*, ou seja, estes métodos irão “setar” os valores dos atributos privados. Para criar um método *setter*, primeiro faz-se necessário analisar o tipo do atributo. Vamos pegar, por exemplo, o atributo saldo da classe Conta; por se tratar de um atributo do tipo *float* (real), o método para setar este atributo deverá receber como parâmetro um elemento também *float*. Como é um método que não terá retorno, então, utiliza-se o tipo de retorno como *void*. Além disso, o padrão para métodos *setters* é sempre utilizar a palavra “set” e, em seguida, concatenar com o nome da variável, no caso em tela, teríamos o método de nome *setSaldo*, que terá sua implementação da seguinte forma:

```
//metodo setSaldo - setter para o atributo saldo
public void setSaldo(float s) {
    this.saldo = s;
}
//metodo setNumConta - setter para o atributo numConta
public void setNumConta(int n) {
    this.numConta = n;
}
```

Além do método de alterar o valor do atributo, precisamos utilizar o método para “resgatar” o valor deste atributo, a estes métodos chamamos de *getters* e a característica deste tipo de método é que ele retorna o valor do atributo para o elemento que o chama. Os métodos “get” são precedidos pela palavra “get” e o nome do atributo, e eles devem retornar elementos do mesmo tipo de atributo em questão. Utilizando o mesmo exemplo do atributo Saldo e o atributo numConta, teremos os métodos *getters* abaixo:

```
//metodo getSaldo - getter para o atributo saldo
public float getSaldo(){
    return this.saldo;
}
//metodo getNumConta - getter para o atributo numConta
public int getNumConta(int n){
    return this.numConta;
}
```

Como exemplo, a Conta foi reimplementada, e utilizando os métodos *getters* e *setters* de maneira adequada, esta ficará como mostrado no código a seguir. Note que para o atributo saldo a implementação do *getter* e *setter* não foi feita, tendo em vista que existem três operações básicas que devem “controlar” esta variável em um ambiente de caixa eletrônico, que são as operações: sacar, transferir e depositar, e serão implementadas posteriormente.

```
package conta;
import cliente.Pessoa;
public class Conta {
    private float saldo;
    private Pessoa titular;
    private int numConta;
    private int tipoConta;
    public int getNumConta() {
        return numConta;
    }
    public void setNumConta(int numConta) {
        this.numConta = numConta;
    }
    public int getTipoConta() {
        return tipoConta;
    }
    public void setTipoConta(int tipoConta) {
        this.tipoConta = tipoConta;
    }
    public Pessoa getTitular() {
        return titular;
    }
    public void setTitular(Pessoa titular) {
        this.titular = titular;
    }
}
```



REFLITA

Dica da ferramenta: no Netbeans é possível a criação “automática” dos métodos *getters* e *setters*, para isso, no código-fonte de sua classe, após a declaração dos atributos, pressione a tecla Alt+Insert e escolha a opção “*getters e setters...*” e então selecione os atributos que deseja gerar os métodos.

Além dos modificadores de acesso, o JAVA permite a utilização de outros modificadores, são eles: *static*, *final* e *abstract*. Cada um deles será abordado em seções, pois eles têm características bem diferentes entre si.

O MODIFICADOR STATIC

O modificador *static* pode ser aplicado a variáveis e métodos, e a principal característica dele é que se tratando de atributos, todos os objetos compartilham do mesmo espaço de memória, e se tratando de método, este pode ser acessado sem a necessidade de instância do objeto.

Você provavelmente já criou alguma aplicação em JAVA e se deparou com o método “public static void main (String args[])”, e deve ter se perguntado: “O que significa todas estas palavras antes do nome do método principal?”. Então vamos por partes. O modificador de acesso *public* indica que este método pode ser visualizado de qualquer classe do projeto, o modificador *static* indica que para acessar o método da classe não é necessário instanciá-lo, e o que acontece é que métodos e variáveis *static* são alocadas em memória antes que qualquer objeto seja criado, por essa razão, um atributo não *static* da classe que não esteja no método *static* não pode ser acessado de forma direta, pois ele só existirá a partir da instância do objeto, por exemplo, no código a seguir da classe Exemplo, temos duas variáveis valor (int) e valor2(static int), note que do método *static*, ao tentar atribuir um inteiro à variável valor, o NetBeans acusa um erro com os seguintes dizeres: “non-static variable valor cannot be referenced from a static context”, traduzindo, isto significa que a variável não *static* valor não pode ser referenciada em um contexto *static*.

```
public class Exemplo {  
    int valor;  
    static int valor2;  
    public static void main(String[] args) {  
        valor = 30;  
        valor2 = 10;  
    }  
}
```

Para resolver este erro, temos duas saídas: uma é transformar o atributo valor em *static*, a outra é criar um objeto (instância) do tipo Exemplo e então acessar e modificar a variável valor. Veja, a seguir, como ficariam as duas soluções.

```
public class Exemplo {  
    static int valor;  
    static int valor2;  
    public static void main(String[] args) {  
        valor = 30;  
        valor2 = 10;  
    }  
}  
  
public class Exemplo {  
    int valor;  
    static int valor2;  
    public static void main(String[] args) {  
        Exemplo e = new Exemplo();  
        e.valor = 30;  
        valor2 = 10;  
    }  
}
```

Uma variável estática é compartilhada por todas as instâncias de uma classe, ou seja, em vez de cada instância da classe ter uma cópia dessa variável, ela é uma única variável compartilhada por todas as instâncias. Para a existência de uma

variável estática não é preciso nem mesmo a criação de uma instância da classe que contenha a variável, é necessário somente que a classe seja carregada na Máquina Virtual Java, já que esta é criada e inicializada no momento de carga da classe.

Exemplificando, imagine que haja um limite para realizar transferências entre contas e que este limite seja igual para todas as contas. Para resolver este problema, basta declarar o atributo `limiteTransferencia` como *static*, desta forma, o limite é alterado para todas as referências de conta. Veja a Figura 69, a seguir, onde é ilustrado a variável `limiteTransferencia` como sendo um atributo compartilhado.

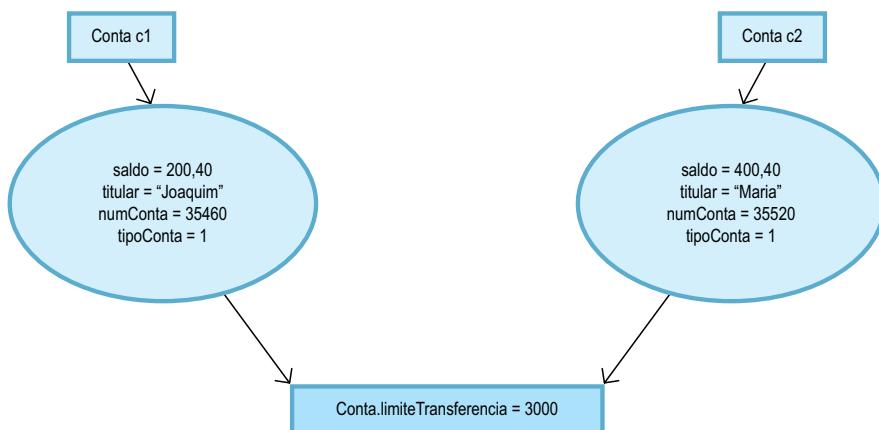


Figura 69 – Ilustração de variável static `limiteTransferencia`

Fonte: o autor

O MODIFICADOR FINAL

Os modificadores *final* restringem ainda mais o acesso aos elementos de uma classe, para atributos, ele faz com que o atributo não possa ser modificado em tempo de execução, ou seja, cria-se uma variável que terá um valor constante do início ao término da execução da aplicação. Para Classes, indica que esta não poderá ser herdada (não poderá ter filhos) e para métodos, indica que o mesmo não poderá ser sobreescrito (usar técnicas de polimorfismo).

Variáveis de instância do tipo *final* podem ser declaradas em conjunto com o modificador *static*. Neste caso, se ele for declarado na inicialização da variável, o valor atribuído a ele será o mesmo para todos os objetos e não poderá ser modificado durante a execução do projeto, se uma variável *static final* não for inicializada, ocorrerá um erro de compilação. Podemos então utilizar o exemplo do limite de transferência para ilustrar a utilização do modificador final.

```
public class Conta{  
    ...  
    private final static float LIMITETRANSFERENCIA=3000;  
    ...  
}
```



Dica de programação: quando se declara um atributo do tipo *final*, usa-se como padrão a utilização do nome do atributo em caixa alta, por se tratar de um atributo constante.

Métodos *final* não podem ser sobreescritos, ou seja, um método *final* em uma superclasse (classe pai) não pode ser reimplementado na subclasse (classe filha). Os métodos declarados como *private* são implicitamente *final*, porque não é possível sobre escrever em uma subclasse.

Uma classe *final* não pode ser superclasse, ou seja, não pode ter classes que herdam suas propriedades. Portanto, quando uma classe é *final*, implicitamente todos os métodos são *final*.

O MODIFICADOR *ABSTRACT*

O modificador *abstract* é aplicado somente a métodos e a classes, métodos abstratos não fornecem implementações e em classes abstratas não é possível a criação de objetos da classe, e normalmente possuem um ou mais métodos abstratos.

O objetivo de criação de classes abstratas é fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e assim poder compartilhar um design comum. Veja na Figura 70, abaixo, a criação da classe Pessoa como sendo uma classe abstrata baseada no projeto do caixa eletrônico, esta classe possui também um método abstrato para cadastro. A declaração do método abstrato “força” a programação do método nas subclasses.

```
public abstract class Pessoa {  
    protected String telefone;  
    protected String nomePessoa;  
    protected Endereco e = new Endereco();  
    //Método abstrato para cadastro de pessoa  
    public abstract void cadastra();  
    public Endereco getE() { ... }  
    public void setE(Endereco e) { ... }  
    public String getNomePessoa() { ... }  
    public void setNomePessoa(String nomePessoa) { ... }  
    public String getTelefone() { ... }  
    public void setTelefone(String telefone) { ... }  
}
```

Figura 70 – Implementação da classe abstrata Pessoa e o método abstrato cadastra

A Figura 71, logo a seguir, mostra a “tentativa” de criação de um objeto de uma classe abstrata, no detalhe a mensagem do IDE de que uma classe abstrata não pode ser instanciada.

```
import cliente.Pessoa;  
  
cliente.Pessoa is abstract; cannot be instantiated  
  
public class CaixaEletronico {  
    public static void main(String[] args) {  
        Pessoa p = new Pessoa();  
  
    }  
}
```

Figura 71 – Tentativa de criação de um objeto da classe abstrata Pessoa

Vamos criar uma subclasse para a classe Pessoa. Note que o método cadastra precisou ser implementado na classe Física, veja na Figura 72 esta implementação.

```
public class Fisica extends Pessoa{  
  
    private String cpf;  
  
    //implementacao do metodo abstrato é imprescindivel  
    @Override  
    public void cadastra() {  
        //leitura via teclado  
        Scanner tec = new Scanner(System.in);  
        System.out.println("Digite o nome");  
        nomePessoa = tec.nextLine();  
        System.out.println("Digite o telefone");  
        telefone = tec.nextLine();  
        System.out.println("Digite o cpf");  
        cpf = tec.nextLine();  
        e.cadastra();  
    }  
}
```

Figura 72 – Classe Pessoa, subclasse da classe abstrata Pessoa

CONSTRUTORES JAVA

Um Construtor não é um método, pois este não possui a declaração de retorno. Mas lembre-se disso: toda classe em Java tem pelo menos um construtor, exceto interfaces.

O que são esses construtores? Um construtor é o primeiro “método” que é executado sempre que uma classe é instanciada. Quando se utiliza a palavra-chave *new*, o construtor será executado e inicializará o objeto. Existem classes que quando inicializadas requerem algum tipo de parâmetro, por exemplo, objetos do tipo Scanner (para realizar leitura de informações para o projeto), que você pode conferir no código a seguir.

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

Quando fazemos isto, estamos inicializando a classe com os dados parâmetros. O construtor inicializa as variáveis de instância da classe, como também executa códigos necessários para a inicialização de uma classe. Em outras palavras, no Construtor você pode determinar o que será realizado assim que seu objeto for instanciado.

Vamos utilizar o exemplo da Classe Pessoa e criar um construtor para ela:

```
public abstract class Pessoa {  
    protected String telefone;  
    protected String nomePessoa;  
    protected Endereco e = new Endereco();  
  
    public Pessoa(){  
        super();  
        System.out.println("Executando o construtor de Pessoa");  
    }  
    ...//outros métodos da classe  
}
```

Note que o construtor é similar a um método, mas ele não tem um tipo de retorno, nem *void*. Outro fato importante, um Construtor sempre terá o mesmo nome da classe. Modificadores de acesso podem ser atribuídos aos construtores, inclusive o *private*. Se o Construtor não for criado junto ao código-fonte da classe, o compilador criará automaticamente um construtor para sua Classe.

Quando uma classe é instanciada, o método *super()* dela é chamado, mesmo que não seja declarado, pois em algum momento a classe terá que ser inicializada. Lembre-se que todas as classes em Java são filhas da classe *Object*, ou seja, *Object* é a mãe de todas as classes (ela fornece métodos como *equals* e *toString* por exemplo).

Ainda no nosso exemplo apresentado na Figura 67, onde objetos do tipo Física são filhos da classe Pessoa, quando criamos uma instância de Física os construtores são executados em forma de pilha: Física chama Pessoa, que chama *Object*. Ou seja, o construtor de Física só será executado por último. Analise os códigos a seguir, e logo após a saída do compilador na Figura 73.

```
public class Fisica extends Pessoa{  
    private String cpf;  
    public Fisica(){  
        System.out.println("Pessoa Fisica");  
    }  
}
```

```
public class CaixaEletronico {  
    public static void main(String[] args) {  
        Fisica f = new Fisica();  
  
    }  
}
```

```
run:  
Executando o construtor de Pessoa  
Pessoa Fisica  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 73 – Saída do Compilador quando se instancia a classe Física

Note que ao realizar a instância da classe Física, o construtor da classe Pessoa também foi acionado.

Uma classe pode possuir mais de um construtor. Você deve estar se perguntando: mas como isso é possível? Para isso, é preciso criar construtores com argumentos diferentes, desta forma, criam-se diversas formas de inicializar um objeto. Como exemplo, vamos utilizar a classe Física, passando como parâmetro no momento da inicialização da classe o nome do titular da conta. Analise os códigos a seguir, e depois a saída do compilador na Figura 74.

```
public class Fisica extends Pessoa{  
    private String cpf;  
    public Fisica(){  
        System.out.println("Pessoa Fisica");  
    }  
    public Fisica(String nome){  
        nomePessoa = nome;  
    }  
}
```

```
public class CaixaEletronico {  
    public static void main(String[] args) {  
        Fisica f = new Fisica();  
        Fisica f2 = new Fisica("Joaquim");  
        System.out.println(f2.getNomePessoa());  
    }  
}
```

```
run:
Executando o construtor de Pessoa
Pessoa Física
Executando o construtor de Pessoa
Joaquim
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 74 – Saída do console

Não há limite para a criação de construtores, você pode criar em uma classe quantos construtores quiser, porém, o que deve diferenciar os construtores são os parâmetros, por exemplo, se um construtor requer uma única *String*, então não se pode criar outro construtor solicitando uma *String*, por mais que sejam variáveis diferentes, porém, se for um construtor solicitando uma *String* e outro solicitando duas *Strings*, é possível, ou seja, o tipo do parâmetro e o número de parâmetros serão os determinantes para que o compilador saiba qual construtor deva ser chamado.

Um ponto importante é que enquanto o construtor não for executado, nenhum acesso à variável de instância ou método será possível, isto quer dizer que um objeto não pode ser utilizado até ser inicializado, o que é óbvio, mas também significa que você não pode tentar sabotar o construtor do objeto antes de chamar `super()`, como pode ser visto no código a seguir:

```
public class Fisica extends Pessoa{
    private String cpf;
    public Fisica(){
        System.out.println("Pessoa Física");
    }
    public Fisica(String nome){
        nomePessoa = nome;
        super();
    }
}
```

A chamada de super() no construtor da Classe Física retorna um erro de compilação, onde será mostrado ao programador uma mensagem de que a chamada ao construtor de super() deve ser a primeira sentença no construtor.

CONSIDERAÇÕES FINAIS

Nesta Unidade você aprendeu sobre encapsulamento e como os modificadores de acesso auxiliam na tarefa de encapsular seus projetos. Também aprendeu a manipular modificadores privados por meio dos métodos públicos *getters* e *setters*.

Além dos modificadores de acesso, você aprendeu a utilizar o modificador final para criar constantes, métodos que não podem ser sobreescritos e classes que não podem ter filhos. Também viu o modo de funcionamento do modificador *static*, que compartilha o mesmo espaço de memória para atributos deste tipo e como os métodos estáticos são alocados em memória antes da instância da classe e podem ser chamados utilizando somente o nome da classe e o nome do método.

Não menos importante, você também aprendeu sobre o modificador *abstract*, que faz com que a classe não possa ser instanciada e os métodos abstratos apenas indicam o que deve ser implementado nas classes filhas. E, por fim, viu também como funcionam os Construtores em JAVA, assim como a forma de inicializar seus objetos utilizando estes construtores.

Na próxima Unidade, você verá os conceitos que envolvem Herança, que são características dos projetos Orientados a Objetos.

ATIVIDADES



1. Crie o projeto com as classes ilustradas na Figura 68 separando os pacotes, assim como criando todos os métodos públicos para acesso aos atributos *private* e *protected*.
2. Considerando que a Classe Jurídico é filha da classe Pessoa (Jurídico herda Pessoa), é possível acessar o atributo *protected nomePessoa* de forma direta? Por quê? Se o atributo fosse do tipo *private*, seria possível o acesso de forma direta?
3. Crie 3 atributos na classe Conta: limiteTransferencia, limiteSaque e taxaJuros, sendo que o limite de transferência é de R\$ 3.000,00, o limite de saque é de R\$ 1.000,00 e o limite de transferência é de R\$ 2.500,00, e atribua os modificadores de modo que estes atributos possam ser acessados e compartilhados por todas as instâncias de conta, assim como não permita a mudança destes atributos em tempo de execução.
4. Dadas as classes Pessoa, PFisica e Vendedor logo abaixo:

```
class Pessoa {  
    Pessoa(){  
        System.out.println("Pessoa Construct");  
    }  
}
```

```
class PFisica extends Pessoa{  
    PFisica(){  
        System.out.println("Pessoa Fisica");  
    }  
}
```

ATIVIDADES



```
public class Vendedor extends PFisica{  
  
    Vendedor() {  
        System.out.println("Vendedor");  
    }  
  
    public static void main(String[] args) {  
        Vendedor v = new Vendedor();  
    }  
}
```

Responda: como será a saída do console quando este projeto for executado?

MATERIAL COMPLEMENTAR



NA WEB

Modificadores Java

Por <javafreee.org>

Como em todas as linguagens de programação, a acessibilidade a uma classe/método deve seguir algumas regras, e é de extrema necessidade que você saiba para não vacilar na hora do exame! Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://javafree.uol.com.br/artigo/6941/Cap-2-Modificadores.html>>.



NA WEB

Modificadores de acesso

Disponível em: <<http://www.youtube.com/watch?v=OEGThN7PmsA>>.

Encapsulamento

Disponível em: <<http://www.youtube.com/watch?v=2SHpfjfxeQ8>>.

Constantes e o Modificador final

Disponível em: <<http://www.youtube.com/watch?v=65mDDzCsDaU>>.



REFLITA

Modificadores são fundamentais para a criação de frameworks e componentes caixa-preta e caixa-branca!



HERANÇA E POLIMORFISMO EM JAVA

UNIDADE

V

Objetivos de Aprendizagem

- Entender o conceito de herança.
- Entender o conceito de Polimorfismo.

Plano de Estudo

A seguir, apresentam-se os tópicos que você estudará nesta unidade:

- O que é herança
- Como implementar herança
- O que é polimorfismo
- A sobrescrita de métodos

INTRODUÇÃO

Olá, seja bem-vindo(a) ao módulo de Herança e Polimorfismo em Java. Nesta Unidade, trataremos de um tema essencial para a Orientação de Objetos, pois por meio de herança podemos otimizar a reutilização de códigos, uma vez que classes com características semelhantes podem ser implementadas de forma agrupada por meio de superclasses e subclasses (classes filhas).

Utilizando o polimorfismo, é possível criar métodos de nomes iguais, porém de comportamentos distintos de acordo com a origem do objeto. Para mostrar a você como utilizar estes recursos, serão utilizados alguns exemplos que utilizam estas técnicas de Herança e Polimorfismo, sendo um exemplo discutido durante a apresentação do tema e outro mostrando um projeto completo desenvolvido com técnicas de Herança e Polimorfismo.

HERANÇA

Vamos começar nossa Unidade mostrando uma perspectiva do mundo real. Imagine um peixe, como na Figura 75 a seguir. Qualquer espécie animal possui características próprias que definem com precisão em qual espécie o animal se enquadra. Vamos pegar como exemplo os peixes, eles possuem brânquias (para respirar), barbatana/nadadeira para se locomover e boca para se alimentar.

De uma maneira geral, todos os peixes possuem estas características, agora, se formos aprofundar um pouco mais no mundo dos peixes e analisarmos as características de cada espécie, por exemplo: tipo de escama, tamanho, coloração, número de dentes, se é de água doce ou salgada, entre outras características, teremos uma infinidade de espécies que se diferem, como o peixe-palhaço que possui características específicas a raça dele, por exemplo, o tamanho reduzido, a coloração alaranjada e as manchas brancas e pretas distribuídas em seu corpo.

Um filhote de peixe-palhaço possui as mesmas características de seus pais, avós, bisavós e outras gerações, ou seja, ele herda todas as características da raça, além disso, ele possui todas as características comuns a qualquer peixe.

Mantendo o raciocínio dos peixes, podemos dizer que um peixe-palhaço é uma raça da espécie peixe, se fôssemos transformar este conceito na programação Orientada a Objetos, basta transformar o peixe-palhaço em uma classe de nome PeixePalhaco, e esta herdaria as características dos Peixes, representada pela classe Peixe.

Desta forma, podemos afirmar que o PeixePalhaco é uma subclasse da superclasse Peixe. Em outras palavras, objetos do tipo PeixePalhaco terão todas as características de objetos do tipo Peixe, porém terão algumas especificidades que os diferem de outros tipos de peixe. Veja na Figura 76, logo a seguir, o esquema de herança que ilustra o caso em tela.



Figura 75 – Peixe-palhaço

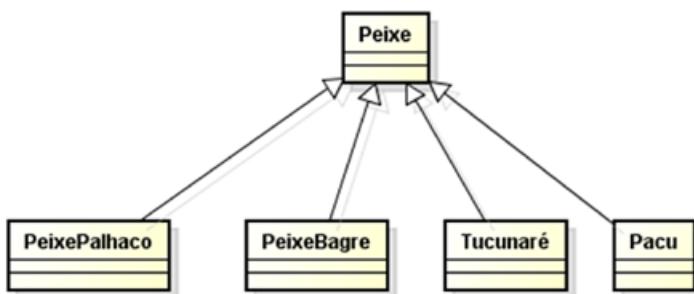


Figura 76 – Superclasse Peixe e respectivas subclases

Fonte: o autor

Todas as raças de peixes ilustradas na Figura 76 herdam todos os atributos e métodos da classe Peixe, ou seja, herdam suas características, e é esse o papel da Herança na Orientação a Objetos. De maneira geral, a herança permite a criação de novas classes (subclasses) a partir de classes já existentes (superclasses), “herdando” características existentes na classe a ser estendida. Esta técnica implica em grande reaproveitamento de código existente, uma vez que não há a necessidade de reimplementação de métodos que já foram criados nas superclasses.

Em relação a Superclasses e Subclasses, podemos dizer que Subclasses são especializações de Superclasses que, por sua vez, são generalizações de Subclasses. Em outras palavras, subclasses são mais especializadas do que as suas superclasses, mais genéricas. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos.

A linguagem Java permite o uso de herança simples, mas não permite a implementação de herança múltipla, que significa que uma classe pode herdar métodos e atributos de mais de uma classe. Para superar essa limitação, o Java faz uso de interfaces, o qual será descrito mais adiante ainda nesta Unidade.

Em Java, a palavra reservada que define que uma classe herda as características de outra é *extends*, ela deve ser utilizada assim que a classe for criada. Veja nos códigos a seguir um exemplo de código que mostra onde deve ser empregada a palavra *extends*.

```
public class Peixe {  
    //atributos  
    private String tipoPele;  
    private int numDentes;  
    //metodos  
    public void nadar(){  
        System.out.println("Mecher as barbatanas");  
    }  
    public void comer(){  
        System.out.println("Procurar comida e comer");  
    }  
}
```

```
public class PeixePalhaco extends Peixe{  
    //implementação da classe peixe palhaco  
    private int numeroListras;  
}
```

Da forma como foram implementados esses códigos, a classe PeixePalhaco herda todos os atributos e métodos da classe Peixe, mas com a diferença de possuir um atributo a mais, “numeroListras”, que é uma especificidade do peixe-palhaço em relação a outros tipos de peixe.

Ainda utilizando dois últimos códigos, vamos criar agora uma classe principal que cria uma instância de peixe-palhaço e chama um de seus métodos.

```
public class Principal
public static void main(String args[]){
    PeixePalhaco pp = new PeixePalhaco();
    pp.nadar();
    PP.comer();
}
```

O código, a seguir, produzirá no compilador uma saída como segue:

```
Mexer barbatanas
Procurar comida e comer
```

Note que os métodos nadar e comer não foram implementados na classe PeixePalhaco, somente na classe Peixe, como PeixePalhaco é uma Subclasse de Peixe, então os métodos da classe Peixe serão herdados automaticamente a objetos do tipo PeixePalhaco.

Agora que você já entendeu o conceito base para herança, vamos retomar o projeto do Caixa Eletrônico que iniciamos a desenvolver na Unidade anterior. Analise a Figura 77 logo abaixo, que ilustra o projeto como um todo a partir de um diagrama de classes:

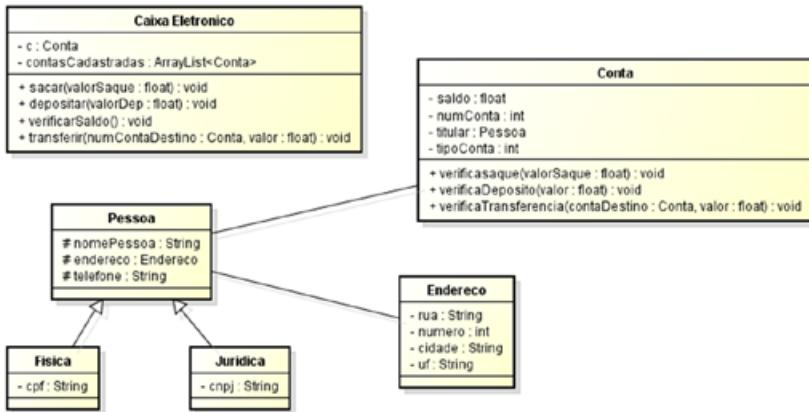


Figura 77 – Projeto de Caixa Eletrônico

Fonte: o autor

Tendo como base o problema de que em um banco existem dois tipos de pessoas, sendo elas: Pessoa Física e Pessoa Jurídica, podemos projetar nossos objetos como a classe Pessoa sendo a superclasse e as classes Física e Jurídica como sendo subclasses. Além disso, podemos criar a classe Pessoa como sendo abstrata, já que “Pessoa” não se refere a ninguém especificamente, só de forma genérica. Sendo assim, teremos os seguintes códigos para cada classe:

```

package cliente;

import java.util.Scanner;

public abstract class Pessoa {
    protected String telefone;
    protected String nomePessoa;
    protected Endereco e = new Endereco();
    //Métodos
    public void cadastra() {...}
    //Getters e Setters
    public Endereco getE() {...}
    public void setE(Endereco e) {...}
    public String getNomePessoa() {...}
    public void setNomePessoa(String nomePessoa) {...}
    public String getTelefone() {...}
    public void setTelefone(String telefone) {...}
}

```

```
package cliente;

import java.util.Scanner;

public class Fisica extends Pessoa{

    private String cpf;

    //sobrescrita do metodo cadastrada
    @Override
    public void cadastrada() {
        System.out.println("----Cadastro de Pessoa Fisica");
        super.cadastrada();
        Scanner tec = new Scanner(System.in);
        System.out.println("Digite o cpf");
        cpf = tec.nextLine();
    }

}

package cliente;

import java.util.Scanner;

public class Juridica extends Pessoa{

    private String cnpj;

    @Override
    public void cadastrada() {
        System.out.println("----Cadastro de Pessoa Juridica----");
        super.cadastrada();
        Scanner tec = new Scanner(System.in);
        System.out.println("Digite o cnpj");
        cnpj = tec.nextLine();
    }

}
```

Note que no momento de declaração das classes Física e Jurídica existe a palavra reservada *extends*, e é esta palavra que indica se uma classe é ou não subclasse de outra. Neste momento, todos os atributos e métodos implementados na classe Pessoa vão automaticamente fazer parte das classes Filhas. Lembrando um pouco sobre a teoria dos modificadores, temos que os atributos da classe Pessoa estão indicados com o modificador de acesso *protected*, e isso significa que as classes filhas de Pessoa terão acesso a estes atributos de forma direta.

Volte no código da classe Pessoa e confira quais são os atributos e métodos dessa classe, feito isso, você já sabe quais são os atributos que a classe Física possui: telefone, nomePessoa, endereço e CPF. Assim como os atributos da classe Jurídica: telefone, nomePessoa, endereço e CNPJ. Além dos atributos, as classes também possuem os métodos implementados na classe Pessoa, o código a seguir mostra quais são os métodos e atributos visíveis de dentro da classe Física.

Agora preste atenção ao código a seguir, quando precisamos “chamar” algum método que faz parte da superclasse Pessoa. Note que é necessário utilizar a palavra reservada super, que nos dá acesso aos métodos e atributos da classe pai. Além disso, quando criarmos uma instância de Física, podemos também ter acesso a todos os métodos da classe Pessoa, como pode ser visto no código a seguir (note que os métodos *getters* e *setters* da classe Pessoa tem visibilidade por meio da classe Física, que também é uma classe Pessoa).

```
public class Fisica extends Pessoa{

    private String cpf;
    //sobrescrever
    @Override
    public void cadastra()
        System.out.println("Cadastrado com sucesso!");
    }
}

    
```

Método	Tipo de Retorno
clone()	Object
equals(Object obj)	boolean
finalize()	void
getClass()	Class<?>
hashCode()	int
notify()	void
notifyAll()	void
setE(Endereco e)	void
setNomePessoa(String nomePessoa)	void

```
public class CaixaEletronico {
    public static void main(String[] args) {
        Fisica f = new Fisica();
        f.●cadстра() void
        Sy ●equals(Object obj) boolean
            ●getClass() Class<?>
        }
    }
} ●getE() Endereco
    ●getNomePessoa() String
    ●getTelefone() String
    ●hashCode() int
    ●notify() void
    ●notifyAll() void
    ●setE(Endereco e) void
    ●setNomePessoa(String nomePe... void
```

O mecanismo de herança nos fornece um benefício notável no desenvolvimento de aplicações. Ao concentrarmos características comuns em uma classe e derivar as classes mais específicas a partir desta, nós estamos preparados para a adição de novas funcionalidades ao sistema. Se mais adiante uma nova propriedade comum tiver que ser adicionada, não precisaremos efetuar alterações em todas as classes. Basta alterar a superclasse e todas as outras classes derivadas serão automaticamente atualizadas.

POLIMORFISMO

Polimorfismo significa várias (poli) formas (morfo). Em Orientação a Objetos, polimorfismo é a capacidade pela qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação, assinatura (o mesmo nome de método), mas que possuem comportamentos distintos (de acordo com a forma de implementação em cada subclasse). Em Java, o conceito de Polimorfismo se manifesta apenas nas chamadas dos métodos. A possibilidade de Polimorfismo se dá pelo fato de que métodos podem ser

sobrescritos pelas subclasses (métodos com o mesmo nome e números de argumentos), ou seja, se o método da superclasse não é suficiente ou não se aplica à classe filha, ele pode ser escrito novamente tendo um comportamento completamente diferente do da superclasse.

O interpretador JAVA se encarrega de chamar corretamente o método a ser executado em tempo de execução. Existe ainda um mecanismo de sobrecarga, onde dois métodos de uma classe podem ter o mesmo nome, porém com assinaturas diferentes (tipos de retorno ou tipos de argumentos diferentes), entretanto, esta sobrecarga não recebe o nome de polimorfismo.

Como dito anteriormente, tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura para o método correto. Em Java, todas as determinações de métodos a executar ocorrem por meio da ligação tardia (ocorrência em tempo de execução) exceto em dois casos: métodos *final*, que não podem ser redefinidos, e métodos *private*, que também não podem ser redefinidos e, portanto, possuem as mesmas características de métodos *final*.

Para que seja implementado o polimorfismo de maneira correta, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos, que é o mesmo que sobrescrita (*Override*) de métodos em classes derivadas. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, um novo corpo em uma classe derivada. É importante observar que, quando o polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não intuitivo.

Para ilustrar todo este processo, vamos verificar o método `cadastra`, que está presente nas três classes do nosso exemplo (Pessoa, Física e Jurídica) e acrescentar a implementação do método `imprime` nestas classes. O código, a seguir, indica a classe abstrata Pessoa, que possui atributos protegidos (podem ser visualizados

por seus herdeiros) e métodos de cadastrA() e imprime() implementados.

```
public abstract class Pessoa {
    protected String telefone;
    protected String nomePessoa;
    protected Endereco e = new Endereco();
    public void cadastrA(){
        Scanner tec = new Scanner(System.in);
        System.out.println("Digite o nome");
        nomePessoa = tec.nextLine();
        System.out.println("Digite o telefone");
        telefone = tec.nextLine();
        e.cadastra();
    }
    public void imprime(){
        System.out.println("Nome: "+getNomePessoa());
    } //Getters e Setters
    public Endereco getE() {...}
    public void setE(Endereco e) {...}
    public String getNomePessoa() {...}
    public void setNomePessoa(String nomePessoa) {...}
    public String getTelefone() {...}
    public void setTelefone(String telefone) {...}
}
```

No código a seguir, temos a implementação da subclasse Física, que conta com a implementação do método cadastrA() e imprime(). Note que antes da assinatura dos métodos, existe uma anotação indicando que o método está sobrescrevendo outro. Caso haja necessidade de utilização do método da superclasse, é necessário invocá-lo como acontece na linha “super.cadastrA()”. Desta forma, o método da classe Pai também será executado.

```
public class Fisica extends Pessoa{  
  
    private String cpf;  
  
    //sobrescrita do metodo cadastrA  
    @Override  
    public void cadastrA() {  
        System.out.println("----Cadastro de Pessoa Fisica");  
        super.cadastra();  
        Scanner tec = new Scanner(System.in);  
        System.out.println("Digite o cpf");  
        cpf = tec.nextLine();  
    }  
    @Override  
    public void imprime(){  
        System.out.println("Pessoa Fisica!");  
        super.imprime();  
        System.out.println("CPF: "+cpf);  
    }  
}
```

A implementação da classe Jurídica é análoga à implementação da classe Física, o que difere são as mensagens que serão apresentadas ao usuário do aplicativo, e o atributo CNPJ existente somente na classe Jurídica. O código, a seguir, mostra a implementação da classe Jurídica.

```
public class Juridica extends Pessoa{  
  
    private String cnpj;  
  
    @Override  
    public void cadastrA() {  
        System.out.println("----Cadastro de Pessoa Juridica----");  
        super.cadastra();  
        Scanner tec = new Scanner(System.in);  
        System.out.println("Digite o cnpj");  
        cnpj = tec.nextLine();  
    }  
    @Override  
    public void imprime(){  
        System.out.println("Pessoa Juridica!");  
        super.imprime();  
        System.out.println("CNPJ: "+cnpj);  
    }  
}
```

Implementadas as classes filhas e os métodos sobrescritos, criou-se um método principal para manipular os objetos do tipo Pessoa, dando a opção para que o usuário do aplicativo selecione qual o tipo de Pessoa que se deseja cadastrar. Caso escolha a opção 1, será criado um objeto do tipo Física dentro do vetor de Pessoas, caso a opção escolhida seja o item 2, um objeto do tipo Jurídica será criado. Caso o usuário queira imprimir os elementos cadastrados até então, ele pode optar pela opção de número 3. Veja esta implementação no código a seguir.

Note que o método `cadastra` é chamado apenas uma vez, isto é possível, pois como as subclasses sobrescrevem um método da superclasse, este método pode ter comportamentos diferentes de acordo com o objeto de origem, e isto é o polimorfismo!

```
public class CaixaEletronico {
    static Pessoa p[] = new Pessoa[10];
    public static int ultimo = 0;
    public static void main(String[] args) {
        Scanner tec = new Scanner(System.in);
        int opcao = 0;
        while (opcao!=4){
            System.out.println("Digite: ");
            System.out.println("1 - Cadastrar Pessoa Física");
            System.out.println("2 - Cadastrar Pessoa Jurídica");
            System.out.println("3 - Imprimir clientes");
            System.out.println("4 - Sair");
            opcao = tec.nextInt();
            tec.nextLine();
            switch(opcao){
                case 1: p[ultimo] = new Fisica(); break;
                case 2: p[ultimo] = new Juridica();break;
                case 3: imprimeP();break;
                case 4: System.out.println("Sair...");break;
                default: System.out.println("Valor invalido"); break;
            }
            if(opcao == 1 || opcao==2){
                p[ultimo].cadastra();
                ultimo++;
            }
        }
    }
}
```

Outro exemplo de aplicação do polimorfismo ainda no exemplo em questão é com o método de impressão, como todas as classes filhas também sobrescrevem o método `imprime` da Superclasse, ele pode ser chamado, que se comportará de acordo com o objeto de origem. Mas e se o não houver sobrescrita do método?

Como o compilador se comporta? Nestes casos, o compilador invoca o método da classe Pai e executa-o. Veja no código, a seguir, a implementação do método de impressão. Note que não é necessário verificar que tipo de Pessoa precisa ser impressa, em tempo de execução isto é resolvido.

```
public static void imprimeP() {
    for (int i = 0; i <= ultimo; i++) {
        p[i].imprime();
    }
}
```

ESTUDO DE CASO – ANIMAIS

A seguir, será mostrado um projeto que implementa Herança e Polimorfismo, e desta forma ilustrar os conceitos aprendidos até aqui.

Os animais do zoológico da cidade de São Paulo precisam ser catalogados pela idade e nome e o zoológico precisa que, além disso, fossem catalogadas as formas de se locomover do animal, assim como o som que eles emitem, pois a ideia do zoológico é transformar este software em algo que possa mostrar para os visitantes algumas curiosidades sobre os animais que ali vivem. Para isso, a administração do zoológico chamou Oswaldo, um programador JAVA para desenvolver um software que auxiliasse nesta catalogação.

Oswaldo então começou a fazer a análise dos requisitos e percebeu que trataba-se de um caso em que a utilização de técnicas de polimorfismo e herança eram imprescindíveis para que o projeto pudesse ser expandido mais tarde, além de auxiliar no processo de manutenção. Desta forma, Oswaldo criou uma classe abstrata chamada Animal, e nela colocou as informações principais solicitadas pelo zoológico para a catalogação. Então ele criou as classes Cachorro, Cavalo e Preguiça, que herdam as propriedades de Animal e implementam os métodos abstratos. A seguir, serão apresentados os métodos desenvolvidos por Oswaldo para resolver esta necessidade do zoológico.

No código a seguir, a classe Animal foi criada como uma classe abstrata, pois “Animal” é uma abordagem geral e precisa ser especificada, esta classe possui dois atributos, nome e idade do animal, além dos métodos abstratos seLocomove() e emiteSom(), que deverão, obrigatoriamente, serem implementados nas classes que herdarem a classe Animal.

```
package animal;
abstract public class Animal {
    private String nome;
    private int idade;
    abstract public void seLocomove();
    abstract public void emiteSom();
    public int getIdade() {
        return idade;
    }
    public void setIdade(int idade) {
        this.idade = idade;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

A Classe Cachorro herda todos os atributos e métodos da classe Animal, e é obrigada a implementar os métodos seLocomove() e emiteSom(). Note ainda que a classe cachorro possui um Construtor que recebe como parâmetro o nome e a idade do Animal. O código, a seguir, ilustra a criação desta classe.

```
package animal;
public class Cachorro extends Animal{
    public Cachorro(String nome, int id ){
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Cachorro: "+super.getNome());
        System.out.println("Idade: "+super.getIdade());
        System.out.println("Cachorro correndo, com suas 4 patas");
    }
    // 
    @Override
    public void emiteSom() {
        System.out.println("Au Au !");
    }
}
```

A Classe Cavalo, assim como a Classe Cachorro, herda todos os atributos e métodos da classe Animal e é obrigada a implementar os métodos seLocomove() e emiteSom(), porém, a implementação difere da classe Cachorro, o que resulta em um comportamento diferente, isto é o polimorfismo dos métodos seLocomove() e emiteSom();.

```
package animal;
public class Cavalo extends Animal {
    public Cavalo(String nome, int id ){
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Cavalo: "+super.getNome());
        System.out.println("Idade: "+super.getIdade());
        System.out.println("Cavalga, Marcha, Trota");
    }
    // 
    @Override
    public void emiteSom() {
        System.out.println("Nhiiiiiiiiiiii ri ri rin !");
    }
}
```

No código, a seguir, temos a implementação da classe Preguiça, seguindo os padrões de implementação dos outros animais.

```
package animal;
public class Preguica extends Animal{
    public Preguica(String nome, int id ){
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Preguiça: "+super.getNome());
        System.out.println("Idade: "+super.getIdade());
        System.out.println("Subindo na Árvore");
    }
    @Override
    public void emiteSom() {
        System.out.println("GRRRRRRRrrrrrrr");
    }
}
```

A seguir, temos a implementação da classe principal que cria e manipula o vetor de animais. Note que no código de criação do vetor, criam-se ponteiros para cada Animal, mas ainda não se sabe qual animal será colocado em cada posição. Quem vai determinar o tipo do animal é o usuário, de acordo com a ordem de inserção por meio do método `cadastraAnimal()`.

Este método questiona ao usuário qual o tipo de animal será armazenado, sendo que o método fica encarregado de criar o objeto do animal desejado. Depois de cadastrado, é possível verificar que animal ocupa cada posição do vetor por meio do método `imprimeAnimais()`. Veja que cada objeto sabe exatamente como deve ser seu comportamento ao chamar os métodos de `seLocomove()` e `emiteSom()`.

```
import java.util.Scanner;
public class AnimalPrincipal {
    Animal vetAni[] = new Animal[30];
    static int tam=0;
    public static void main(String[] args) {
        // TODO code application logic here
        int opcao=0;
        Scanner scan = new Scanner(System.in);
        AnimalPrincipal ap = new AnimalPrincipal();
        while(opcao!=3){
            System.out.println("Digite a opção desejada:");
            System.out.println("1 - Cadastrar");
            System.out.println("2 - Listar");
            System.out.println("3 - Sair");
            opcao = scan.nextInt();
            scan.nextLine();
            switch(opcao){
                case 1: ap.cadastraAnimal();
                break;
                case 2: ap.imprimeAnimais();
                break;
                case 3: System.out.println("Saindo...");
                break;
                default: System.out.println("Opção inválida");
                break;
            }
        }
    }
    public void cadastrAAnimal(){
        Scanner scan = new Scanner(System.in);
        System.out.println("Digite o tipo de animal:");
        System.out.println("1 - Cachorro");
        System.out.println("2 - Cavalo");
        System.out.println("3 - Preguiça");
        int i = scan.nextInt();
        scan.nextLine();
    }
}
```

```
if (i==1 || i==3 || i==2){  
    System.out.println("Digite o nome do animal");  
    String n = scan.nextLine();  
    System.out.println("Digite a idade do animal");  
    int id = scan.nextInt();  
    scan.nextLine();  
    if (i==1)  
        vetAni[tam]= new Cachorro(n,id);  
    else if(i==2)  
        vetAni[tam]= new Cavalo(n,id);  
    else if(i==3)  
        vetAni[tam]= new Preguica(n,id);  
    tam++;  
}  
public void imprimeAnimais(){  
    Scanner scan = new Scanner(System.in);  
    for (int i=0;i<tam;i++){  
        System.out.println("Codigo do Animal: "+i);  
        vetAni[i].seLocomove();  
    }  
    System.out.println("Digite o codigo do animal que deseja  
ver");  
    int cod = scan.nextInt();  
    vetAni[cod].seLocomove();  
    vetAni[cod].emiteSom();  
}  
}
```

Execute o projeto Animal e teste você mesmo as funcionalidades das técnicas de Herança e Polimorfismo aqui apresentadas.

CONSIDERAÇÕES FINAIS

Nesta Unidade, você estudou conceitos muito importantes para a criação de projetos Orientados a Objetos. Herança e Polimorfismos ajudam e muito no desenvolvimento de aplicações que são transformadas do mundo real para o mundo computacional, e estes mecanismos possibilitam maior flexibilidade do projeto, aumentando assim a reutilização de códigos.

Por meio de Herança, você viu que é possível herdar métodos e atributos de classes pai, e que o Java consegue identificar qual é o tipo de objeto que está sendo trabalhado. Com o Polimorfismo, você pôde perceber a capacidade de sobrescrita do método, além de entender como um método implementado em uma superclasse e sobreescrito nas subclasses pode agilizar o desenvolvimento, uma vez que não se faz necessária a criação de mecanismos de identificação e desvios para “chamar” o método correto.

Você também viu os conceitos de Interfaces, que nada mais são do que um contrato entre a Interface a classe que se “compromete” a implementar todos os métodos abstratos identificados na interface, além disso, você também pôde ver como o JAVA pode simular a herança múltipla utilizando interfaces.

ATIVIDADES



1. No projeto de animais, crie as classes Gato, Galinha e Avestruz, e crie os métodos necessários para cada Animal.

MATERIAL COMPLEMENTAR



NA WEB

Entendendo e aplicando herança em Java

Por Hudson Geovane

A herança é um princípio da POO que permite a criação de novas classes a partir de outras previamente criadas. Essas novas classes são chamadas de subclasses, ou classes derivadas; e as classes já existentes, que deram origem às subclasses, são chamadas de superclasses, ou classes base. Deste modo, é possível criar uma hierarquia dessas classes, tornando, assim, classes mais amplas e classes mais específicas. Uma subclass herda métodos e atributos de sua superclasse; apesar disso, pode escrevê-los novamente para uma forma mais específica de representar o comportamento do método herdado.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://www.devmedia.com.br/entendendo-e-aplicando-heranca-em-java/24544>>. Acesso em: 15 maio 2013.



NA WEB

Herança em Java

Disponível em: <<http://www.youtube.com/watch?v=l9cGMw4QORQ>>.



NA WEB

O uso de polimorfismo em Java

Por Higor Medeiros

Veja neste artigo os conceitos, utilização, implementação e a importância do polimorfismo.

Também veremos onde o polimorfismo é utilizado e de que forma ele pode ajudar para termos projetos melhores.

Saiba mais sobre o assunto no artigo completo.

Disponível em: <<http://www.devmedia.com.br/uso-de-polimorfismo-em-java/26140>>.



NA WEB

Polimorfismo com classes abstratas

Disponível em: <<http://www.youtube.com/watch?v=FH0a-L4OQB4>>.



REFLITA

Toda aplicação Java deve envolver a organização de suas classes de dados de forma polimórfica!



CONCLUSÃO

Neste livro, busquei mostrar a você uma introdução à programação com a linguagem Java. A fim de possibilitar o seu entendimento, a Unidade I apresentou o histórico da linguagem Java, o processo de compilação e execução e a máquina virtual Java.

Na Unidade II, trabalhamos estruturas básicas da linguagem Java, por exemplo, tipos primitivos, declarações e inicializações de variáveis e estruturas importantes ao fluxo de controle e de dados de um programa Java. Você está lembrado de quais são estas estruturas? As estruturas são: de controle, de repetição e de seleção.

A Unidade III foi dedicada exclusivamente ao entendimento da diferença entre classes e objetos Java e como uma classe é composta. Mostrei os principais elementos de uma classe: atributos, métodos e construtores, bem como o conceito de JavaBeans e POJO (*Plain Old Java Object*).

Na Unidade IV apresentei os principais modificadores de acesso da linguagem Java, sendo eles: *private*, *public*, *protected* e *default*. Apresentei também os conceitos essenciais sobre encapsulamento, que foram complementados com atividades de autestudo.

E, para finalizar, vimos na Unidade V o conceito de herança, sendo um dos conceitos mais importantes em programação orientada a objetos. Além disso, mostrei como acessar os membros de uma classe (atributos e métodos) por meio de herança e instanciação de objetos.

Espero ter alcançado o objetivo inicial deste livro, que era apresentar uma introdução à programação em Java. Desejo que você seja muito feliz profissionalmente utilizando os conceitos apresentados aqui, e se puder ajudar de alguma forma, estou à sua disposição.

Prof. Dr. Edson A. Oliveira Junior



GABARITO

GABARITO DAS ATIVIDADES DE AUTOESTUDO

UNIDADE I

1. Porque uma vez escritos e compilados em *bytecodes*, só é necessária uma máquina virtual Java para um sistema operacional específico, ficando o programa independente de plataforma.
2. As principais características estão relacionadas à sintaxe da linguagem, com as seguintes melhorias: *for enhanced*, *generics* e *autoboxing*.
3. JCP é a *Java Community Process*. Para submeter uma mudança, uma pessoa deve propor uma JSR (*Java Specification Release*) e enviar à JCP.
4. Java SE para o desenvolvimento de sistemas desktop e distribuídos; Java EE para aplicações corporativas Web e de componentes de negócio; Java ME para o desenvolvimento de aplicações para dispositivos móveis.
5. *Bytecode* é o código gerado pelo compilador Java. Ele é interpretado por meio das instruções presentes na máquina virtual Java.
6. Após escrito o código .java, ele é compilado por meio do compilador Java gerando o *bytecode*. O *bytecode* é então carregado e interpretado pela máquina virtual Java que executa suas instruções.
7. O *classpath* indica o local exato onde estão disponíveis todas as classes necessárias para que um programa seja compilado e executado.

UNIDADE II

1.
 - a) 2147483647
 - b) -2147483648
 - c) -2147483645
 - d) 2147483647
 - e) -2
 - f) -4
2.
 - a) Não existe “then” em Java.
 - b) *If* exige o uso de parênteses na expressão.
 - c) Nada errado.
 - d) Falta um “;” antes do *else*.



GABARITO

3.

a)

```
int idade = 10;
if(idade > 0 && idade <= 12) {
    System.out.println("criança...");
} else if(idade > 12 && idade <= 17) {
    System.out.println("adolescente");
} else if(idade >= 18 && idade <= 65) {
    System.out.println("adulto");
} else if(idade > 66) {
    System.out.println("idoso");
}
```

b)

```
int mes = 5;
int dias = 0;

switch(mes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        dias = 31;
    break;
    case 4:
    case 6:
    case 9:
    case 11:
        dias = 30;
    break;
    case 2:
        dias = 28;
    break;
}
```



GABARITO

```
    default:  
        dias = 0;  
    }  
;  
default:  
    dias = 0;  
}
```

c)

```
for(int i = 0; i < 100; i++) {  
    System.out.println("i: " + i);  
}
```

d)

```
int z = 10;  
while(z < 5) {  
    System.out.println("z: " + z);  
    z--;  
}
```



GABARITO

UNIDADE III

1.

```
// classe Conta...
public class Conta {
    double saldo;
    Conta() {
        saldo = 0.0;
    }

    public void depositar(double quantia) {
        saldo = saldo + quantia;
    }

    public void sacar(double quantia) {
        if(saldo >= quantia) {
            saldo = saldo - quantia;
        }
    }
}
//conta poupança
public class Inicio {

    public static void main(String[] args) {

        Conta contaCorrente = new Conta();
        Conta poupanca = new Conta();

        contaCorrente.depositar(1000.00);
        contaCorrente.sacar(500.00);

        System.out.println("Saldo da conta corrente: " + contaCorrente.
saldo);

        poupanca.depositar(2500.00);
        System.out.println("Saldo da poupanca: " + poupanca.saldo);
        poupanca.sacar(3000.00);
        System.out.println("Saldo da poupanca: " + poupanca.saldo);
    }
}
```



GABARITO

UNIDADE IV

1.

```
package localizacao;
public class Endereco {
    private String rua;
    private int numero;
    private String cidade;
    private String uf;

    public Endereco() { super(); }
    public String getCidade() { return cidade; }
    public void setCidade(String cidade) { this.cidade = cidade; }
    public int getNumero() { return numero; }
    public void setNumero(int numero) { this.numero = numero; }
    public String getRua() { return rua; }
    public void setRua(String rua) { this.rua = rua; }
    public String getUf() { return uf; }
    public void setUf(String uf) { this.uf = uf; }
}
```

```
package pessoal;
import localizacao.Endereco;
public class Pessoa {

    protected String nomePessoa;
    protected Endereco endereco;
    protected String telefone;

    public Pessoa() { super(); }
    public Endereco getEndereco() { return endereco; }
    public void setEndereco(Endereco endereco) { this.endereco =
        endereco; }
    public String getNomePessoa() { return nomePessoa; }
    public void setNomePessoa(String nomePessoa) { this.nomePessoa =
        nomePessoa; }
    public String getTelefone() { return telefone; }
    public void setTelefone(String telefone) { this.telefone =
        telefone; }
}
```



GABARITO

```
package pessoal;

public class Fisica extends Pessoa {

    private String cpf;

    public Fisica() { super(); }

    public String getCpf() { return cpf; }

    public void setCpf(String cpf) { this.cpf = cpf; }
}
```

```
package pessoal;
public class Juridica extends Pessoa {

    private String cnpj;

    public Juridica() { super(); }
    public String getCnpj() { return cnpj; }
    public void setCnpj(String cnpj) { this.cnpj = cnpj; }
}
```



GABARITO

```

package caixa;
import pessoal.Pessoa;
public class Conta {
    private double saldo;
    private int numConta;
    private Pessoa titular;
    private int tipoConta;

    public Conta() { super(); }
    public int getNumConta() { return numConta; }
    public void setNumConta(int numConta) { this.numConta = numConta;
    }
    public double getSaldo() { return saldo; }
    public void setSaldo(double saldo) { this.saldo = saldo; }
    public int getTipoConta() { return tipoConta; }
    public void setTipoConta(int tipoConta) { this.tipoConta =
        tipoConta; }
    public Pessoa getTitular() { return titular; }
    public void setTitular(Pessoa titular) { this.titular = titular;
    }
}

```

```

package caixa;
import java.util.ArrayList;
public class CaixaEletronico {

    private Conta c;
    private ArrayList contasCadastradas = new ArrayList<Conta>();

    public void sacar(float valorSaque) {}
    public void depositar(float valorDep) {}
    public void verificarSaldo(){}
    public void transferir(int numContaDestino, float valor) {}
}

```

2. Por herança, o atributo *protected* nomePessoa da classe Pessoa pode ser acessado por qualquer classe que estende tal classe independentemente em qual pacote esteja. Se o atributo for *private*, este não pode ser acessado de forma direta a partir da classe Jurídica.



GABARITO

3. Acrescente os seguintes atributos à classe Conta:

```
public static final double limiteTransferencia = 3000.0;
public static final double limiteSaque = 1000.0;
public static final double taxaJuros = 1.5;
```

4.

```
Pessoa Construct
Pessoa Fisica
Vendedor
```

A saída será:

UNIDADE V

1.

```
package animal;
public class Gato extends Animal {
    public Gato(String nome, int id) {
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Gato: " + super.getNome());
        System.out.println("Idade: " + super.getIdade());
        System.out.println("Gato correndo, com suas 4 patas");
    }
    @Override
    public void emiteSom() {
        System.out.println("Miauuuu... !");
    }
}
```



GABARITO

```
package animal;
public class Galinha extends Animal {
    public Galinha(String nome, int id) {
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Galinha: " + super.getNome());
        System.out.println("Idade: " + super.getIdade());
        System.out.println("Galinha correndo, com suas 2 patas");
    }
    @Override
    public void emiteSom() {
        System.out.println("Cocoricó...!");
    }
}
```

```
package animal;
public class Avestruz extends Animal {
    public Avestruz(String nome, int id) {
        super.setNome(nome);
        super.setIdade(id);
    }
    @Override
    public void seLocomove() {
        System.out.println("Avestruz: " + super.getNome());
        System.out.println("Idade: " + super.getIdade());
        System.out.println("Avestruz correndo, com suas 2 patas");
    }
    @Override
    public void emiteSom() {
        System.out.println("Ruuuu... !");
    }
}
```



REFERÊNCIAS

DEITEL, Harvey M. **Java - Como Programar.** 8. ed. Prentice Hall Brasil, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java.** V.1 – Fundamentos. 8. ed. Pearson, 2010.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java.** V.2 - Advanced Features. Prentice Hall, 2008.

SIERRA, Kathy; BATES, Bert. **Use A Cabeça! – Java.** 2. ed. Alta Books, 2008.

