

Universidade do Sul de Santa Catarina

Programação Orientada a Objeto

Disciplina na modalidade a distância

4ª edição revista e atualizada

Palhoça

UnisulVirtual

2007

Créditos

Unisul - Universidade do Sul de Santa Catarina

UnisulVirtual - Educação Superior a Distância

Campus UnisulVirtual

Avenida dos Lagos, 41
Cidade Universitária Pedra Branca
Palhoça – SC - 88137-100
Fone/fax: (48) 3279-1242 e
3279-1271
E-mail: cursovirtual@unisul.br
Site: www.virtual.unisul.br

Reitor Unisul

Gerson Luiz Joner da Silveira

Vice-Reitor e Pró-Reitor Acadêmico

Sebastião Salésio Heerdt

Chefe de Gabinete da Reitoria

Fabian Martins de Castro

Pró-Reitor Administrativo

Marcus Vinícius Anátoles da Silva
Ferreira

Campus Sul

Diretor: Valter Alves Schmitz Neto
Diretora adjunta: Alexandra
Orsoni

Campus Norte

Diretor: Ailton Nazareno Soares
Diretora adjunta: Cibele Schuelter

Campus UnisulVirtual

Diretor: João Vianney
Diretora adjunta: Jucimara
Roesler

Equipe UnisulVirtual

Administração

Renato André Luz
Valmir Venício Inácio

Avaliação Institucional

Dênia Falcão de Bittencourt

Biblioteca

Soraya Arruda Waltrick

Capacitação e Apoio Pedagógico à Tutoria

Angelita Marçal Flores
(Coordenadora)
Caroline Batista
Enzo de Oliveira Moreira
Patrícia Meneghel
Vanessa Francine Corrêa

Coordenação dos Cursos

Adriano Sérgio da Cunha
Aloísio José Rodrigues
Ana Luisa Mülbart
Ana Paula Reusing Pacheco
Charles Cesconetto
Diva Marília Flemming
Fabiano Ceretta
Itamar Pedro Bevilacqua
Janete Elza Felisbino
Jucimara Roesler
Lauro José Ballock
Lívia da Cruz (Auxiliar)
Luiz Guilherme Buchmann
Figueiredo
Luiz Otávio Botelho Lento
Marcelo Cavalcanti
Maria da Graça Poyer
Maria de Fátima Martins
(Auxiliar)
Mauro Faccioni Filho
Michelle D. Durieux Lopes Destri
Moacir Fogaça
Moacir Heerdt
Nélio Herzmann
Onei Tadeu Dutra
Patrícia Alberton
Raulino Jacó Brüning
Rodrigo Nunes Lunardelli
Simone Andréa de Castilho
(Auxiliar)

Criação e Reconhecimento de Cursos

Diane Dal Mago
Vanderlei Brasil

Desenho Educacional

Design Instrucional
Daniela Erani Monteiro Will
(Coordenadora)
Carmen Maria Cipriani Pandini
Carolina Hoeller da Silva Boeing
Flávia Lumi Matuzawa
Karla Leonora Dahse Nunes
Leandro Kingeski Pacheco
Ligia Maria Soufen Tumolo
Márcia Loch
Viviane Bastos
Viviani Poyer

Acessibilidade

Vanessa de Andrade Manoel

Avaliação da Aprendizagem

Márcia Loch (Coordenadora)
Cristina Klipp de Oliveira
Douglas Fabiani da Cruz
Silvana Denise Guimarães

Design Gráfico

Cristiano Neri Gonçalves Ribeiro
(Coordenador)
Adriana Ferreira dos Santos
Alex Sandro Xavier
Evandro Guedes Machado
Fernando Roberto Dias
Zimmermann
Higor Ghisi Luciano
Pedro Paulo Alves Teixeira
Rafael Pessi
Vilson Martins Filho

Disciplinas a Distância

Tade-Ane de Amorim
Cátia Melissa Rodrigues

Gerência Acadêmica

Patrícia Alberton

Gerência de Ensino

Ana Paula Reusing Pacheco

Logística de Encontros Presenciais

Márcia Luz de Oliveira
(Coordenadora)
Aracelli Araldi
Graciele Marinês Lindenmayr
Leticia Cristina Barbosa
Kênia Alexandra Costa Hermann
Priscila Santos Alves

Formatura e Eventos

Jackson Schuelter Wiggers

Logística de Materiais

Jeferson Cassiano Almeida da
Costa (Coordenador)
José Carlos Teixeira
Eduardo Kraus

Monitoria e Suporte

Rafael da Cunha Lara
(Coordenador)
Adriana Silveira
Andréia Drewes
Caroline Mendonça
Cristiano Dalazen
Dyego Rachadel
Edison Rodrigo Valim
Francielle Arruda
Gabriela Malinverni Barbieri
Jonatas Collaço de Souza
Josiane Conceição Leal
Maria Eugênia Ferreira Celeghin
Rachel Lopes C. Pinto
Vinícius Maykot Serafim

Produção Industrial e Suporte

Arthur Emmanuel F. Silveira
(Coordenador)
Francisco Asp

Relacionamento com o Mercado

Walter Félix Cardoso Júnior

Secretaria de Ensino a Distância

Karine Augusta Zanoni
Albuquerque
(Secretária de ensino)
Ana Paula Pereira
Andréa Lucí Mandira
Carla Cristina Sbardella
Deise Marcelo Antunes
Djeime Sammer Bortolotti
Franciele da Silva Bruchado
Grasiela Martins
James Marcel Silva Ribeiro
Jenniffer Camargo
Lamuniê Souza
Lauana de Lima Bezerra
Liana Pamplona
Marcelo José Soares
Marcos Alcides Medeiros Junior
Maria Isabel Aragon
Olavo Lajús
Priscilla Geovana Pagani
Rosângela Mara Siegel
Silvana Henrique Silva
Vanilda Liordina Heerdt
Vilmar Isaurino Vidal

Secretária Executiva

Viviane Schalata Martins

Tecnologia

Osmar de Oliveira Braz Júnior
(Coordenador)
Jefferson Amorin Oliveira
Marcelo Neri da Silva
Ricardo Alexandre Bianchini

Apresentação

Parabéns, você está recebendo o livro didático da disciplina de **Programação orientada a objeto**.

Este material didático foi construído especialmente para este curso, levando em consideração o seu perfil e as necessidades da sua formação. Como os materiais estarão, a cada nova versão, recebendo melhorias, pedimos que você encaminhe suas sugestões sempre que achar oportuno via professor tutor ou monitor.

Recomendamos, antes de você começar os seus estudos, que verifique as datas-chave e elabore o seu plano de estudo pessoal, garantindo assim a boa produtividade no curso.

Lembre: você não está só nos seus estudos, conte com o Sistema Tutorial da UnisulVirtual sempre que precisar de ajuda ou alguma orientação.

Desejamos que você tenha um excelente êxito neste curso!

Equipe UnisulVirtual

Andréa Bordin

Programação Orientada a Objeto

Livro didático

Design instrucional

Flavia Lumi Matuzawa

Karla Leonora Dahse Nunes

4ª edição revista e atualizada

Palhoça

UnisulVirtual

2007

Copyright © UnisulVirtual 2007

Nenhuma parte desta publicação pode ser reproduzida por qualquer meio sem a prévia autorização desta instituição.

Edição – Livro Didático

Professor Conteudista

Andréa Bordin

Design Instrucional

Karla Leonora Dahse Nunes, Flávia Lumi Matuzawa
Leandro Kingeski Pacheco (4. ed. rev. e atual.)

ISBN 978-85-7817-001-1

Projeto Gráfico e Capa

Equipe UnisulVirtual

Diagramação

Rafael Pessi

Revisão Ortográfica

B2B

005.117

B72 Bordin, Andréa

Programação orientada a objeto : livro didático / Andréa Bordin ; design instrucional Flavia Lumi Matuzawa, Karla Leonora Dahse Nunes, [Leandro Kingeski Pacheco]. – 4. ed. rev. e atual. – Palhoça : UnisulVirtual, 2007.
306 p. : il. ; 28 cm.

Inclui bibliografia.

ISBN 978-85-7817-001-1

1. Programação orientada a objetos (Computação). I. Nunes, Karla Leonora Dahse. II. Matuzawa, Flavia Lumi. III. Pacheco, Leandro Kingeski. IV. Título.

Ficha catalográfica elaborada pela Biblioteca Universitária da Unisu

Sumário

Apresentação	03
Palavras da professora	09
Plano de estudo	11
UNIDADE 1 – Conhecendo a Linguagem de Programação Java	15
UNIDADE 2 – Preparando o ambiente para programar em Java	33
UNIDADE 3 – Sintaxe básica da linguagem Java	49
UNIDADE 4 – Implementando os primeiros programas em Java	67
UNIDADE 5 – Implementando programas em Java com controle de fluxo	83
UNIDADE 6 – Desenvolvendo programas modularizados em Java	99
UNIDADE 7 – Introdução à Programação Orientada a Objeto (POO).....	111
UNIDADE 8 – Conceitos de Orientação a Objeto	137
UNIDADE 9 – Aplicando os conceitos de OO através de exemplos práticos	159
UNIDADE 10 – Modificadores	183
UNIDADE 11 – Objetos como atributos de outros objetos	205
UNIDADE 12 – Associação na prática	221
UNIDADE 13 – Herança	237
UNIDADE 14 – Herança na prática	259
Para concluir o estudo	275
Referências	277
Sobre a professora conteudista.....	279
Respostas e comentários das atividades de auto-avaliação	281

Palavras da professora



Olá!

Você está iniciando a disciplina de Programação Orientada a Objetos. Nessa disciplina, primeiramente, será apresentada a você uma linguagem de programação. O nome da linguagem é Java. Através dessa linguagem de programação, você poderá desenvolver, no computador, os algoritmos que elaborou nas disciplinas de Lógica de Programação I e II. Você conseguirá ver seus programas funcionando, poderá testá-los e modificá-los, já que terá o conhecimento de lógica, e, também, de uma linguagem de programação.

Para isso, as unidades iniciais desse livro serão dedicadas a passar todos os detalhes de sintaxe dessa linguagem. Será necessária muita atenção da sua parte, pois o desenvolvimento de instruções de um programa em uma linguagem de programação é uma tarefa muito detalhada. Seu programa pode não funcionar porque você esqueceu de programar, por exemplo, um ponto e vírgula.

Mas, essa atenção será recompensada com a sua satisfação ao ver o seu programa funcionar. É uma sensação muito boa ver o primeiro programa em funcionamento.

Conhecer uma linguagem de programação é um pré-requisito importante para que você possa aprender e colocar em prática os conceitos da programação orientada a objetos, que é o tópico principal desse livro e será tratado na maior parte das unidades.

A programação orientada a objetos tem esse nome porque o seu objetivo é resolver um problema analisando como ele funciona no mundo real, ou seja, quais os objetos que atuam no problema, quais as características (atributos) e comportamentos desses objetos e como eles se relacionam.

É muito importante que você entenda os conceitos do paradigma orientado a objetos, pois, a maioria dos sistemas, atualmente, é desenvolvida dentro dele.

Você não estará sozinho nessa caminhada, o livro e o professor tutor irão sanar suas dúvidas.

Boa Sorte!!



Plano de estudo

O plano de estudos visa a orientá-lo/a no desenvolvimento da Disciplina. Nele, você encontrará elementos que esclarecerão o contexto da Disciplina e sugerirão formas de organizar o seu tempo de estudos.

O processo de ensino e aprendizagem na UnisulVirtual leva em conta instrumentos que se articulam e se complementam. Assim, a construção de competências se dá sobre a articulação de metodologias e por meio das diversas formas de ação/mediação.

São elementos desse processo:

- o livro didático;
- o Espaço UnisulVirtual de Aprendizagem - **EVA**;
- as atividades de avaliação (complementares, a distância e presenciais);
- o Sistema Tutorial.

Ementa

Introdução aos conceitos de programação orientada a objetos. Abstração e modelo conceitual. Modelo de Objetos. Classes, atributos, métodos, mensagens/ações. Conceitos e Técnicas de programação. Desenvolvimento de sistemas com Classes. Bibliotecas, reusabilidade. Aplicações em ambiente WEB.

Carga horária

120 horas – 8 créditos.

Objetivos da disciplina

- Compreender as bases da Abstração.
- Aplicar a Modularidade e Componibilidade para resolver um problema.
- Organizar o modelo usando Hierarquia.
- Implementação dos princípios da Orientação a Objeto (Objeto, Encapsulamento, Classe, Atributo, Operação, Relacionamentos, Herança, Generalização e Polimorfismo).


Conteúdo programático/objetivos

Os objetivos de cada unidade definem o conjunto de conhecimentos que você deverá deter para o desenvolvimento de habilidades e competências necessárias à sua formação. Fique atento à página inicial de cada unidade para saber qual é o objetivo a ser alcançado.



Agenda de atividades/ Cronograma

- Verifique com atenção o EVA, organize-se para acessar periodicamente o espaço da Disciplina. O sucesso nos seus estudos depende da priorização do tempo para a leitura; da realização de análises e sínteses do conteúdo; e da interação com os seus colegas e tutor.
- Não perca os prazos das atividades. Registre no espaço a seguir as datas, com base no cronograma da disciplina disponibilizado no EVA.
- Use o quadro para agendar e programar as atividades relativas ao desenvolvimento da Disciplina.

Atividades obrigatórias	
Demais atividades (registro pessoal)	

UNIDADE 1

1

Conhecendo a Linguagem de Programação Java



Objetivos de aprendizagem

- Entender o que é uma linguagem de programação.
- Conhecer a história e características da linguagem.
- Identificar os tipos de aplicações Java.



Seções de estudo

Seção 1 Linguagens de Programação

Seção 2 História da linguagem Java

Seção 3 Linguagem Java

Seção 4 Plataforma Java

Seção 5 Tipos de Aplicações Java



Para início de conversa

Vamos começar a disciplina de Orientação a Objetos revisando o que é uma linguagem de programação, e conhecendo a história e características da linguagem de programação Java. Você também aprenderá vários conceitos relacionados a linguagem Java nas 5 unidades seguintes. Esse conhecimento será fundamental para o entendimento e aplicação dos conceitos do paradigma de programação orientado a objetos, que é o que vamos tratar nessa disciplina. Vamos começar?

Bom estudo!

SEÇÃO 1 - Linguagens de Programação

Uma **linguagem de programação** é um método padronizado para expressar instruções para um computador. É um conjunto de regras usadas para definir um programa de computador. Uma linguagem permite que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas sob várias circunstâncias.

Um conjunto de palavras, composto de acordo com essas regras, constitui o **código fonte** de um programa.

Programar diretamente em código de máquina costuma ser exaustivamente difícil, pois requer o conhecimento dessa sequência de números correspondente a uma sequência de instruções. Ex. decorar sequências como 1011101110110110110011001010 para cada instrução do processador.



Código fonte é o conjunto de palavras escritas de forma ordenada, contendo instruções em uma das linguagens de programação existentes no mercado, de maneira lógica.

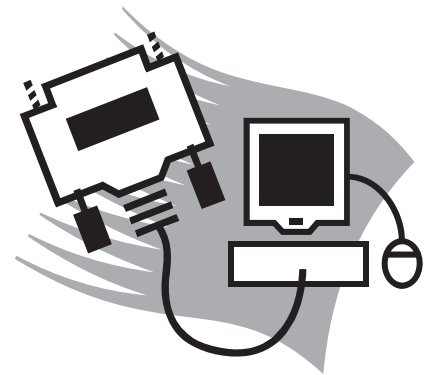
Esse código fonte é, depois, traduzido para **código de máquina**, que é executado pelo processador.



Código de máquina consiste de uma sequência de números (0 e 1) que significam uma sequência de instruções que são reconhecidas e serão executadas pelo processador do computador.

Linguagens de programação são classificadas em **alto nível** e **baixo nível**.

As **linguagens de alto nível** se caracterizam por possuírem instruções ou comando expressos sintaticamente em inglês, ou seja, mais próximo de uma linguagem humana (por isso, alto nível). Ao se utilizar uma linguagem de programação de alto nível para desenvolver um programa, está se adquirindo produtividade, pois ela permite expressar as intenções do programador mais facilmente do que quando comparado com o uso de uma linguagem que um computador entende nativamente (código de máquina). Existem muitas linguagens de programação de alto nível: Java, C, Pascal, Object Pascal, Cobol, etc.



As **linguagens de baixo nível** são aquelas cujas instruções ou comandos se aproximam bastante da linguagem ou código de máquina. Normalmente, cada instrução nesta linguagem representa uma instrução executada pelo microcomputador. A vantagem deste tipo de linguagem é a grande velocidade de execução dos programas e o tamanho dos mesmos que são mais compactos. O exemplo mais conhecido é a linguagem *Assembly*.



Uma linguagem de programação pode ser convertida em código de máquina por **compilação** ou **interpretação**.

Se o método utilizado traduz todo o código do programa (instruções que compõem o programa) para código de máquina e só depois o programa pode ser executado (ou rodado), diz-se que o programa foi **compilado** e que o mecanismo utilizado para a tradução é um **compilador** (que também é um programa). A versão compilada do programa tipicamente é armazenada em um arquivo.exe (com extensão .exe) de forma que o programa

pode ser executado um número indefinido de vezes sem que seja necessária nova compilação. Linguagens como Pascal e C são compiladas.

Se o código do programa é traduzido à medida em que vai sendo executado, isso acontece nas linguagens Javascript, Python ou Perl, em um processo de tradução de trechos seguidos de sua execução imediata, então, diz-se que o programa foi **interpretado** e que o mecanismo utilizado para a tradução é um **interpretador**. Programas interpretados são geralmente mais lentos do que os compilados.



SEÇÃO 2 - História da linguagem Java



Java é ao mesmo tempo uma **linguagem de programação** e uma **plataforma**. Você entenderá essa diferença mais à frente, estudando as próximas unidades.

Site principal da linguagem
Java: <http://java.sun.com>

Ela é desenvolvida e mantida pela **Sun** e foi planejada inicialmente para ser usada no desenvolvimento de programas que eram executados por processadores de eletrodomésticos.

Os projetistas de sistemas de controle desses processadores, descontentes com linguagens convencionais de programação, como C, propuseram a criação de uma linguagem específica para uso em processadores de aparelhos domésticos, como geladeiras e torradeiras. Todo o descontentamento dos projetistas residia no fato de que programas escritos e compilados em C são fortemente dependentes da plataforma para a qual foram desenvolvidos. Como o ramo de eletro-eletrônicos está em constante evolução, para cada novo liquidificador lançado no mercado com um novo processador embutido, um novo programa deveria ser escrito e compilado para funcionar no novo processador, ou então, na melhor das hipóteses, o antigo programa poderia ser reaproveitado, mas teria de ser re-compilado para o novo processador.

Os projetistas de software de eletrodomésticos desejavam que o software por eles fabricado fosse seguro e robusto, capaz de funcionar em um ambiente tão adverso quanto uma cozinha. E que fosse confiável também, pois quando ocorre alguma falha em um aparelho eletro-eletrônico, peças mecânicas são trocadas, gerando um custo a mais para o fabricante.

No início de 1990, Patrick Naughton, Sun Fellow e James Gosling, começaram a definir as bases para o projeto de uma nova linguagem de programação, apropriada para eletrodomésticos, sem os problemas já tão conhecidos de linguagens tradicionais como C e C++. O consumidor era o centro do projeto e o objetivo era construir um ambiente de pequeno porte e integrar esse ambiente em uma nova geração de máquinas para “pessoas comuns”.

A especificação da linguagem terminou em agosto de 1991 e recebeu o nome de “Oak” [Carvalho]. Por problemas de *copyright* (já existia uma linguagem chamada Oak) o nome foi mudado em 1995 para **Java**, em homenagem à ilha de Java, de onde vinha o café consumido pela equipe da Sun.

Em 1992, Oak foi utilizada pela primeira vez em um projeto chamado *Projeto Green*, que tinha por propósito desenvolver uma nova interface de usuário para controlar os aparelhos de uma casa. Tal interface consistia em uma representação animada da casa, que era exibida em um computador manual chamado *star seven*, bisavô dos *palmtops* de hoje e que tinha uma tela sensível ao toque que permitia a manipulação dos eletrodomésticos. Essa interface foi totalmente escrita em Oak e evoluiu de um projeto de interface para redes de televisão *pay-per-view*. Contudo, o padrão proposto por esses dois projetos, não vingou.

Em meados de 1993, pode-se dizer que Oak ia “mal das pernas”, ou seja, os projetos propostos não eram economicamente viáveis e não se via um grande futuro no desenvolvimento de aparelhos que suportassem essa nova linguagem.

Justamente nessa época, a World Wide Web estava em seu nascimento, trazendo um novo horizonte para a **Internet**.

É importante lembrar que a Internet já existia muito antes do surgimento da WWW. Esta nada mais é que um conjunto de protocolos que permite um acesso mais amigável aos recursos disponíveis na Internet. Dentre esses protocolos, por exemplo, o mais conhecido é o de transferência de hipertexto [http].

Com o lançamento do primeiro *browser* do mercado, o **Mosaic**, ocorreu à equipe de desenvolvimento da Sun, que uma linguagem independente de plataforma, segura e robusta como a que estava sendo desenvolvida para eletrodomésticos, caberia como uma “luva” para uso na Internet, uma vez que um aplicativo gerado nessa linguagem poderia rodar nos diversos tipos de computadores ligados na Internet. Ou seja, poderia rodar em qualquer computador com qualquer sistema operacional, por exemplo, PCs rodando OS/2, estações RISC rodando AIX Unix ou SparcStations rodando **Solaris**.

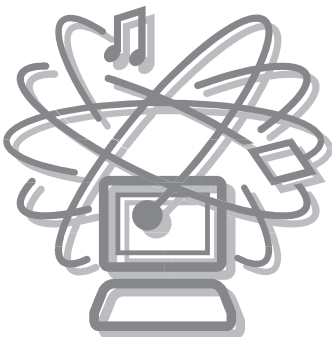
23 de maio de 1995 é a data do lançamento da linguagem Java no mundo.

Com o novo ânimo trazido pelo advento da WWW, a equipe da Sun desenvolveu um browser totalmente escrito em Java. Seu desenvolvimento terminou no início de 1995 e ele foi denominado de *HotJava*.



O grande diferencial de **HotJava** para outros browsers da época (como o Mosaic e o Netscape Navigator) é que ele permitia a inserção de programas escritos em Java dentro de páginas HTML comuns.

HotJava como browser não teve sucesso comercial, mas abriu os olhos dos desenvolvedores para um fato muito importante: as páginas HTML estariam fadadas a serem estáticas e sem ações embutidas em si, não houvesse uma linguagem padrão pela qual fossem escritos programas que pudessem ser embutidos nas páginas Web. HotJava demonstrou que isso era possível (ou seja, incluir um programa, no caso escrito em Java, em uma página HTML rodando em um browser preparado para dar suporte à execução do programa, no caso o próprio HotJava).



O grande avanço de Java veio logo a seguir, quando a Netscape anunciou que sua próxima versão do browser Navigator iria dar suporte a aplicativos Java embutidos em documentos HTML. Em seguida, a Microsoft anunciou o mesmo para o seu Internet Explorer. Após isso, Java estourou no mundo e começou a ser utilizada também fora da internet no desenvolvimento de softwares stand-alone.

E como dito anteriormente Java é ao mesmo tempo uma linguagem de programação e uma plataforma, conforme você verá a seguir.

SEÇÃO 3 – Linguagem de programação Java

Enquanto linguagem de programação, Java é considerada uma **linguagem de alto nível** que possui muitas características. Abordaremos algumas delas nessa seção. Eis algumas características da linguagem Java:

- simplicidade;
- interpretada;
- orientada a objetos;
- multiplataforma;
- segura.

Você já aprendeu o que é uma linguagem de programação de alto nível na Seção 1 - Linguagens de Programação.



Para saber um pouco mais sobre linguagem de programação, consulte os seguintes sites:

<http://www.inf.ufrgs.br/tools/java/introjava.pdf>

<http://java.sun.com/docs/white/langenv/>

Simplicidade

Java é uma linguagem simples e de fácil aprendizado ou migração, pois possui um reduzido número de construções. Contém uma biblioteca (**API Java**) de programas (conhecidos em Java como classes) que fornecem grande parte da funcionalidade básica da linguagem, incluindo rotinas de acesso à rede e criação de interface gráfica.

Application Programming Interface ou Interface de Programação de Aplicativos.

Orientada a Objetos

A partir da unidade 7 você aprenderá os conceitos do paradigma orientado a objetos.

Baseada no paradigma da **Orientação a Objetos**, cujo conceito essencial é encapsular em um bloco de memória de variáveis e métodos de manipulação dessas variáveis. A linguagem permite a modularização das aplicações, reuso de código e manutenção simples do código já implementado.

Interpretada

Linguagens de programação podem ser tanto compiladas como interpretadas.

Quando se utiliza uma linguagem compilada, após escrever o código fonte (que contém instruções numa linguagem de alto nível), é necessário que o mesmo seja traduzido (compilado) para código binário (executável). Quem faz essa tradução é um software chamado **compilador**.

Você já leu sobre isso na seção 1
- Linguagens de Programação.

As **linguagens compiladas** têm a vantagem de produzir código executável de alta performance (rápido), o qual está ajustado para o funcionamento em um tipo específico de processador ou sistema operacional. Programas compilados, chamados de **código binário**, só podem rodar no tipo de computador para o qual foram compilados, uma vez que esses programas consistem, na realidade, em instruções em linguagem de máquina, entendidas e executadas pelo microprocessador.

Nas **linguagens interpretadas**, o código fonte é interpretado por um programa chamado **interpretador** que percorre o código fonte e executa as ações indicadas pelas instruções no arquivo. O interpretador é, na realidade, o único aplicativo que está sendo executado. Um benefício das linguagens interpretadas reside no fato dos programas interpretados poderem rodar em uma variedade de plataformas diferentes, pois estes só existem em código fonte.



A linguagem Java é tanto compilada como interpretada.

Após escrever um programa em Java, seu código fonte deve ser **compilado**, gerando um arquivo binário (com código binário).

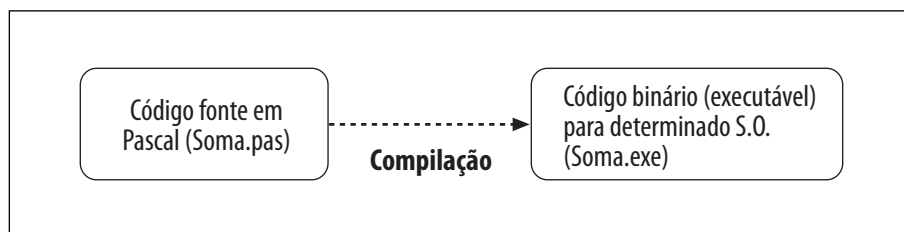
Esse arquivo não pode ser executado porque ele não contém instruções que são entendidas diretamente pelos processadores disponíveis no mercado atualmente. Os programas Java são compilados em um formato intermediário chamado **bytecode**. Esse bytecode só pode ser executado em qualquer plataforma ou sistema operacional através de um **interpretador** Java (máquina virtual) específico de uma plataforma ou sistema operacional.

Bytecode é o termo dado ao código binário (executável) gerado pela compilação de um código fonte em Java.

Multiplataforma

A característica de **multiplataforma** da linguagem Java indica que um programa desenvolvido nela pode ser executado em plataformas de hardware e sistemas operacionais diferentes. Vamos explicar com detalhes o que isso significa.

Na maioria das linguagens de programação como C e Pascal, o código fonte de um programa é **compilado** para uma plataforma de hardware e sistema operacional específicos.

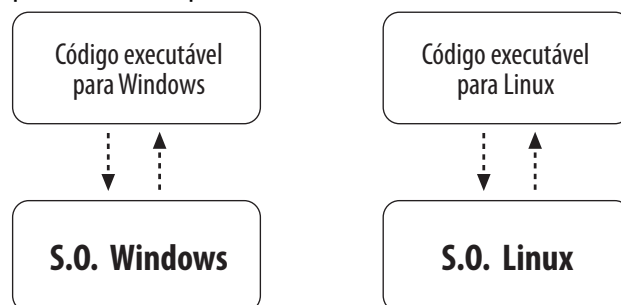


Isso faz com que o **código binário** (executável) do programa seja executado somente por computadores que tenham esse sistema operacional (S.O.).

Código executável e código binário são sinônimos.



- Código fonte compilado para o S.O. Windows irá gerar um código executável que será executado somente pelo sistema operacional Windows.
- Código fonte compilado para o S.O. Linux irá gerar um código executável que será executado somente pelo sistema operacional Linux.



Bibliotecas são rotinas, pequenos programas prontos.

Para que o mesmo programa seja executado nesses dois sistemas operacionais é necessário compilá-lo para Windows e para Linux. Porém, na maioria das vezes o programa utiliza **bibliotecas** que são dependentes de um determinado sistema operacional.



Ao programar a interface gráfica de um programa (telas) deve-se levar em conta que a interface gráfica do Windows é bem diferente da do Linux. Nesse caso, não basta apenas compilar o programa novamente, é necessário alterar o código fonte para o uso de rotinas que são suportadas pelo s.o. Linux, ou seja, reescrever pedaços do programa para que ele seja suportado por outro S.O.

Java evita que isso aconteça porque utiliza o conceito de **máquina virtual**, também conhecida como **JVM - Java Virtual Machine**. A máquina virtual pode ser entendida como uma camada extra entre o sistema operacional e o programa, responsável por “traduzir” o que o seu programa deseja fazer para a linguagem do sistema operacional no qual ela está rodando no momento.

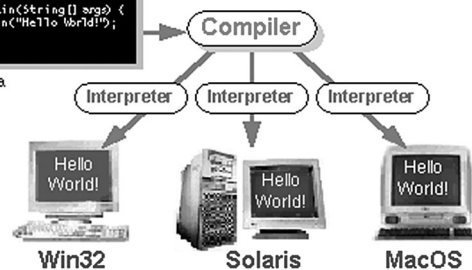
A máquina virtual executa as instruções que estão no bytecode. Esse bytecode gerado pelo processo de compilação pode ser executado pela máquina virtual instalada em computadores com plataformas e sistemas operacionais diferentes. Isso caracteriza um slogan muito utilizado pela Sun: **Write once, Run anywhere** (Escreva o código fonte uma vez e rode em qualquer lugar).



A figura abaixo ilustra esse processo:

Java Program

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
HelloWorldApp.java
```



O programa em Java é escrito e compilado uma vez. Isso gera o bytecode que será interpretado por máquinas virtuais instaladas em sistemas operacionais diferentes.

Segurança

O processo de compilação - geração de bytecodes - é projetado para a detecção prévia dos possíveis erros, evitando que os erros se manifestem em tempo de execução. O uso de código para tratamento de exceções - ***exception handling*** - permite manter a consistência da aplicação no caso de erros.

Além de diminuir as possibilidades de erro de programação, a linguagem tem um esquema de segurança para garantir a integridade de código - principalmente no caso de código originário de rede insegura.

Técnicas de programação utilizadas no momento do desenvolvimento do programa que evitam que o programa tenha algum erro de execução.

SEÇÃO 4 – Plataforma Java

Para entender Java enquanto plataforma, é preciso entender o conceito de plataforma.



Uma **plataforma** é o ambiente de hardware e software no qual um programa roda.

Exemplos de plataformas populares são: Windows 2000, Linux, Solaris, e MacOS. A maioria das plataformas podem ser descritas como uma combinação de sistema de operacional e hardware. A plataforma Java difere das outras plataformas no sentido de que é uma plataforma baseada em software que roda em cima de outras plataformas baseadas em hardware.

A plataforma Java tem dois componentes:

- Máquina virtual - *Java Virtual Machine* (**JVM**)
- API Java - *Java Application Programming Interface*

.....
Você já aprendeu um pouco sobre
o que é uma JVM no tópico
Multiplataforma da Seção 3.

A **máquina virtual** nada mais é do que um *software* que é instalado sobre uma plataforma de *hardware*-sistema operacional e é necessária para interpretar os *bytecodes* gerados pela compilação de um código fonte em Java.

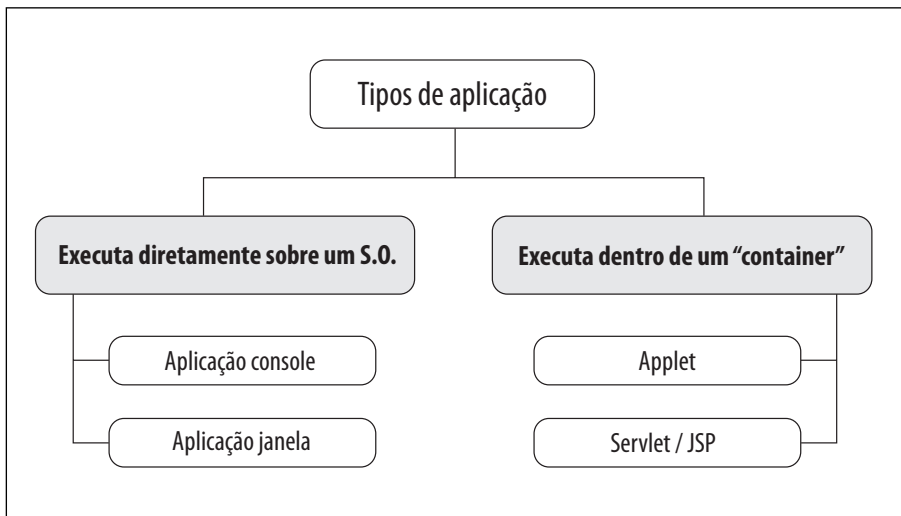
A **API Java** é uma grande coleção de componentes de *software* (programas-classes prontas) que podem ser utilizados no desenvolvimento de programas na linguagem Java, como, por exemplo, componentes que permitem a construção de interfaces gráficas (telas) de programas. Esses componentes de software, na realidade, são conjuntos de **classes** e **interfaces** organizadas em pacotes (*package*).



Fique atento, pois voltaremos a falar desses conceitos nas unidades seguintes!

A seguinte figura mostra um programa rodando sob a plataforma Java. Como a figura mostra, a API Java e máquina virtual isolam o programa da plataforma de hardware.

SEÇÃO 5 – Tipos de aplicações Java



Os programas ou aplicações escritos em Java podem ser de dois tipos:

1. Aqueles que são executados diretamente pelo sistema operacional.
2. Aqueles que são executados através de outro programa chamado genericamente de “container”.

Executados diretamente pelo sistema operacional



Aplicação console são aplicações Java que são executadas a partir da JVM instalada no sistema operacional e o seu resultado é mostrado na tela do shell (por exemplo, janela do MS-DOS), sem recursos gráficos.

Na figura abaixo, temos um exemplo simples do código fonte desse tipo de aplicação. Esse código deve ser armazenado em um arquivo de mesmo nome da classe que ele define, seguido da extensão **.java**.

```

Arquivo MeuApplication.java
public class MeuApplication {
    public static void main (String[] args) {
        System.out.println( "Este é meu application!" );
    }
}
  
```



Aplicação janela são aplicações Java que são executadas a partir da JVM instalada no sistema operacional, porém, utiliza recursos gráficos como janelas, etc.

Executados através de um “container”

Esse tipo de programa precisa de um outro software para que sua execução seja realizada.



Applet são aplicações Java que precisam de um web browser (navegador web, por exemplo, Internet Explorer) para serem executados. Isso quer dizer que applets são aplicações voltadas para o ambiente Internet/Intranet e que são transportadas pela rede junto com hipertextos HTML.

Para executar um applet dentro de um browser, a classe que representa o applet deve ser chamada dentro de um arquivo HTML. Um applet aparece numa página web semelhante a uma imagem, mas ao contrário das imagens um applet pode ser interativo capturando entradas do usuário, respondendo a elas e apresentando conteúdo mutável.



Eles podem ser usados para construir animações, gráficos, jogos, menus de navegação, etc.

Ao desenvolver um applet pode-se aproveitar recursos extras oriundos do próprio browser (uma barra de status, por exemplo).



Servlet/JSP: Um servlet é um tipo de programa Java que pode, por exemplo, receber dados de um formulário HTML por meio de uma requisição HTTP, processar os dados, atualizar a base de dados de uma empresa e gerar alguma resposta dinamicamente para o cliente que fez a requisição. Ou seja, é através do desenvolvimento de servlets que podemos desenvolver uma APLICAÇÃO WEB DINÂMICA que interaja com o usuário.

Um servlet não envia e recebe dados direto do usuário via protocolo http. Antes dos dados chegarem ao servlet eles são recebidos e tratados por outro programa chamado genericamente de container web que repassa os dados para o servlet. Por isso, servlet rodam “em cima” de um container web.



Síntese

Nesta unidade você estudou o que é uma linguagem de programação, para que ela serve e sua classificação. É importante ter em mente que sem uma linguagem de programação você não pode programar ou instruir o computador a realizar alguma tarefa.

É através de uma linguagem de programação que elaboramos o conjunto instruções organizadas logicamente (programa) que indicam o quê o computador deve fazer. Porém, a maioria das linguagens de programação é de alto nível, ou seja, sua sintaxe é semelhante à linguagem humana, portanto não entendida pelo computador.

É por isso que todo o programa (conjunto de instruções) deve ser compilado ou interpretado, isto é, traduzido para a linguagem binária que o computador entende.

A linguagem que iremos utilizar no decorrer dessa disciplina se chama **Java**. Essa linguagem se tornou muito popular no mercado de desenvolvimento de sistemas, por duas razões principais: ela é **gratuita**, ou seja, não é preciso pagar para obter os softwares necessário para construir um programa nessa linguagem e, por ser **multi-plataforma**, permitindo que um programa desenvolvido numa plataforma de hardware/sistema operacional funcione perfeitamente em outra plataforma.

Além disso, possui uma série de outras características que foram abordadas durante essa unidade, sendo possível desenvolver diversos tipos de aplicação com ela.



Atividades de auto-avaliação

- 1) Um programa feito em uma linguagem de programação de alto nível não pode ser entendido diretamente nessa linguagem pelo computador. O que precisar ser feito com esse programa em alto nível para que suas instruções possam ser entendidas pelo computador?

- 2) Explique a relação existente entre a característica Multiplataforma da linguagem Java e a expressão "write once run anywhere".

- 3) Explique o que é a máquina virtual (JVM-Java Virtual Machine) da linguagem Java.



Saiba mais

História da linguagem Java

<http://campus.fortunecity.com/psychology/196/javatudo.html>

Preparando o ambiente para programar em Java



Objetivos de aprendizagem

- Aprender a instalar e configurar o “kit” de desenvolvimento de sistemas J2SE Development Kit (JDK).
- Aprender a editar, compilar e executar um programa em Java.



Seções de estudo

Seção 1 Instalando o J2SE Development Kit (JDK)

Seção 2 Editando, Compilando e Executando o primeiro programa em Java



Para início de conversa

Depois de conhecer um pouco da história e características da linguagem Java, você vai, a partir de agora, aprender a preparar o ambiente no seu computador para desenvolver o primeiro programa em Java. Para isso, você vai precisar estar atento a uma série de detalhes que essa tarefa exige.

Até o momento, você não precisou se preocupar com isso. Nas disciplinas anteriores você aprendeu lógica de programação, ou seja, como estruturar logicamente as instruções que o computadores irá executar, sem se preocupar em passá-las para uma linguagem de programação.

A partir de agora, você irá programar utilizando uma linguagem de programação e, para isso precisará lembrar dos conhecimentos de lógica das disciplinas anteriores, além de muita atenção, pois uma linguagem de programação é cheia de detalhes sintáticos.

Você irá gostar muito dessa unidade. É aqui que vai, finalmente, colocar os conhecimentos em prática e ver seu primeiro programa funcionando.



SEÇÃO 1 - Instalando o J2SE

Para desenvolver programas na linguagem Java é necessário primeiramente escolher qual a plataforma desejada (J2SE, J2ME ou J2EE).



Nós adotaremos nesse curso a plataforma J2SE (Java 2 Platform Standard Edition), pois seu conjunto de recursos oferecidos é suficiente para o desenvolvimento de nossos programas.

Cada plataforma tem suas versões, a cada nova versão, novos recursos que facilitam o desenvolvimento de programas são adicionados. A versão mais atual da plataforma J2SE é a 5.0.

Após a escolha da plataforma, precisamos instalar e configurar o J2SE Development Kit (**JDK**), ou kit de desenvolvimento de softwares da plataforma Java 2 SE. Esse kit consiste de alguns programas como o compilador, JVM (Java Virtual Machine), API Java que são necessários para o desenvolvimento dos programas na plataforma J2SE.

O JDK da versão 5.0 pode ser baixado (fazer download) do endereço: <http://java.sun.com/j2se/1.5.0/download.jsp>.

Vamos mostrar passo a passo como fazer o *download* do JDK.

1) Acesse o endereço <http://java.sun.com/j2se/1.5.0/download.jsp> e escolha a opção **JDK 5.0 Update 6**.

A seguinte seqüência de telas irá aparecer:

Tela 1

The screenshot shows the Sun Developer Network (SDN) page for J2SE 5.0. The page has a navigation bar with 'Products and Technologies' and 'Technical Topics'. The main content area is titled 'J2SE 5.0 Download Java 2 Platform Standard Edition 5.0'. There are several sections: 'Downloads', 'Reference', 'Community', and 'Learning'. The 'Downloads' section has a link to 'Download JDK 5.0 Update 6'. A red arrow points to this link.

2) Clique no link **Download JDK 5.0 Update 6**.

Tela 2

The screenshot shows the Sun Developer Network (SDN) page for J2SE(TM) Development Kit 5.0 Update 6. The page has a navigation bar with 'Products and Technologies' and 'Technical Topics'. The main content area is titled 'Download'. There is a section for 'J2SE(TM) Development Kit 5.0 Update 6' with a list of download links. A red arrow points to the 'Accept License Agreement' button.

Windows Platform - J2SE(TM) Development Kit 5.0 Update 6		
Windows Online Installation, Multi-language	jdk-1_5_0_06-windows-i586-p.exe	59.86 MB
Windows Online Installation, Multi-language	jdk-1_5_0_06-windows-i586-p-iftw.exe	237.81 KB

3) Clique no botão **Accept** e no link **Windows Offline Installation, Multilanguage**.

Tela 3



Sun Developer Network
Products and Technologies Technical Topics

» search tips Search In Developers' Site »

Download

J2SE(TM) Development Kit 5.0 Update 6
For easier, more reliable downloads, try Sun Download Manager.

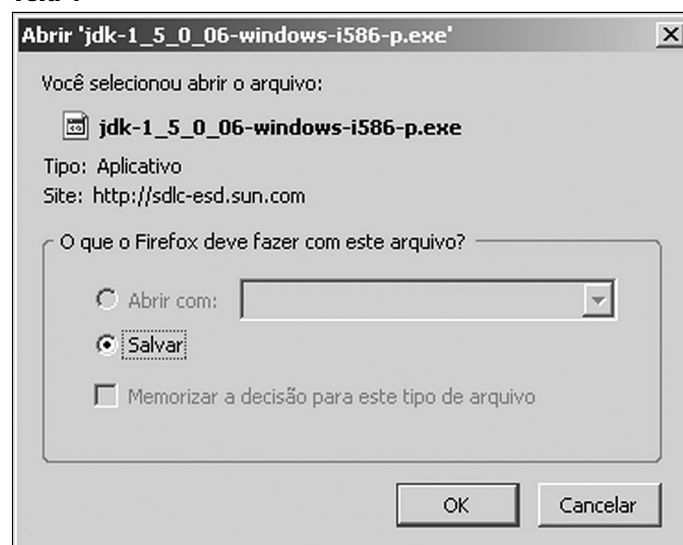
- Solaris 64-bit requires users to first install 32-bit.
- Information on picking the right format to download
- Installation instructions:
 - English
 - Japanese
- For Windows, choose "Windows Online Installation" for the quickest download and installation on a machine connected to the Internet. The default download size is **40.9MB**, which is the maximum download. The size may decrease if features are deselected, to a minimum of **18.2MB**.

NOTE: The list offers files for different platforms - please be sure to select the proper file(s) for your platform. Carefully review the files listed below to select the ones you want, then click the link(s) to download. If you don't complete your download, you may return to the Download Center anytime, sign in, then click the "Download/Order History" link on the left to continue. For any download problems or questions, please see the Download Center FAQ. How long will the download take? ⓘ

Windows Platform - J2SE(TM) Development Kit 5.0 Update 6			
⚙	Windows Offline Installation, Multi-language	jdk-1_5_0_06-windows-i586-p.exe	59.86 MB
⚙	Windows Online Installation, Multi-language	jdk-1_5_0_06-windows-i586-p-rtw.exe	237.81 KB
Linux Platform - J2SE(TM) Development Kit 5.0 Update 6			
⚙	Linux RPM in self-extracting file	jdk-1_5_0_06-linux-i586-rpm.bin	44.97 MB

4) Clique no link **Windows Offline Installation, Multilanguage**.

Tela 4



5) Clique no botão Salvar. Logo após, o download do arquivo executável contendo o J2SDK começará. Quando terminar o download, aparecerá uma janela para você escolher o diretório ou pasta dentro do disco rígido do seu computador onde o arquivo será salvo. Esse arquivo executável contém aproximadamente 50 Mb.



Convém lembrar que o conjunto de softwares contidos no J2SDK é gratuito (free), ou seja, não precisamos comprá-los para instalar no nosso computador.

O procedimento de instalação no Windows é muito simples, basta você executar o arquivo, seguir os passos e instalar no diretório desejado.



IMPORTANTE !!

Após a instalação, é necessário configurar duas **variáveis de ambiente** do sistema operacional. Essas variáveis se chamam **PATH** e **CLASSPATH**.

Isso é necessário para quê?

Para que você possa executar o compilador Java e a máquina virtual (JVM) de qualquer diretório do seu computador e para que a JVM encontre as classes compiladas que são usadas no seu programa.

Para configurar essas variáveis de ambiente no Windows, você deve atentar para a versão que está utilizando, pois em cada versão a configuração das variáveis de ambiente é feita de uma maneira diferente.

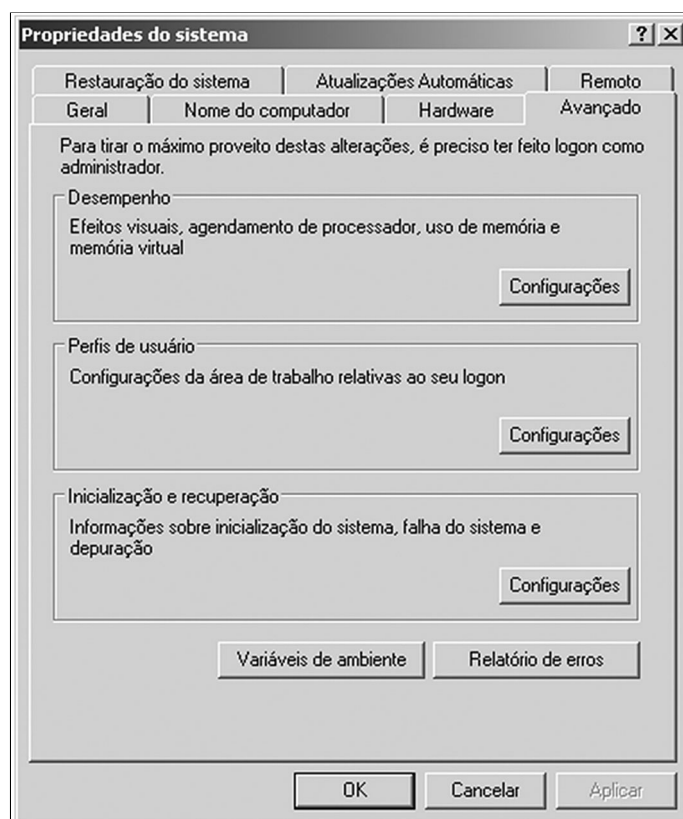
Configurando variáveis de ambiente no Windows 2000 e XP

Você deve configurar as variáveis de ambiente seguindo os seguintes passos:

1) Clique com o botão direito no ícone 'Meu Computador' e escolha a opção 'Propriedades'.

Outra opção é ir ao menu Iniciar-> Configurações->Painel de controle-> Sistema-> Avançado-> Variáveis de sistema.

A seguinte tela irá aparecer:

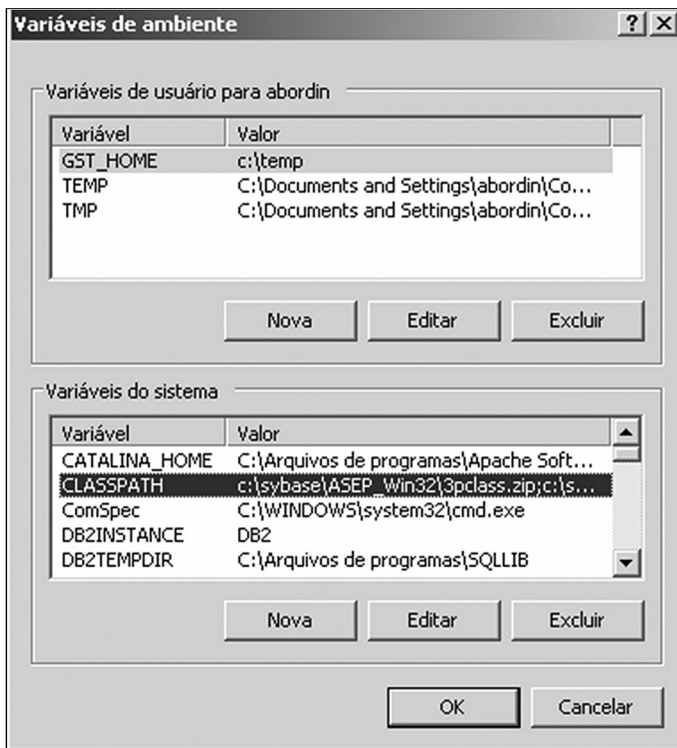


2) Clique na guia (aba) 'Avançado'

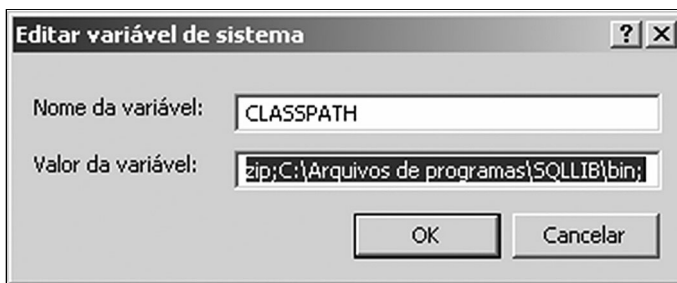
3) Clique no botão 'Variáveis de ambiente'. São duas as variáveis que você deve criar ou mudar: CLASSPATH E PATH.



A seguir será demonstrado como configurar a variável de ambiente **CLASSPATH**.



No quadro variáveis do sistema procure a variável CLASSPATH. Caso ela exista, clique no botão 'Editar'.



O campo **Valor da variável** provavelmente estará preenchido, portanto, posicione o cursor no final do conteúdo desse campo, digite ;. (ponto e vírgula e ponto) e clique no botão OK.

Caso a variável CLASSPATH não exista, clicar no botão **Nova**. A janela acima irá aparecer. No campo **Nome da variável** você deverá digitar CLASSPATH e no campo **Valor da variável** você deverá digitar . (ponto).

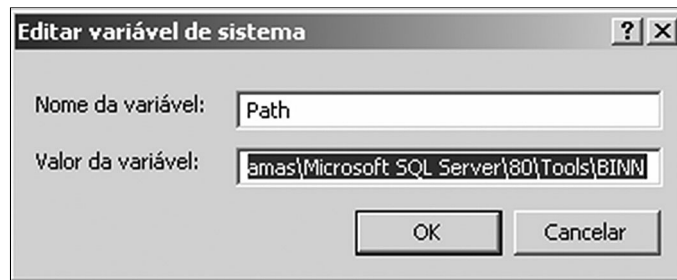


Após configurar a variável CLASSPATH vamos configurar a variável de ambiente **PATH**.

Na próxima seção você entenderá a importância dessa variável ser configurada.

A variável PATH indica para o sistema operacional onde o compilador, máquina virtual (JVM) e outros executáveis utilizados no desenvolvimento de programas em Java estarão instalados no seu computador.

No quadro ‘variáveis do sistema’, procure a variável PATH. Caso ela exista, clique no botão Editar.



O campo **Valor da variável** provavelmente estará preenchido, portanto posicione o cursor no final do conteúdo desse campo e digite ; (ponto e vírgula) seguido do caminho (pode-se falar path) onde o diretório do J2SE JDK\bin foi instalado.



;C:\jdk1.5.0_04\bin (nesse caso o JDK está instalado no diretório raiz)

;C:\Arquivos de programas\Java\jdk1.5.0_05\bin
(nesse caso o JDK está instalado dentro da pasta Arquivos de programas\Java)

Verifique pelo Windows Explorer o conteúdo do diretório \bin do J2SDK. Você verá que nesse diretório existem somente arquivos executáveis, sendo alguns: javac.exe (compilador da linguagem Java) e java.exe (máquina virtual da linguagem Java).

Configurando variáveis de ambiente no Windows 98

No Windows 98, você deve fazer as alterações das variáveis de ambiente no arquivo autoexec.bat.

Edite o arquivo e inclua as seguintes linhas:

```
SET JAVA_HOME=c:\jdk1.5.0_04\bin (diretório onde o JDK foi instalado)
SET PATH=%PATH%;%JAVA_HOME%\bin
SET CLASSPATH=.;%CLASSPATH%
```

Abra o arquivo e procure uma variável chamada PATH.

Se ela for encontrada adicione ; (ponto e vírgula) no final da linha e logo após o caminho onde o diretório J2SE JDK\bin foi instalado.

Se ela não for encontrada, digite em uma nova linha PATH= seguido do caminho onde o diretório J2SE JDK\bin foi instalado.

Configurando variáveis de ambiente no Linux

No sistema operacional Linux, a configuração das variáveis passa por processo semelhante, porém em vez de utilizar o ; (ponto e vírgula) deve-se utilizar : (dois pontos).



Para mais detalhes sobre instalação e configuração no Linux consulte:

http://www.guj.com.br/content/articles/java5_linux/guj_java_linux.pdf

SEÇÃO 2 - Editando, compilando e executando o primeiro programa em Java

Os programas em Java consistem em partes chamadas **classes**. Essas, por sua vez, consistem em partes chamadas **métodos** que realizam tarefas e muitas vezes retornam informações ao completarem tarefas.

Um **método** pode ser uma função ou um procedimento.

Os programadores podem construir novas classes ou podem utilizar as classes existentes em bibliotecas de classes prontas que Java possui. Essas bibliotecas de classes são conhecidas como Java API (Application Programming Interface ou Interface de Programas Aplicativos).



DICA!!

Releia o item Multiplataforma da unidade anterior para entender melhor essa seção.

Desenvolver um programa em Java consiste basicamente de três etapas.

1) Edição

Desenvolvimento do programa de acordo com a sintaxe da linguagem, utilizando um **software editor de texto** como o Bloco de Notas do Windows ou uma **IDE** (Integrated Development Environment - ambiente de desenvolvimento integrado) como o JCreator ou JBuilder. O arquivo contendo o programa deve ser salvo com a extensão `.java`.

Uma IDE é um software que possui recursos de editor de texto e opções que facilitam o desenvolvimento de um programa.

2) Compilação

Após a edição do programa, o mesmo deve ser compilado, ou seja, é gerado um arquivo contendo os bytecodes do programa. Os bytecodes são códigos em linguagem de máquina que serão interpretados (entendidos) pela JVM (Java Virtual Machine) que deverá estar instalada no computador. O arquivo contendo os bytecodes possui extensão `.class`.

Comando para compilação: `javac` `nomedoprograma.java`

`javac` é o nome do software compilador que deve ser executado.

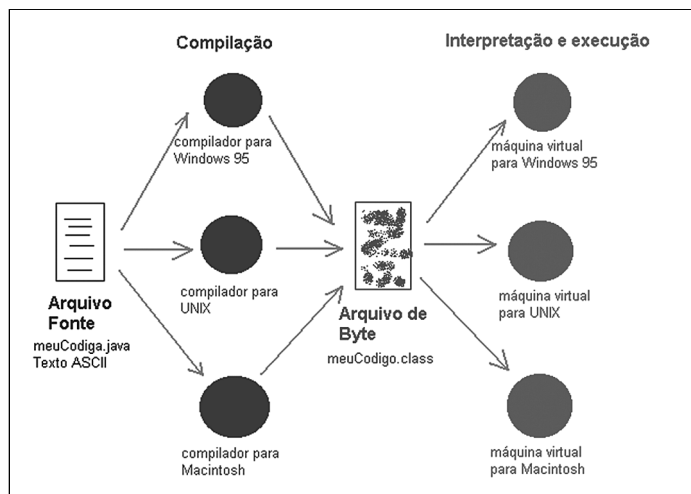
3) Interpretação

O processo de interpretação é realizado a partir dos bytecodes gerados pela compilação, que estão no arquivo `.class`. Ele é realizado pela JVM instalada no computador a qual é dependente do sistema operacional do mesmo.

Comando para interpretação: `java` `nomedoprograma`

`java` é o nome do software que representa a JVM instalada no computador.

A figura abaixo ilustra essas etapas:



Agora que você já tem o J2SDK instalado, podemos partir para a parte prática.

Meu primeiro programa Java

Primeiramente vamos, então, criar uma pasta chamada **CURSOWEB** no drive C:\ do seu computador para que possamos organizar os programas e exercícios que faremos.

Crie a pasta pela janela DOS
(c:\md cursoweb) ou pelo
Windows Explorer

Execute o Bloco de Notas (Notepad) agora vamos **digitar** nosso primeiro programa em Java. Copie as linhas abaixo:

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

Não se preocupe em entender o código. É apenas um exemplo e explicaremos esses comandos mais tarde. Salve o arquivo como AloMundo.java na pasta CURSOWEB criada anteriormente. Certifique-se que o arquivo foi salvo com esse nome e não assim: AloMundo.java.txt



Cuidar com as letras maiúsculas e minúsculas. Isso é **muito importante**. O código digitado deve estar exatamente igual ao que você está lendo.

A seguir, vamos **compilar** o programa. A compilação irá gerar os *bytecodes*.

Acesse a Janela DOS e dentro da pasta CURSOWEB execute:

```
javac AloMundo.java
```

Se não houver erro, depois de alguns segundos, você deve ter acesso ao prompt do DOS novamente.

Se der erro, você tem que identificar o que está causando o erro. As mensagens de erros estarão em inglês e, provavelmente, no caso desse programa, estarão relacionadas a algum erro de digitação do código fonte ou no comando de compilação.

Se o erro estiver no código fonte, retorne ao mesmo pelo Bloco de Notas, arrume o erro, salve o arquivo novamente, retorne ao prompt do DOS e execute o comando de compilação novamente.

Você deve repetir essa sequência de passos até que o comando de compilação não gere mais erros.

Nesse momento terá sido gerado o arquivo AloMundo.class (bytecode).

Para executar o programa, acesse a Janela DOS e dentro da pasta CURSOWEB execute digite:

java AloMundo

Você deve ter recebido como resposta a frase “Alo Mundo !”. Isso significa que tudo está certo e você acabou de criar seu primeiro programa em Java.

Ao executar o programa java você está acionando a máquina virtual de java para que ela interprete o bytecode gerado na etapa anterior.



ATENÇÃO !!

Não se preocupe se você não conseguir executar os passos descritos acima nesse momento. Na unidade 4, você aprenderá com mais detalhes a editar, compilar e executar um programa.

A intenção agora foi só dar a você após a instalação do J2SDK uma noção de como desenvolver um programa em Java.



Síntese

Nessa unidade você aprendeu a instalar e configurar a ferramenta de trabalho básica para desenvolver programas em Java, o J2SE Development Kit (JDK).

Também aprendeu a executar as três etapas do desenvolvimento de qualquer programa: edição, compilação e execução. Isso foi feito através de um pequeno exemplo que teve o objetivo de ilustrar o que vamos fazer daqui para a frente.



Atividades de auto-avaliação

- 1) Desenvolva um pequeno programa na linguagem Java que imprima o seu nome da tela. Mude o nome do programa para MeuNome.java, ou seja, no lugar de AloMundo você deve digitar MeuNome.



Saiba mais

Hello, World - Seu primeiro programa em Java!

<http://www.guj.com.br/java/tutorial/artigo.16.1.guj>

UNIDADE 3

Sintaxe básica da linguagem Java

3



Objetivo de aprendizagem

- Conhecer a sintaxe básica da linguagem.



Seção de estudo

Seção 1 Sintaxe básica da linguagem Java



Para início de conversa

Nessa unidade você obterá as informações necessárias para programar instruções na linguagem Java. Você já estudou como desenvolver programas nas disciplinas de Lógica de Programação I e II, entretanto estas disciplinas utilizaram um pseudocódigo para isso.

O computador não entende as instruções programadas em pseudocódigo, é necessário que você “ traduza ” as instruções que estão em pseudocódigo para uma linguagem de programação. Para isso, você deve aprender a sintaxe ou maneira de escrever os comandos dessa linguagem de programação.

No nosso caso, vamos utilizar a linguagem Java, portanto, é necessário aprender a sintaxe dessa linguagem.

Agora precisaremos prestar bastante atenção. Começaremos a ver como uma linguagem de programação é detalhada em sua sintaxe.

SEÇÃO 1 - Sintaxe básica da linguagem Java

Nessa seção iremos discorrer sobre a sintaxe (maneira de escrever) dos conceitos básicos envolvidos na construção de programas em Java. Conceitos como tipos de dados e estruturas de controle (decisão, repetição), entre outros.

Como não é trivial escrever programas que utilizam interface gráfica (tela com janela, botões, etc), inicialmente, você aprenderá apenas recursos para criar aplicativos ou programas simples dentro da janela do MS-DOS.

Como inserir Comentários no seu código fonte em Java?

É sempre interessante a colocação de **comentários em programas**. Os comentários permitem que a manutenção posterior do código seja mais rápida e serve para indicar o que o programa faz. Os comentários em Java podem ser de dois tipos:

// (duas barras)

ou os sinais

/* (início da marcação)

***/ (fim da marcação)**

// (duas barras)

Utiliza-se duas barras (//) em qualquer posição da linha. Tudo o que aparecer à direita das duas barras será ignorado pelo compilador.



```
A=A+1; // incremento da variável A
```

Símbolo de atribuição

O que significa essa instrução?

Uma variável de memória chamada A está **recebendo** o valor dela mesma acrescido de 1.

/* e */

Veja que o sinal de atribuição na linguagem Java é o sinal de = . Em Lógica I vocês utilizavam o sinal ←

Existem ocasiões em que várias linhas de comentário são necessárias. Nesse caso, utilizamos os sinais de /* e */ para indicar início e fim de bloco de comentários, como no exemplo:



```
/* Programa de Exemplo – Esse programa não faz nada.
Criado por XYZ em Abril de 2005
Versao 1.0
*/
```

Tipos de dados em Java

Ao desenvolver qualquer programa, freqüentemente você necessita *declarar variáveis de memória*. Para declarar uma variável de memória você deve especificar o tipo de dado que será armazenado nela.

Java é uma linguagem que necessita que todas as **variáveis** tenham um tipo de dado declarado.

Você já estudou os conceitos de variável de memória e tipo de dados. Isso foi aprendido nas Disciplinas de Lógica de Programação I e II.



Existem oito tipos de dados primitivos em Java. Seis deles são numéricos, um, é caracter e o outro é booleano (lógico).

Os tipos abaixo guardam valores numéricos inteiros (sem casas decimais). Valores negativos são permitidos.

* Representa o espaço na memória que uma variável desse tipo irá ocupar. Portanto, tenha cuidado ao declarar o tipo de uma variável de memória porque ele poderá estar ocupando mais espaço do que necessário.

Tipo	Tamanho*	Faixa de Valores**
int	4 bytes	-2.147.483.648 até 2.147.483.647
short	2 bytes	-32.768 até 32.767
byte	1 byte	-128 até 127
long	8 bytes	-9.223.372.036.854.775.808 até 9.223.373.036.854.775.807

** Diz respeito à faixa de valores que poderão ser armazenados em variáveis desse tipo.

Na maioria das ocasiões, declarar uma variável de memória com o tipo **int** é suficiente para armazenar valores numéricos sem ponto flutuante (números com casas decimais).

Tipos numéricos que representam valores com ponto flutuante (com casas decimais):

Esses tipos representam o tipo real que você aprendeu em Lógica I.

Tipo	Tamanho	Faixa de Valores
float	4 bytes	1,4E-45 a 3,4E+38
double	8 bytes	4,9E-324 a 1,7E+308

O **tipo caracter** serve para representar apenas uma letra ou número.

O tipo caracter não foi aprendido em Lógica I e será apresentado aqui.

Tipo	Tamanho	Faixa de Valores
char	2 bytes	

Ao armazenar um dado do tipo **caracter** em uma variável desse tipo, você deve representar esse dado entre aspas simples, por exemplo: 'h'. Caracteres representados por aspas duplas ("h") na verdade são **Strings**.

O tipo lógico ou booleano pode assumir apenas dois valores: verdadeiro, que aqui em Java é *true* e falso, que em Java é *false*. Esse tipo é usado apenas para testes lógicos.

String é como se chama o tipo Literal que você aprendeu em Lógica I.

Tipo	Tamanho	Faixa de Valores
boolean	1 byte	true ou false

Declaração de variáveis de memória em Java



A declaração de variáveis em Java, como em várias outras linguagens, exige que o **tipo de dado** da variável seja declarado.

Lembre-se que você já viu os tipos de dados existentes na linguagem Java na seção anterior.

Em Java, você inicia a declaração indicando o tipo de dados que será armazenado na variável e o nome dela, como no exemplo:

Java	Pseudocódigo
int a	a: <u>Numérico</u>
double d	d: <u>Numérico</u>
byte b	b: <u>Numérico</u>
char ch	c: <u>Alfanumérica</u>
boolean ok	ok: <u>Lógica</u>

Observe na tabela acima, a forma de declarar variáveis de memória em Java em relação à maneira que você declarava em pseudocódigo. O tipo deve sempre começar com letra minúscula e deve vir antes do nome da variável.

A primeira variável na tabela é do tipo **int** (numérico inteiro) e se chama **a**. Isso significa que foi alocado um espaço na memória com o nome (identificador) **a**, nesse espaço só poderão ser armazenados valores do tipo **int** e ele **ocupará 4 bytes de espaço na memória RAM**.

Note que todas as declarações terminam com o “ponto-e-vírgula”. Os nomes das variáveis devem ser iniciados com qualquer letra, seguidas por uma seqüência de letras ou dígitos. O tamanho do nome da variável não tem limites.

É possível declarar diversas variáveis em uma linha, bem como atribuir valores a elas na declaração, como nos exemplos abaixo:

```
int a,b;

int a = 10;
/* Isto é uma inicialização de variável de memória, ou seja,
quando ela é criada, um valor inicial é colocado dentro dela*/
```

Definindo constantes em Java

Você pode definir constantes em Java utilizando a palavra reservada **final**. Essa palavra indica que você definiu o valor da uma variável e que esse valor não pode ser modificado.

```
final double TEMPERATURA = 25.4;
```

Observe que o nome da constante nesse caso está em caixa alta, portanto, durante todo o programa ela deverá ser referenciada dessa maneira, em caixa alta. Essa regra vale para nomes de variáveis de memória. Isso acontece porque a linguagem Java é ***case sensitive***.

Case sensitive significa que faz diferença utilizar letras maiúsculas e minúsculas.

Operadores Aritméticos em Java

Os operadores aritméticos **+** **-** ***** **/** são utilizados para a adição, subtração, multiplicação e divisão.

A divisão retorna resultado inteiro se os operadores forem inteiros e valores de ponto flutuante (com vírgula) em caso contrário.

Se for necessário ter o valor do resto da divisão, utilizamos o % (**operador mod**).



Vamos revisar esse operador?

O operador mod é equivalente a função RESTO() aprendida em Lógica I.

Observe a seguinte operação:

2 mod 2 ou 2 % 2 (em Java)

Qual o resultado dessa operação?

0, porque o resto da divisão de 2 por 2 é 0.

Operadores relacionais e booleanos (Lógicos) em Java

Esses operadores servem para avaliar expressões lógicas. Para verificar a igualdade entre dois valores, usamos o sinal == (dois sinais de igual). O operador usado para verificar a diferença (não igual) é o !=.

Temos ainda os sinais de maior (>), menor (<), maior ou igual (>=), menor ou igual (<=).

Veja a tabela abaixo:

Operador Relacional	Significado
==	Igualdade
!=	Diferença
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual

Veja a tabela dos operadores lógicos:

Operador Lógico	Significado
&&	E
	OU
!	NÃO

Declarando variáveis do tipo Literal

Ao declarar uma variável do tipo **Literal** ou **Alfanumérica** em Java, você deve usar o tipo chamado **String** (com S maiúsculo).

Ao declarar uma variável do tipo String você estará criando um objeto em memória. Nessa variável String poderá ser armazenada uma sequência de caracteres, sejam eles letras ou números.



```
String e = ""; // String vazia. Note as aspas duplas.  
String oi = "Bom dia";
```

As Strings podem ser concatenadas (unidas), utilizando o sinal de +, como no exemplo:

```
String um = "Curso";  
String dois = "Web";  
String result = um + dois;
```

Qual o conteúdo da variável result?

CursoWeb

Esse conceito será entendido nas unidades de orientação a objetos. Saiba no momento que duas String não podem ser comparadas com o sinal de ==.



Importante

Uma String não deve ser comparada com outra usando o sinal ==, pois elas são **objetos**.

Existem dois métodos especiais para comparar objetos, um deles se chama *equals* e o outro *equalsIgnoreCase*. Assim, a comparação da String **a** com a String **b** seria:

```
a.equals(b)
```

ou

```
a.equalsIgnoreCase(b)
```

Ou seja, o que está na variável String **a** é igual ao que está na variável String **b**?

No primeiro caso, a diferença entre caracteres maiúsculos e minúsculos são avaliados. Isso quer dizer que, se em **a** tiver **WEB** e em **b** tiver **web**, essas duas variáveis não serão iguais. O segundo método ignora essas diferenças. Nesse caso as duas variáveis seriam iguais.

Declarando vetores e matrizes em Java

Vetores são estruturas utilizadas para armazenar um conjunto de dados do mesmo tipo. Todos os dados armazenados devem ser do mesmo tipo de dados. Por exemplo, todos os valores podem ser inteiros (tipo `int`).

Você aprendeu a trabalhar com vetores e matrizes em Lógica de Programação II, lembra?

Um vetor pode ser definido assim na linguagem Java:

```
int vetor[] = new int[100];
```

Aqui teremos um vetor de 100 posições (de 0 a 99) de valores inteiros. Os valores do vetor devem ser acessados segundo sua posição (também conhecido como índice), começando da posição 0.

Exemplo:

```
vetor[0] = 3; // estou armazenando o valor 3 na primeira posição do vetor  
vetor[1] = 5; // estou armazenando o valor 5 na segunda posição do vetor
```

Podemos iniciar um vetor com valores no momento da sua declaração, como abaixo:

```
int[] impares = {2,3,5,7,9,11,13} //foi criado um vetor chamado impares com 7 posições.
```

Como acessar a posição 3?

```
impares[2];
```

É possível definir vetores de várias dimensões. Com duas dimensões eles são conhecidos como matriz. A definição de uma matriz em Java é a seguinte:

```
int matriz[][] = new int[5][6];
```

Foi criada uma matriz de 5 linhas por 6 colunas.



Como acessar o valor que está na 1ª linha e 3ª coluna dessa matriz?

Lembre-se que na linguagem Java os índices de acesso tanto de vetor como de matriz começam **SEMPRE** em 0.

```
matriz[0][2]
```

Estou acessando o valor que está na primeira linha (linha 0) e na terceira coluna (coluna 2).

Entrada e saída de dados em Java

Em Lógica I e Lógica II você estudou o que é uma **instrução de entrada de dados** e uma **instrução de saída de dados**.

Vamos estudar agora como **implementar** (programar) essas instruções na linguagem Java.

Qualquer instrução de entrada de dados deve ser precedida por uma declaração de variável de memória, pois, o dado digitado pelo usuário na entrada de dados deve ficar armazenado em algum lugar, ou seja, numa variável de memória.

Vamos revisar várias entradas e saídas de dados em pseudocódigo primeiramente e, mostrar em seguida, a sua implementação na linguagem Java.

1) Entrada e saída de dados do tipo numérico inteiro

PSEUDOCÓDIGO

```
1  idade: numérico
2  Leia idade
3  Escreva "A idade é ", idade
```

JAVA

```
1  int idade;
2  idade=Integer.parseInt(JOptionPane.showInputDialog("Entre com a idade"));
3  JOptionPane.showMessageDialog(null, "A idade é "+idade);
```

Como você deve ter observado, a implementação em Java das três linhas do pseudocódigo é simplesmente uma **TRADUÇÃO** das mesmas.

Na linha 1 foi declarada uma variável de memória chamada idade do tipo int (int corresponde ao tipo numérico inteiro).

A linha 2 corresponde a uma instrução de entrada de dados, no caso, da idade. Em Java vamos usar a instrução **JOptionPane**.

showInputDialog(“ ”) para programar todas as entradas de dados. Essa instrução faz com que apareça uma janelinha ou caixa de diálogo (figura 1) com um campo para o usuário digitar o dado.



Figura 1 – Tela de Entrada de Dados

Entre as aspas da instrução deve estar o texto que aparecerá dentro da caixa e que deve informar o quê o usuário deve digitar.

Após a digitação, o usuário deve clicar no botão OK.

O valor digitado pelo usuário nessa caixa **SEMPRE** será do tipo String (literal), conseqüentemente, não podemos programar para que ele seja armazenado diretamente na variável idade que só aceita dados do tipo int. Para resolver esse problema temos que usar a instrução de **CONVERSÃO** de dados **Integer.parseInt()** que converte de String para int porque o dado (idade) que o usuário digitará na caixa é do tipo String, portanto, deve ser convertido para int para ser armazenado (atribuído) na variável idade.



O valor a ser convertido deve ser colocado entre os parênteses da instrução.

A linha 3 corresponde a uma instrução de saída de dados, no caso, do valor da idade digitado. Em Java vamos usar a instrução **JOptionPane.showMessageDialog(null, “ ”+ variável)** para mostrar qualquer palavra, frase, valor no vídeo. Essa instrução faz com que apareça uma janelinha ou caixa de diálogo (figura 2) com a palavra, frase programada entre as “ ” (aspas) seguido do conteúdo da variável de memória (se existir).

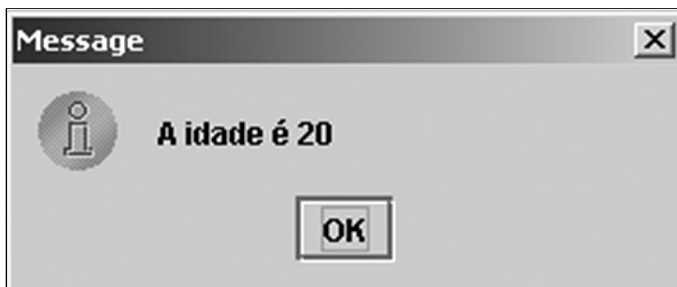


Figura 2 – Tela de saída de dados

Entre as aspas da instrução deve estar o texto fixo que aparecerá dentro da caixa, que, normalmente, é um rótulo ou legenda para o resultado (conteúdo de uma variável de memória) a ser mostrado. Após a visualização o usuário deve clicar no botão OK. Nessa instrução de saída de dados, o parâmetro null sempre teve ser digitado depois da abertura de parêntese.



Observe que no final de todas as instruções em Java deve ser digitado ; (ponto e vírgula).

2) Entrada e saída de dados do tipo numérico real

PSEUDOCÓDIGO

```
1  valor: numérico
2  Leia valor
3  Escreva "O valor é ", valor
```

JAVA

```
1  double valor;
2  valor=Double.parseDouble(JOptionPane.showInputDialog("Entre com o valor"));
3  JOptionPane.showMessageDialog(null, "O valor é "+valor);
```

Na linha 1 foi declarada uma variável de memória chamada idade do tipo double (double corresponde ao tipo numérico real).

Declara-se variável do tipo double sempre que houver necessidade de se armazenar um número com casas decimais.

A linha 2 corresponde a uma instrução de entrada de dados, no caso, valor. Novamente use a instrução **JOptionPane.showInputDialog()** para programar essa entrada de dado. O valor digitado pelo usuário na caixa de diálogo que irá aparecer SEMPRE será do tipo String (literal), deste modo não podemos programar para que ele seja armazenado diretamente na variável valor que só aceita dados do tipo double.

Para resolver esse problema temos que usar a instrução de **CONVERSÃO** de dados **Double.parseDouble()**, que converte de String para double porque o dado (valor) que o usuário digitará na caixa é do tipo String, assim, deve ser convertido para double para ser armazenado (atribuído) na variável valor. **O valor a ser convertido deve ser colocado entre os parênteses da instrução.**

A linha 3 corresponde a uma instrução de saída de dados, no caso, do valor digitado. Usamos a instrução **JOptionPane.showMessageDialog()** para mostrar na caixa a legenda o “O valor é” seguido do dado armazenado na variável valor.

Observe o operador + depois da legenda que está entre aspas. Esse operador não está somando a legenda com o valor da variável valor e sim **CONCATENDO** (unindo) a legenda que está entre aspas (um dado String) com o valor da variável valor que é do tipo double. Com isso, o tipo da variável valor (double) é automaticamente ou implicitamente convertido para String, permitindo assim que duas Strings sejam concatenadas.

3) Entrada e saída de dados do tipo literal

PSEUDOCÓDIGO

```
1  Nome: literal
2  Leia Nome
3  Escreva "O nome é ", nome
```

JAVA

```
1  String Nome;
2  Nome=JOptionPane.showInputDialog("Entre com o nome");
3  JOptionPane.showMessageDialog(null, "O nome é "+Nome);
```

Na linha 1 foi declarada uma variável de memória chamada Nome do tipo String (String corresponde ao tipo literal).

A linha 2 corresponde a uma instrução de entrada de dados em Java. O valor digitado pelo usuário na caixa de diálogo que irá aparecer SEMPRE será do tipo String (literal), deste modo, NESSE CASO especificamente, podemos programar para que ele seja armazenado diretamente na variável Nome que só aceita dados do tipo String, **não sendo necessário** usar uma instrução de **CONVERSÃO** de dados. Mas apenas nesse caso!

A linha 3 corresponde a uma instrução de saída de dados do nome digitado.



Síntese

Vamos revisar as principais instruções aprendidas nessa unidade:

Instrução de Entrada de Dados em Java

```
JOptionPane.showInputDialog(" ");
```

Observe as letras maiúsculas e minúsculas da instrução.

Instrução de Saída de Dados em Java

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

Instruções de Conversão de Tipo de Dados

<code>Integer.parseInt ()</code>	Converte um valor do tipo String (literal) para o tipo int (numérico inteiro).
<code>Double.parseDouble()</code>	Converte um valor do tipo String (literal) para o tipo double (numérico real)
<code>Float.parseFloat ()</code>	Converte um valor do tipo String (literal) para o tipo float (numérico real)
<code>Integer.toString ()</code>	Converte um valor do tipo int (numérico inteiro) para o tipo String (literal)
<code>Double.toString()</code>	Converte um valor do tipo double (numérico real) para o tipo String (literal)



Atividades de auto-avaliação

- 1) Faça as seguintes declarações de variáveis na linguagem Java:
 - a. Declare uma variável de memória para armazenar um valor numérico inteiro, um valor numérico real, um valor do tipo lógico, um valor do tipo Literal.
 - b. Declare um vetor de 5 posições para armazenar valores inteiros.
 - c. Declare uma matriz de 3 x 2 para armazenar valores do tipo literal.
- 2) Produza o seguinte trecho de código na linguagem Java:

Criar uma variável do tipo literal e armazenar o valor "Unisul"; criar outra variável do tipo literal e armazenar o valor "Virtual"; concatenar (unir) essas duas variáveis e armazenar em uma terceira variável.
- 3) Traduza os seguintes trechos de pseudocódigo para a linguagem Java:

```
2.1)
A,B: numérico
A ← 5
B ← 6
C ← A+B
Escreva C;
```

- 4) Escreva um trecho de código em Java que leia o nome de uma pessoa e a sua idade. Logo após, escreva o nome da pessoa e a sua idade.



Saiba mais

Fundamentos da Linguagem

http://www.jspbrasil.com.br/jsp/tutoriais/tutorial.jsp?idTutorial=002_002

Implementando os primeiros programas em Java



Objetivos de aprendizagem

- Entender as etapas necessárias para o desenvolvimento de um programa.
- Implementar programas em Java sem controle de fluxo.



Seção de estudo

Seção 1 Implementando os primeiros programas em Java



Para início de conversa

Depois de ter estudado as unidades 1, 2 e 3, você está pronto para “colocar a mão na massa”, ou seja, começar a implementar os primeiros programas em Java, pois já viu como instalar o J2SDK e aprendeu a sintaxe básica da linguagem.

Agora, falta começar a programar!

Para isso, você vai ter que seguir passo a passo e com muita atenção às orientações dessa unidade. Vamos desenvolver programas em Java sem controle de fluxo, com fluxo seqüencial, ou seja, sem SE, ENQUANTO ou PARA.

E então?

Você está pronto para começar?

SEÇÃO 1 - Implementando os primeiros programas em Java

Agora que você estudou a sintaxe básica da linguagem Java na unidade 3, está pronto para começar a desenvolver os primeiros programas nessa linguagem. Esses programas não vão envolver estruturas de controle de fluxo (se – enquanto – para), serão programas com um fluxo seqüencial (do início ao fim).

Porém, antes de seguir no estudo dessa seção, gostaria que você voltasse à unidade 2, seção 2: “Editando, Compilando e Executando o primeiro programa em Java” para relembrar os três passos para desenvolver um programa em Java.

Passos iniciais

1º Passo: certifique-se que o SDK da linguagem Java está instalado no seu computador e se está devidamente configurado.

- Para verificar se ele está **instalado** abra o Windows Explorer e procure por alguma pasta chamada jdk1.5...
- Para verificar se ele está **configurado** entre na opção prompt do MS-DOS ou prompt de Comando ou clique no botão Iniciar, opção Executar e digite cmd. Com isso você irá entrar na janela no MS-DOS. Nessa janela, digite **javac** (java é nome do compilador), se aparecer uma mensagem do tipo: “javac não é reconhecido como um comando interno ou externo.....” significa que o SDK não está devidamente configurado. Para isso, volte à seção 1 da unidade 2 – Instalando o J2SE que explica passo a passo como instalar e configurar o J2SE.

Os três pontinhos no final significam que o nome da pasta normalmente começa por jdk, mas pode terminar em 1.4.1 ou 1.4.1_02, 1.5.0, etc. dependendo da versão que você instalou na sua máquina.

2º Passo: crie uma pasta dentro do diretório (pasta) raiz do seu computador.

O nome da pasta deve se chamar CURSOWEB.

Vamos agora começar a desenvolver em Java os algoritmos que você desenvolveu na disciplina de Lógica de Programação I.

Você começará revendo o enunciado do exercício 3 que está nas atividades de auto-avaliação da seção 1, Unidade 5, da disciplina de Lógica de Programação I. Primeiramente, será mostrado o pseudocódigo do algoritmo e, posteriormente, o algoritmo implementado na linguagem Java de acordo com a lógica empregada no pseudocódigo mostrado a seguir.

Para quem quiser aprender a manipular arquivos (criar pastas, copiar arquivos, apagar arquivos, etc) dentro da janela do MS-DOS, consulte o endereço: <http://www.infowester.com/tutdos.php>

É importante ressaltar essa frase “de acordo com o pseudocódigo mostrado a seguir”. Por quê? Por que podemos desenvolver um algoritmo de diversas maneiras e isso altera como o algoritmo é implementado na linguagem de programação.

a) Crie um algoritmo em pseudocódigo que cadastre e mostre, no vídeo do computador, os dados de um professor: nome, endereço, cidade, estado, CEP, RG, data de nascimento, grau de escolaridade e curso que leciona.

PSEUDOCÓDIGO

```
Início
    NOME, ENDERECO, CIDADE, ESTADO, CEP,
    DATANASC: literal
    GRAUESC, CURSO: literal
    RG: numérico
    Leia NOME
    Leia ENDERECO
    Leia CIDADE
    Leia ESTADO
    Leia CEP
    Leia RG
    Leia DATANASC
    Leia GRAUESC
    Leia CURSO
    Escreva "Nome", NOME
    Escreva "Endereço", ENDERECO
    Escreva "Cidade", CIDADE
    Escreva "Estado", ESTADO
    Escreva "CEP", CEP
    Escreva "RG", RG
    Escreva "Data de Nascimento", DATANASC
    Escreva "Grau de Escolaridade", GRAUESC
    Escreva "Curso", CURSO
Fim
```

JAVA

```

0  import javax.swing.*;
1  class DadosProfessor {
2  public static void main(String args[])
3  {
4  String NOME, ENDERECO, CIDADE, ESTADO, CEP, DATANASC, GRAUESC, CURSO;
5  int RG;
6  NOME = JOptionPane.showInputDialog("Entre com o Nome");
7  ENDERECO = JOptionPane.showInputDialog("Entre com o Endereço");
8  CIDADE = JOptionPane.showInputDialog("Entre com a Cidade");
9  ESTADO = JOptionPane.showInputDialog("Entre com o Estado");
10 CEP = JOptionPane.showInputDialog("Entre com o CEP");
11 RG = Integer.parseInt(JOptionPane.showInputDialog("Entre com o RG"));
12 DATANASC = JOptionPane.showInputDialog("Entre com a Data de Nascimento");
13 GRAUESC = JOptionPane.showInputDialog("Entre com o Grau de Escolaridade");
14 CURSO = JOptionPane.showInputDialog("Entre com o Curso");
15 JOptionPane.showMessageDialog(null, "Nome " + NOME);
16 JOptionPane.showMessageDialog(null, "Endereço " + ENDERECO);
17 JOptionPane.showMessageDialog(null, "Cidade " + CIDADE);
18 JOptionPane.showMessageDialog(null, "Estado " + ESTADO);
19 JOptionPane.showMessageDialog(null, "CEP " + CEP);
20 JOptionPane.showMessageDialog(null, "RG " + RG);
21 JOptionPane.showMessageDialog(null, "Data de Nascimento " + DATANASC);
22 JOptionPane.showMessageDialog(null, "Grau de Escolaridade " + GRAUESC);
23 JOptionPane.showMessageDialog(null, "Curso " + CURSO);
24 System.exit(0);
25 }
26 }

```

Antes de digitar esse código vamos analisá-lo linha a linha.

- **Linha 0:** nos programas que faremos essa instrução será sempre necessária e deve estar na primeira linha. Posteriormente ela será explicada.
- **Linha 1:** todo o programa desenvolvido em Java é chamado de classe ou **class**. Em Java todo o programa ou **class** deve ter um nome. O nome do nosso programa

Vamos nos referenciar a classe como **class** daqui para a frente.

é `DadosProfessor`. Logo, a primeira linha deve ser a declaração **class** seguido do nome do programa e o símbolo de início da class { (abre chave);

- **Linha 2:** todo o programa (conjunto de instruções) feito em Java deve estar dentro de pequenos módulos também conhecidos como sub-rotinas ou métodos, que podem estar na forma de **procedimentos ou funções**. Pelo menos um método deve existir (a princípio) em nossos programas (**class**) Java e ele é chamado de **main** (em inglês: principal). Todo o conjunto de instruções deve estar (a princípio) dentro do método **main** e ele SEMPRE deve ser declarado da maneira como está na linha 2.
- **Linha 3:** corresponde a instrução de início de bloco. O símbolo que indica início de bloco em Java é {
- **Linha 4:** declaração de oito variáveis de memória do tipo String (literal)
- **Linha 5:** declaração de uma variável de memória do tipo int (inteiro)
- **Linha 6:** instrução de entrada de dados em Java. Observe a instrução **JOptionPane.showInputDialog** (“Entre com o Nome”). Essa instrução faz com que apareça uma janelinha ou caixa de diálogo na tela com um campo para que o usuário do programa digite o nome (figura 3). **Qualquer valor digitado nessa caixa de diálogo sempre será do tipo String.** Após a digitação do nome, o usuário deve clicar no botão OK. O valor digitado na caixa será armazenado(atribuído) na variável NOME que só aceita dados do tipo String porque ela foi declarada como sendo do tipo String.
- **Linhas 7, 8, 9 e 10:** a mesma explicação se repete para as linhas de 7 a 10.

.....
 Você aprendeu funções e
 procedimento em Lógica II.

.....
 Nós iremos utilizar essa instrução
 de entrada de dados `JOptionPane.
 showInputDialog` no decorrer deste
 livro.

- **Linha 11:** instrução de entrada de dados em Java. Observe a instrução `JOptionPane.showInputDialog("Entre com o RG")`. Essa instrução faz com que apareça uma janelinha ou caixa de diálogo na tela com um campo para que o usuário do programa digite o RG. Qualquer valor digitado nessa caixa de dialogo sempre será do tipo `String`. Se o valor é do tipo `String`, então ele não pode ser armazenado, atribuído diretamente a variável `RG` que só aceita valores do tipo `int` (inteiro). Por causa disso, programamos a instrução **`Integer.parseInt(..)`** que irá converter o tipo do dado digitado na caixa de diálogo (`RG`) para o tipo `int` (inteiro). Depois do `RG` digitado ser convertido para `int`, ele pode ser armazenado na variável `RG`.
- **Linhas 12,13,14:** a mesma explicação da linha 6 se repete para as linhas 12 a 14.
- **Linha 15:** a instrução de escrita ou saída de dados em Java que iremos utilizar é: **`JOptionPane.showMessageDialog`** (`null`, "Nome " + `NOME`). Ela faz com que apareça na tela uma janelinha ou caixa de diálogo com a frase que foi programada entre " " seguido do conteúdo de alguma variável de memória (figura 4).
- **Linhas 16 a 23:** a mesma explicação da linha 15 se repete para as linhas de 16 a 23.
- **Linha 24:** essa instrução faz com que o programa termine adequadamente e não fique "pendurado" na janela do MS-DOS.
- **Linha 25:** instrução de fim do método `main` } (fecha chave).
- **Linha 26:** instrução de fim da class } (fecha chave).

Nós iremos utilizar essa instrução de saída de dados `JOptionPane.showMessageDialog` no decorrer deste livro.

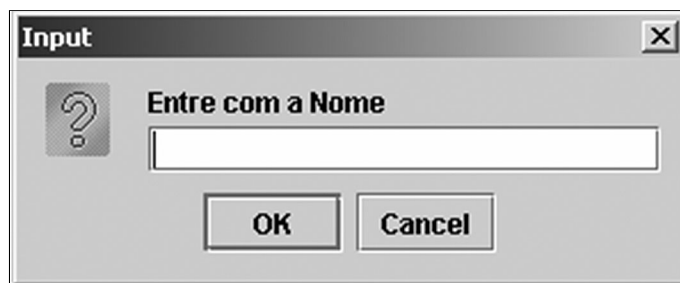


Figura 3 – Tela de Entrada de Dados

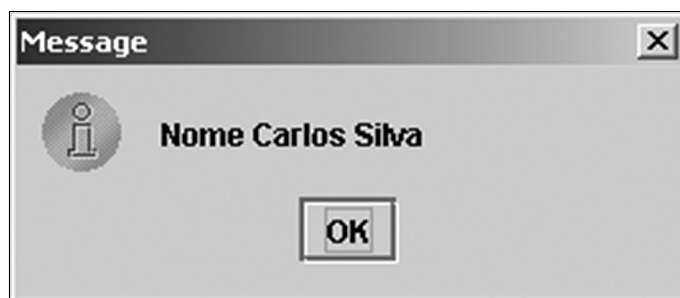


Figura 4 – Tela de Saída de Dados

Depois de analisar linha a linha do código, vamos partir para a prática, ou seja, para o desenvolvimento do programa.

Para isso vamos primeiramente revisar as três partes do desenvolvimento de um programa: **EDIÇÃO, COMPILAÇÃO E EXECUÇÃO**.

1) Na **EDIÇÃO**

você deve digitar o programa acima num editor de texto. Vamos usar o Bloco de Notas do Windows. Abra o Bloco de Notas, digite o programa CUIDANDO letras minúsculas e maiúsculas e salve-o na pasta CURSOWEB criada anteriormente.



Saiba mais

Você pode utilizar uma IDE para o desenvolvimento de seus programas. Uma IDE é uma ambiente de desenvolvimento (software) onde o programador pode digitar o programa, compilar e executar. Existem várias IDE's no mercado. Algumas mais profissionais como JBuilder, Eclipse, NetBeans e outras mais simples como JCreator, JEdit.

**IMPORTANTE**

Na hora de salvar o arquivo contendo o código digitado, utilize o mesmo nome da class, ou seja, se a class se chama `DadosProfessor`, o nome do arquivo deve se chamar `DadosProfessor.java`. Não esqueça a extensão (`.java`)

2) Depois de digitar o programa, você deve COMPILAR.

Para isso, vamos entrar na janela do MS-DOS e acessar a pasta `WEBCURSO`. Para conseguir acessar essa pasta pela janela do MS-DOS você deve utilizar alguns comandos do MS-DOS.

Revisão de comandos básicos do MS-DOS:

<code>cd \</code>	Comando que permite acessar a pasta (diretório) raiz
<code>cd ..</code>	Comando que permite voltar um nível da árvore de pastas
<code>cd <nome da pasta></code>	Comando que permite acessar, entrar em uma pasta
<code>md <nome da pasta></code>	Comando que permite criar uma pasta

Estando dentro da pasta `WEBCURSO` você deve digitar o comando para compilar o programa (class) `DadosProfessor.java`

```
C:\webcurso\javac DadosProfessor.java
```

A sintaxe do comando de compilação é:

```
javac <nome da class.java>
```

**IMPORTANTE**

Na hora de compilar o programa não esqueça de digitar o nome da classe seguido de `.java`. Preste atenção para digitar o nome da class da mesma maneira que você a salvou.

Por exemplo: Se você salvou o arquivo como dadosProfessor.java (d minúsculo), deve digitar no momento da compilação dessa mesma maneira. Depois de digitar o comando de compilação, tecele ENTER. Após isso, o software compilador irá varrer linha por linha o código fonte digitado transformando-o para um código (bytecode) que a JVM (máquina virtual da linguagem Java) irá entender. Se forem encontrados erros de sintaxe (digitação) no programa, a compilação não terminará com sucesso e será apresentado na tela, a quantidade de erros encontrados, a linha onde cada erro ocorreu dentro do programa e uma breve descrição do mesmo.

Veja as figuras abaixo:

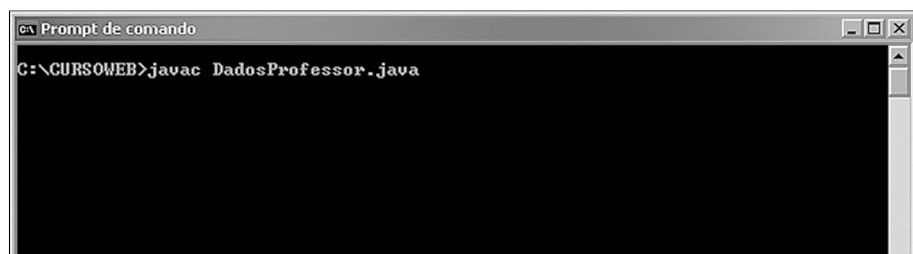


Figura 5 – Comando de compilação

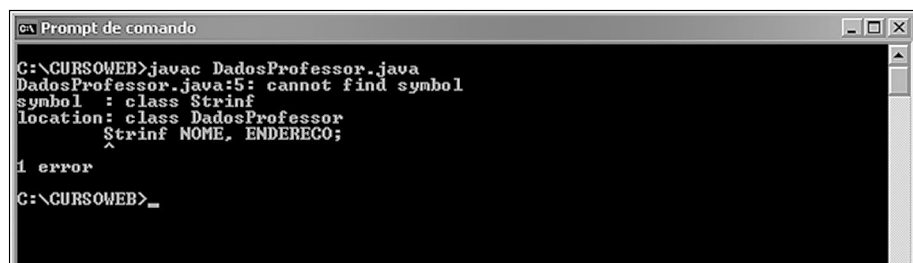


Figura 6 – Tela com os erros de compilação

Como você pode notar na figura, o compilador detectou 1 erro no programa (mas ele poderia ter detectado mais) e mostrou na tela as informações relativas ao erro que encontrou.

Logo na primeira linha, após o comando de compilação, ele identifica a linha em que ocorreu o erro (linha 5) e, logo em seguida, uma breve descrição do erro ("cannot find symbol") que, nesse caso, quer dizer que a compilação não reconheceu algum comando, palavra (que ele chama de symbol-símbolo) na linha 5.

Na segunda linha, ele identifica qual é o symbol não reconhecido. Nesse caso ele se chama Strinf. Aqui já podemos perceber que na linha 5, deveríamos ter digitado String em vez de Strinf (String é um symbol conhecido pelo compilador, Strinf não).

Na terceira linha, ele identifica em qual class esse erro ocorreu. Nesse caso foi na class DadosProfessor. Logo embaixo, ele identifica o que foi digitado na linha 5.

Com essas informações você deve **RETORNAR** ao código fonte digitado (arquivo DadosProfessor.java), ou seja, deve abrir o arquivo e arrumar o erro.

Em seguida deve **SALVAR** e **COMPILAR** o arquivo novamente.

Esse processo deve se repetir até que apareça a seguinte tela.

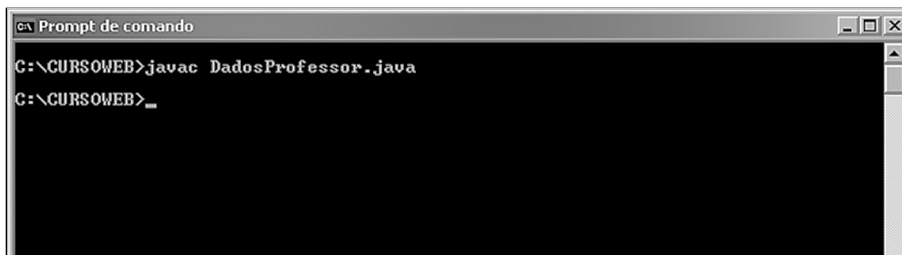


Figura 7 – Tela após compilação ter sido realizada com sucesso.

Como você pode observar, o compilador não detectou nenhum erro.

O resultado do processo de compilação é um outro arquivo chamado DadosProfessor.class. Observe que a extensão é .class. É esse arquivo que contem o código (bytecode) que será **interpretado** pela JVM (máquina virtual da linguagem Java) instalada no seu computador.

Se você acessar o conteúdo da pasta CURSOWEB, verá que existe dois arquivos lá: DadosProfessor.java e DadosProfessor.class.

Dica: Leia novamente a seção 1.3.1 que explicar a diferença entre linguagens interpretadas e compiladas.

3) Após a compilação, o terceiro passo é **EXECUTAR**

o programa, ou seja, testar para ver se está funcionando corretamente. Para isso, digite:

```
C:\cursoweb\java DadosProfessor
```

A sintaxe do comando que executa ou interpreta um arquivo .class é:

```
javac <nome da class>
```

Observe que ao executar o comando javac, estamos invocando a JVM da linguagem Java e informando a ela que a class que será interpretada.



IMPORTANTE!

Você não deve digitar a extensão .class, somente o nome do arquivo .class gerado após a compilação.

No caso desse programa, a sequência de execução seguirá o formato das seguintes telas:

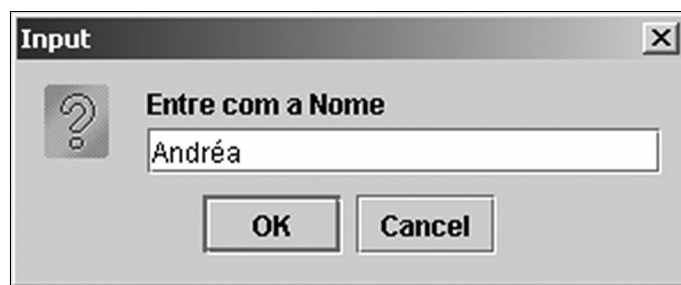


Figura 8 – Tela de entrada do Nome

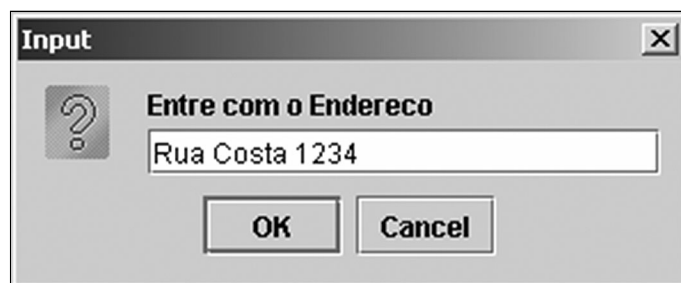
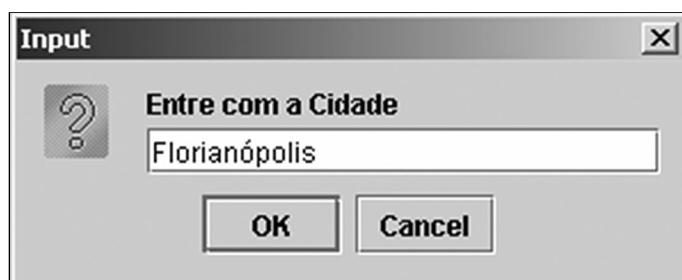
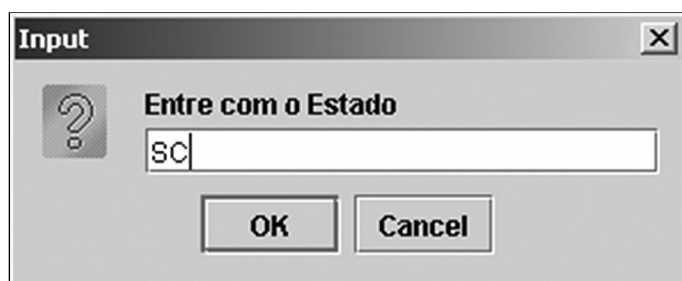


Figura 9 – Tela de entrada do Endereço



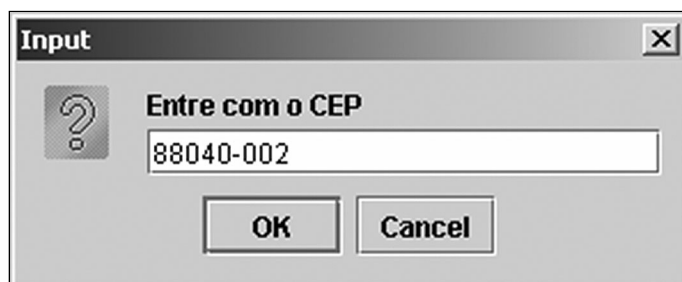
A screenshot of a Windows-style dialog box titled "Input". It contains a question mark icon, the text "Entre com a Cidade", a text input field containing "Florianópolis", and two buttons labeled "OK" and "Cancel".

Figura 10 – Tela de entrada da Cidade



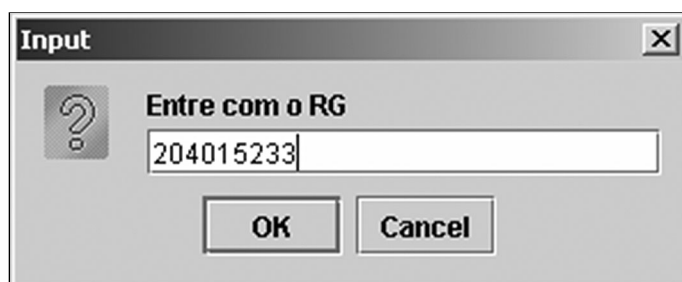
A screenshot of a Windows-style dialog box titled "Input". It contains a question mark icon, the text "Entre com o Estado", a text input field containing "SC", and two buttons labeled "OK" and "Cancel".

Figura 11 – Tela de entrada do Estado



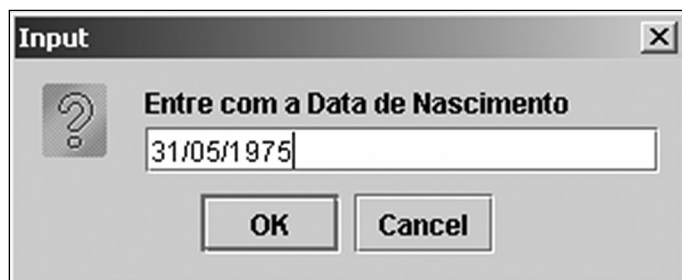
A screenshot of a Windows-style dialog box titled "Input". It contains a question mark icon, the text "Entre com o CEP", a text input field containing "88040-002", and two buttons labeled "OK" and "Cancel".

Figura 12 – Tela de entrada do CEP



A screenshot of a Windows-style dialog box titled "Input". It contains a question mark icon, the text "Entre com o RG", a text input field containing "204015233", and two buttons labeled "OK" and "Cancel".

Figura 13 – Tela de entrada do RG



A screenshot of a Windows-style dialog box titled "Input". It contains a question mark icon, the text "Entre com a Data de Nascimento", a text input field containing "31/05/1975", and two buttons labeled "OK" and "Cancel".

Figura 14 – Tela de entrada da Data de Nascimento

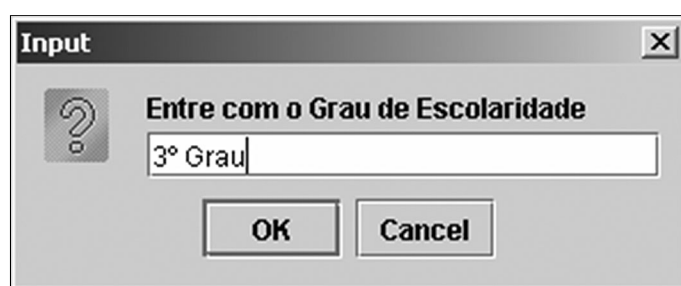


Figura 15 – Tela de entrada do Grau de Escolaridade

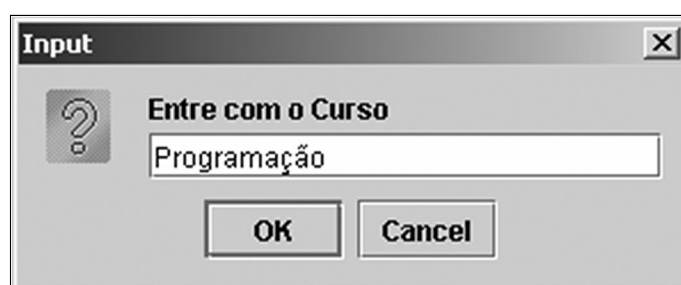


Figura 16 – Tela de entrada do Curso

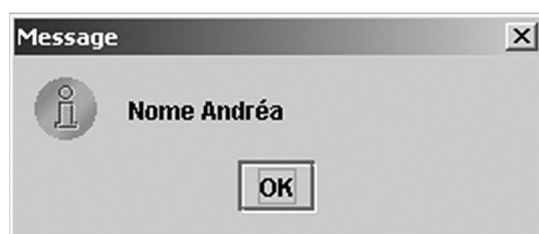


Figura 17 – Tela de saída do Nome

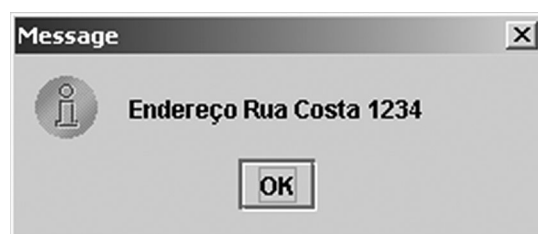


Figura 18 – Tela de saída do Endereço



Figura 19 – Tela de saída da Cidade



Figura 20 – Tela de saída do Estado

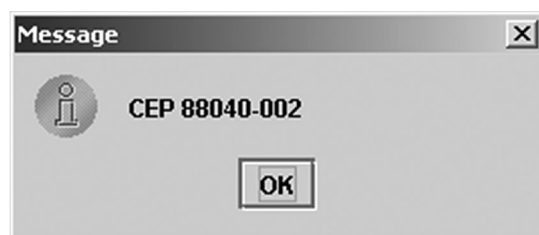


Figura 21 – Tela de saída do CEP

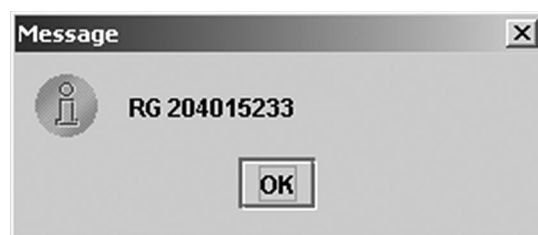


Figura 22 – Tela de saída do RG

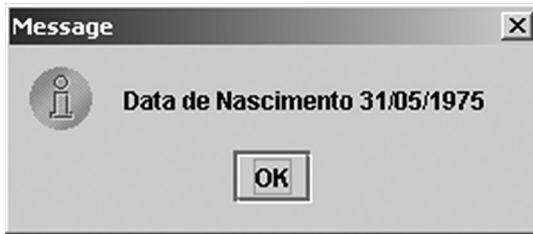


Figura 23 – Tela de saída da Data

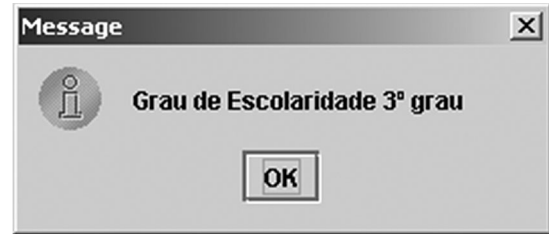


Figura 24 – Tela de saída do Grau Esc.

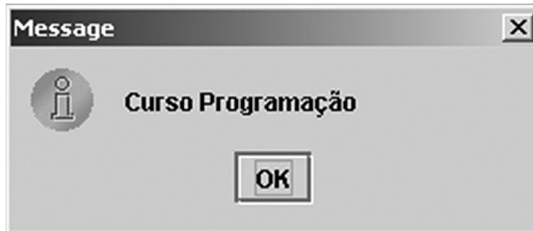


Figura 25 – Tela de saída do Curso



Síntese

Nesta unidade, você desenvolveu os primeiros programas (sem controle de fluxo) na linguagem Java, com entrada e saída de dados.

É importante que você pratique as informações passadas nessa unidade, ou seja, que você saiba como se declara variáveis de memória, como programa uma instrução de entrada de dados e como se programa uma instrução de saída de dados, tudo dentro da estrutura de classe da linguagem Java.



Atividades de auto-avaliação

- 1) Desenvolver em Java TODOS os exercícios de auto-avaliação da seção 1 da Unidade 5 da disciplina de Lógica de Programação I.



Saiba mais

FURGERI, Sergio. **Java 2 Ensino Didático**. São Paulo: Érica, 2002.

Implementando programas em Java com controle de fluxo



Objetivo de aprendizagem

- Implementar programas em Java com Controle de Fluxo (condicional e repetição).



Seção de estudo

Seção 1 Estruturas de Controle de Fluxo em Java



Para início de conversa

Agora que você está um pouco mais experiente no desenvolvimento de programas vai começar a implementar programas em Java com controle de fluxo, ou seja, programas com SE, ENQUANTO ou PARA.

Vamos começar?

SEÇÃO 1 – Estruturas de controle de fluxo em Java

Vamos apresentar aqui as estruturas que nos permitem controlar o fluxo de execução de programa (estruturas para processamento condicional e de repetição) em Java, mas antes, temos que aprender a delimitar blocos e conceituar o escopo.

Um bloco é uma série de linhas de código situadas entre um abre e fecha chaves ({ }). Podemos criar blocos dentro de blocos. Dentro de um bloco temos um determinado escopo, que determina a visibilidade e tempo de vida de variáveis de memória.

Por exemplo:

```
{  
  int x = 10;  
  // aqui eu tenho acesso ao x  
  {  
    int z = 20;  
    // aqui eu tenho acesso ao x e ao z  
  }  
  // aqui eu tenho acesso ao x; o z está fora do escopo  
}
```

Assim, é permitida a declaração de variáveis com um mesmo nome, desde que elas não estejam compartilhando o mesmo escopo. A definição dos blocos ajuda a tornar o programa mais legível e a utilizar menos memória porque as variáveis somente são declaradas no momento que serão utilizadas.

Estrutura de controle para processamento condicional

Em Java a estrutura condicional **se – senão** possui a sintaxe mostrada abaixo:

```
if (expressão)
    comando ou { bloco }
else // opcional
    instrução ou { bloco } // opcional
```

ATENÇÃO: quando houver mais de 1 comando ou linha de instrução dentro do if ou do else, você deve delimitar o início e o fim desse bloco de instruções com { (abre chave) e } (fecha chave)

Vamos trabalhar com alguns exemplos de programas envolvendo estrutura condicional.

Estrutura condicional simples

```
se numero > 20 então
    numero ← numero/2
fim-se
```

Em Java a sintaxe dessa estrutura condicional simples é a seguinte:

```
if (numero > 20)
    numero = numero / 2;
```

Note que a condição ou expressão lógica depois do **if** está envolvida entre parênteses. Toda a condição depois de um if deve obrigatoriamente estar entre parênteses. Nesse exemplo temos somente uma instrução “dentro” do if:

```
numero = numero / 2.
```

Nesse caso, não é necessário delimitar o bloco de instruções dentro do if com { } (abre e fecha chave). A estrutura de controle **if** em Java não precisa ser fechada ou terminada.

Estrutura condicional composta

```
se numero > 20 então  
    numero ← numero/27  
    senão  
        numero ← numero/4  
fim-se
```

Em Java, a sintaxe dessa estrutura condicional composta é a seguinte:

```
if (numero > 20)  
    numero = numero / 2;  
else  
    numero = numero / 4;
```

Nesse caso, também não é necessário delimitar o bloco de instruções tanto dentro do **if** como dentro do **else** com **{ }** (abre e fecha chave), pois tanto dentro de um como dentro de outro só existe uma linha de instrução.



Quando isso será necessário?

Observe o mesmo exemplo estendido:

```
if (numero > 20){  
    numero = numero / 2;  
    valor=valor + numero;  
}else  
    numero = numero / 4;
```

Nesse caso, dentro da instrução **if** temos duas instruções agora. Quando existirem dentro de um **if** ou dentro de um **else** existir mais de uma instrução, você deve colocar essas instruções dentro de um bloco delimitado por **{ }** (abre e fecha chave).

Exemplo de um algoritmo completo em pseudocódigo:

```

Início
    NOTA1, NOTA2, NOTA3, MEDIA: numérico
    Leia NOTA1, NOTA2, NOTA3
    MEDIA ← (NOTA1 + NOTA2 + NOTA3)/3
    se MEDIA > 7.0 então
        escreva "A média é maior que 7.0"
    senão se MEDIA = 7.0 então
        escreva "A média é igual a 7.0"
    senão se MEDIA < 7.0 então
        escreva "A média é menor a 7.0"
Fim

```

Esse algoritmo na linguagem Java fica assim:

```

0  import javax.swing.*;
1  class Media{
2      public static void main(String args[])
3      {
4          double NOTA1, NOTA2, NOTA3, MEDIA;
5          NOTA1=Double.parseDouble(JOptionPane.showInputDialog("Entre com a Nota 1"));
6          NOTA2= Double.parseDouble(JOptionPane.showInputDialog("Entre com a Nota 2"));
7          NOTA3= Double.parseDouble(JOptionPane.showInputDialog("Entre com a Nota 3"));
8          MEDIA=(NOTA1+NOTA2+NOTA3)/3;
9          if (MEDIA > 7.0)
10             JOptionPane.showMessageDialog(null,"A média é maior que 7.0");
11         else
12             if (MEDIA == 7.0)
13                 JOptionPane.showMessageDialog(null,"A média é igual a 7.0");
14             else
15                 JOptionPane.showMessageDialog(null,"A média é menor que 7.0");
16                 System.exit(0);
17     }
18 }

```

Vamos analisar linha a linha:

Java possui uma biblioteca de classes que estão prontas e disponíveis para os programadores usarem em seus programas. Quando você instala o j2sdk essa biblioteca de classes é instalada também.

- **Linha 0:** essa instrução informa ao compilador o path (caminho) de algumas classes da biblioteca de classes da linguagem Java usadas nesse programa. As classes dessa **biblioteca** que estamos usando são: JOptionPane e Double.
- **Linha 1:** todo o programa desenvolvido em Java é chamado de classe ou **class**. Toda classe deve ter um nome. O nome do nosso programa é Media. Logo, a primeira linha deve ser a declaração da **classe (class)** seguido do nome do programa e o símbolo de início da class };
- **Linha 2:** todo o programa (conjunto de instruções) feito em Java deve estar dentro de pequenos módulos também conhecidos como métodos. Pelo menos um método deve existir (a princípio) em nossos programas (**class**) Java. Esse método é chamado de **main** (em inglês: principal). Todo o conjunto de instruções que corresponde ao programa deve estar (a princípio) dentro do método **main** e ele SEMPRE deve ser declarado da maneira como está na linha 2.
- **Linha 3:** instrução de início do programa. O símbolo que indica início de bloco em Java é {.
- **Linha 4:** declaração de quatro variáveis de memória do tipo double (real).
- **Linha 5:** instrução de entrada de dados em Java. Programamos uma instrução de entrada de dados em Java com a JOptionPane.showInputDialog (“Entre com a Nota 1”). Essa instrução faz com que apareça uma

janelinha ou caixa de diálogo na tela com um campo para o usuário do programa digitar a primeira nota. Lembre-se que qualquer valor digitado nessa caixa de diálogo sempre será do tipo String. Se o valor é do tipo String, então ele não pode ser armazenado, atribuído diretamente a variável `NOTA1`. Por causa disso, programamos a instrução `Double.parseDouble(..)` que irá converter o tipo do dado digitado na caixa de diálogo para o tipo `double` (real). Depois da nota ser convertida para `double`, ela pode ser armazenada na variável `NOTA1` que só aceita dados do tipo `double`.

- **Linhas 6 e 7:** a mesma explicação se repete para as linhas 4 e 5.
- **Linha 8:** cálculo da média e atribuição da média a variável `MEDIA`.
- **Linha 9:** começo da instrução `if` que testará as três possibilidades da variável `MEDIA`, se a variável `MEDIA` for maior que 7.
- **Linha 10:** é executado somente essa instrução que escreve na tela a frase: “A média é maior que 7.0”. A instrução de escrita ou saída de dados em Java que estamos utilizando é: `JOptionPane.showMessageDialog` (`null`, “A média é maior que 7.0”). Ela faz com que apareça na tela uma janelinha ou caixa de diálogo com a frase que foi programada entre “ ”.
- **Linhas 11 a 15:** segue com as outras possibilidades de verificação de `MEDIA`
- **Linha 16:** instrução do método `main` }.

- **Linha 17:** instrução de fim da class }.

Estrutura de controle para processamento com repetição

1. *enquanto/faça/fim-enquanto*

Em Java a estrutura **enquanto/faça/fim-enquanto** possui a sintaxe mostrada abaixo:

```
while (expressão)
instrução ou { bloco }
```

Na sintaxe de Java essa estrutura não tem o comando equivalente ao **faça** e nem ao **fim-enquanto**.

Você já estudou que nesse tipo de estrutura de repetição a expressão ou condição lógica é avaliada uma vez antes da execução da primeira instrução. Caso seja verdadeira, a instrução ou bloco é executado. Ao final da instrução ou bloco, o fluxo de execução do programa volta (*loop*) e a expressão é avaliada novamente. Se for necessário mais de uma instrução, será preciso colocar o bloco das instruções entre { } .

Vamos trabalhar com um exemplo de programa envolvendo estrutura de repetição.

```
Início
NOME: literal
NUMALUNOS: numérico
NUMALUNOS ← 1
    enquanto NUMALUNOS <= 100 faça
        leia NOME
        escreva "Aluno cadastrado: " NOME
        {incrementa-se a variável NUMALUNOS a cada repetição}
        NUMALUNOS ← NUMALUNOS + 1
    fim-enquanto
Fim
```

Em Java a implementação desse programa que utiliza estrutura de repetição é a seguinte:

```

1  import javax.swing.*;
2  class Alunos{
3      public static void main(String args[])
4      {
5          String nome;
6          int NUMALUNOS;
7          NUMALUNOS = 1;
8          while (NUMALUNOS <= 100) {
9              NOME=JOptionPane.showInputDialog("Entre com o Nome");
10             JOptionPane.showMessageDialog(null,"Aluno cadastrado"+NOME);
11             //incrementa-se a variável NUMALUNOS a cada repetição
12             NUMALUNOS = NUMALUNOS + 1;
13         }
14         System.exit(0);
15     }
16 }

```

Observe que depois do **while** temos a condição ou expressão lógica que será avaliada e ela está entre parênteses. Toda a condição depois de um **while** deve estar, OBRIGATORIAMENTE, entre parênteses.

Dentro da instrução **while** existem quatro instruções, ou seja, existe mais de uma instrução a ser repetida 100 vezes, portanto, essas instruções devem estar entre { }, que delimitam o início e o fim de bloco.

Vamos analisar linha a linha:

- **Linha 1:** Essa instrução serve para informar ao compilador, o “caminho” ou diretório onde algumas class usadas no programa, se encontram.
- **Linha 2:** declaração da class. O nome da class é Alunos.
- **Linha 3:** declaração do método main. Lembre-se que todas as instruções do nosso programa estarão a princípio dentro desse método main.
- **Linha 4:** Início do método main {. Esse símbolo de início poderia estar no final da linha 3.

Lembre-se que **class** é sinônimo de programa em Java.

- **Linha 5:** declaração de uma variável de memória do tipo String (literal).
- **Linha 6:** declaração de uma variável de memória do tipo int (inteiro).
- **Linha 7:** a variável NUMALUNOS é inicializada, ou seja, recebe o valor inicial 1;
- **Linha 8:** a condição (NUMALUNOS <= 100) é avaliada. Se ela resultar no valor lógico true (verdadeiro), serão executadas as quatro instruções seguintes (linhas 9,10,12). Observe que, como existem mais de uma instrução dentro da estrutura de repetição, elas estão delimitadas por { }.
- **Linha 11:** essa instrução não será compilada porque está precedida do símbolo de comentário //. Tudo o que está depois desse símbolo é ignorado pelo compilador.
- **Linha 13:** fim do bloco de instruções da estrutura de repetição.
- **Linha 14:** instrução de fim do método main }.
- **Linha 15:** instrução de fim do programa }.

2. para/faça/fim-para-faça

Em Java a estrutura **para/faça/fim-para-faça** possui a sintaxe mostrada abaixo:

```
for (inicialização da variável de controle ; condição de repetição ;  
    incremento da variável de controle)  
    instrução ou { bloco }
```

Esse tipo de estrutura de repetição, conhecida também como “repetição controlada”, deve ser usada quando se sabe o número exato de vezes que uma instrução ou bloco de instruções deve ser executado (repetido).

Depois do **for** estão as informações para controle dessa repetição. São três blocos de informações separados por ; (ponto e vírgula).

O primeiro é responsável por declarar e inicializar a variável de controle usada para controlar essa repetição, por exemplo, `int i=1`. A variável de controle é do tipo `int`, se chama `i` e é inicializada com o número 1.

O segundo trata da própria condição de repetição, por exemplo, `i<=100`. Isso significa que enquanto a variável `i` for menor ou igual a 100 a instrução ou bloco será executado. (repetido).

O terceiro trata do incremento (soma) da variável de controle a cada vez que uma instrução ou bloco for executado, por exemplo, `i++`. Essa instrução equivale a `i=i+1`, ou seja, a cada vez que a instrução ou bloco for executado, a variável `i` é incrementada em 1.

Vamos trabalhar com um exemplo de programa envolvendo estrutura de repetição controlada.

```

Início
para l de 1 até 100 passo 1 faça
    leia NOME
    escreva "Aluno cadastrado: " NOME
fim-para
Fim
  
```

Em Java, a implementação desse programa que utiliza essa estrutura de repetição é a seguinte:

```

1  import javax.swing.*;
2  class Repete{
3      public static void main(String args[])
4      {
5          String NOME;
6          for (int l=1; l<=100 ; l++)
7          {
8              NOME=JOptionPane.showInputDialog("Entre com o Nome");
9              JOptionPane.showMessageDialog(null,"Aluno cadastrado"+NOME);
10         }
11     }
12 }
  
```



Síntese

Nessa unidade você avançou um pouco mais e passou a desenvolver programas com controle de fluxo de execução, ou seja, usando estruturas que controlam o fluxo de execução dos programas como a estrutura se (if), a estrutura enquanto (while) e a estrutura para-faça (for). Com elas você irá conseguir desenvolver a maioria dos programas.



Atividades de auto-avaliação

- 1) Desenvolver em Java todos os exercícios de auto-avaliação da seção 2, da Unidade 5, da disciplina de Lógica de Programação I (Exercícios com estrutura condicional).

- 2) Desenvolver em Java todos os exercícios de auto-avaliação da Unidade 6, da disciplina de Lógica de Programação I (Exercícios com estrutura de repetição).



Saiba mais

Tutorial sobre Variáveis primitivas e Controle de fluxo

<http://www.caelum.com.br/download/fj-joo/fj-2.pdf>

Desenvolvendo programas modularizados em Java



Objetivos de aprendizagem

- Rever o conceito de modularização de programas.
- Entender o conceito de procedimento.
- Implementar programas modularizados em Java.



Seção de estudo

Seção 1 Modularização de programas em Java



Para início de conversa

Você chegou ao final das unidades relacionadas especificamente à linguagem Java nessa disciplina. Até o momento, você aprendeu todas as informações básicas, porém muito importantes sobre a linguagem Java. Isso permitirá que você entenda e aplique (desenvolvendo programas) os conceitos do paradigma de programação orientado a objeto o qual veremos na próxima unidade.

Na disciplina de Lógica II, você estudou como modularizar programas somente através de funções. Para completar a parte prática mais intensa dessa disciplina, você estudará agora, nesta unidade, como modularizar programas em Java e também uma nova maneira de modularizar um programa através de procedimentos. Vamos continuar?

SEÇÃO 1 - Modularização de programas em Java

Na Unidade 5 da disciplina de Lógica de Programação II, você estudou como modularizar programas. Vamos lembrar?

A **modularização** é importante porque divide o problema (programa) em partes menores permitindo a reutilização de algumas partes do mesmo programa ou em outros programas. A divisão de um programa em partes menores facilita a manutenção do programa, uma vez que, se for necessário alterar alguma parte do código, isso será feito em uma só parte. Com a modularização, o programa fica mais legível e organizado.

Em Lógica de Programação II você viu que um programa pode ser modularizado (dividido) em partes chamadas de funções e que essas funções podem receber valores ou parâmetros e retornar algum valor também.

Isso está correto, mas agora você vai aprender que um programa pode também ser dividido em partes conhecidas como PROCEDIMENTOS.

Portanto, um programa pode ser dividido em partes conhecidas como **FUNÇÕES** e **PROCEDIMENTOS**.



O procedimento é parecido com a função, ou seja, é uma parte bem delimitada do programa e pode receber valores como parâmetros, mas ele **NÃO PODE RETORNAR NENHUM VALOR**, enquanto que uma função **SEMPRE** deve retornar algum valor.

Partindo dessa diferença entre **procedimento** e **função** você deve escolher como modularizar o seu programa. É recomendado colocar trechos de código que envolvam cálculos ou algum processamento, onde sempre se chegue a um único resultado dentro de funções, já que função, sempre deve retornar um único valor. Exemplo: soma de dois números, cálculo da área do retângulo, verificar se um determinado valor existe num vetor, etc. Os demais trechos de código, nos quais não se acha um único valor, devem ser colocados dentro de procedimentos. Exemplo: escrever na tela um resultado ou frase, etc.

Um programa pode ser modularizado com procedimento e funções juntos. Vamos demonstrar isso utilizando o exemplo da Unidade 5 da disciplina de Lógica II. Mas, dessa vez, vamos implementar direto na linguagem Java.



```
import javax.swing.*;
class Calcula{
    public static void main(String args[])
    {
        int n1,n2, resultado;
        n1=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero 1"));
        n2=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero 2"));
        resultado = soma (n1,n2);
        JOptionPane.showMessageDialog(null,"Resultado"+resultado);
    }

    public static int soma(int n1, int n2)
    {
        return n1 + n2;
    }
}
```

Vamos analisar linha a linha:

Pacote ou package é o nome dado ao diretório ou pasta onde as classes da biblioteca de classes da linguagem estão agrupadas. Elas são agrupadas por funcionalidades em comum em pacotes com nomes determinados. Exemplos: Classes que representam objetos gráficos como janelas, botões, barras estão agrupados no pacote `javax.swing`.

- **Linha 1:** Essa instrução serve para informar ao compilador o “caminho”, diretório ou pacote (**package**) onde algumas class usadas no programa se encontram.
- **Linha 2:** declaração da class `Calcula`.
- **Linha 3:** declaração do método `main`. Observe que algumas instruções do programa estão dentro do método `main` (linha 5 a linha 9) e que existe um outro método nessa classe cujo nome é `soma` e que contem outras instruções. Portanto, nessas classes existem dois métodos: `main` e `soma`. Qual deles será chamado automaticamente quando esse programa for executado? O método `main`. Ele é o método que é chamado primeiramente, mesmo que existam outros métodos na classe.
- **Linha 4:** Início do método `main` {.
- **Linha 5:** declaração de três variáveis de memória do `int`.
- **Linha 6:** entrada de dados do primeiro número e armazenamento na variável `n1`.
- **Linha 7:** entrada de dados do segundo número e armazenamento na variável `n2`;
- **Linha 8:** **ATENÇÃO** para essa instrução. Nela está sendo chamada, invocada a função `soma`. Chamamos uma função pelo próprio nome dela. Após o nome, deve vir abre e fecha parênteses (). Se a função estiver esperando valores (veja na linha 12, a função está esperando dois valores do tipo `int`) eles devem ser passados para a função e isso deve ser feito dentro dos parênteses. Isso se chama passagem de parâmetros. É o que está sendo feito nessa linha, logo depois da chamada da função é abertos parênteses e dois valores separados por vírgula, são passados para a função `soma`. São os valores que foram digitados e armazenados nas variáveis `n1` e `n2`. Vamos supor que o usuário do programa digitou

o valor 2 (armazenado em n1) e o valor 3 (armazenado em n2), então são esses os valores que serão passados para a função soma e armazenados RESPECTIVAMENTE nas variáveis locais n1 e n2 da função soma. Como é uma função que está sendo chamada e uma função SEMPRE retorna um valor, ela NUNCA deve ser chamada isolada numa linha de programa e sim, precedida de uma instrução de atribuição como nessa linha, onde o resultado da chamada da função está sendo armazenado na variável resultado.

- **Linha 9:** instrução de saída de dados. O conteúdo da variável resultado é impresso na caixa de diálogo.
- **Linha 10:** instrução de fim do método main }
- **Linha 11:** linha em branco. É opcional. Recomenda-se para o código ficar mais legível.
- **Linha 12:** declaração da função soma. Java não usa a palavra function para declarar uma função. Toda função começa com a palavra function seguido do tipo de dado que está sendo retornado mais o nome da função. Como essa função soma dois dados inteiros, o tipo de dado do resultado da soma também é inteiro, por isso, logo depois de function vem o tipo int, seguido do nome da função. A função soma, espera receber dois valores ou parâmetros. Quando uma função recebe parâmetros, eles devem ser recebidos, armazenados em variáveis de memória declaradas dentro dos parênteses. Nesse caso, a função soma recebe dois parâmetros ou valores. O primeiro será armazenado dentro da variável local n1 e o segundo dentro da variável local n2. Deve-se repetir o tipo de cada variável declarada dentro dos parênteses. As variáveis declaradas dentro de um método, seja ele função ou procedimentos, são chamadas variáveis LOCAIS, ou seja, elas só podem ser acessadas dentro do próprio método onde foram declaradas. As variáveis n1 e n2 do método main são diferentes das variáveis n1 e n2 da função soma. Qualquer variável declarada dentro do método main não pode ser usada pela função soma e vice-versa.

- **Linha 13:** início da função {.
- **Linha 14:** nessa linha as variáveis n1 e n2 são somas e o resultado da soma é retornado para a linha dentro do método main que chamou a função soma, no caso, linha 8. Esse valor retornado será armazenado na variável resultado. Uma função sempre deve retornar UM E SOMENTE UM valor. Para retornar um valor use a palavra return.
- **Linha 15:** fim da função }.
- **Linha 16:** fim da classe }. Todos os métodos de um programa sejam eles procedimentos e funções, devem estar dentro de uma classe, ou seja, devem estar implementados antes do fim da classe.



Onde podemos implementar um procedimento nesse mesmo programa?

Você já observou que um procedimento é um bloco de programa parecido com uma função, mas ele não retorna nenhum valor.

Então, vamos pensar. No programa mencionado, qual a parte que pode ser modularizada em um procedimento?

A saída de dados, ou seja, a linha 9.

Vamos refazer o código anterior, agora com um procedimento.


```

1 import javax.swing.*;
2 class Calcula{
3     public static void main(String args[])
4     {
5         int n1,n2, resultado;
6         n1=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero 1"));
7         n2=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero 2"));
8         resultado = soma (n1,n2);
9         imprime("Resultado", resultado);
10    }
11
12    public static int soma(int n1, int n2)
13    {
14        return n1 + n2;
15    }
16
17    public static void imprime(String legenda, int valor)
18    {
19        JOptionPane.showMessageDialog(null, legenda + valor);
20    }
21 }
22

```

Vamos analisar algumas linhas que mudaram nesse programa:

Nessa nova versão do programa, o procedimento chamado **imprime** está sendo chamado na **linha 9**. Ao invés de programar a instrução de saída de dados para mostrar na tela a palavra “Resultado” seguido do conteúdo da variável de memória resultado, é chamado o procedimento imprime e passado para ele dois valores ou parâmetros: a palavra “Resultado” (note que a palavra está entre “” porque é um valor literal do tipo String) e o variável resultado (onde está armazenado o resultado da soma). Esses dois parâmetros são recebidos pelo procedimento imprime na **linha 17** e armazenados respectivamente na variável local do tipo String legenda e na variável local do tipo int chamada valor.

Na **linha 18** está o símbolo de início do procedimento {.

Na **linha 19** está a instrução de saída de dados JOptionPane.showMessageDialog (null, legenda + valor) que mostrará na tela o conteúdo da variável legenda seguido do conteúdo da variável valor.



Qual o conteúdo da variável `legenda`?

A palavra “Resultado” que será passada como parâmetro para o procedimento `imprime` e armazenada na variável local do tipo `String`, chamada `legenda`.



Qual o conteúdo da variável `valor`?

O valor armazenado na variável `resultado` no método `main()` que será passado como parâmetro para o procedimento `imprime` e armazenado na variável local do tipo `int` chamada `valor`. Não sabemos qual valor será. Isso dependerá na entrada de dados do usuário do programa, mas sabemos que será o resultado da soma de dois valores.

Na linha 20, está o símbolo de fim do procedimento `}`.



OBSERVE o cabeçalho da definição do procedimento na **linha 17**. Qual a diferença para o cabeçalho da definição da função na **linha 12**? Não é necessário olhar os parâmetros!

A diferença é que, como uma função **SEMPRE** deve retornar um valor, o tipo do valor retornado deve estar especificado no cabeçalho da função. Veja na **linha 12**, o tipo do valor retornado pela função `soma` é **`int`**, pois os dois valores somados (`n1` e `n2`) também são `int`. O tipo de retorno da função deve estar antes do nome da função.

Como o procedimento **NUNCA** retorna um valor, a palavra **`void`** deve ser especificada no lugar do tipo, ou seja, antes do nome do procedimento.



Relembrando: Como identificar em um código Java o que é função e o que é procedimento?

Olhe o cabeçalho de definição de ambos. A função deve sempre ter um tipo de retorno e o procedimento sempre deve ter a palavra void que indica que nenhum valor será retornado.



Síntese

Nessa unidade você aprendeu a desenvolver programas modularizados, isto é, divididos em módulos, chamados de procedimentos ou funções. Aprendeu, também, a diferença entre um procedimento e uma função. Esse entendimento será muito importante para as unidades futuras, principalmente quando entrarmos na unidade de orientação a objetos. Modularizar programas é importante porque facilita a manutenção do código (alterações futuras) e torna os programas mais legíveis, mais organizados.



Atividades de auto-avaliação

- 1) Faça um programa em Java que leia 3 números e calcule a sua média. O programa deve ser modularizado. Utilize uma função para calcular e retornar a média dos números. Essa função deve receber os três números como parâmetro.

- 2) Crie um programa em Java que leia o raio de uma esfera (do tipo real) e passe esse valor para a função `volumeEsfera`. Essa função deve calcular o volume da esfera na tela e retornar o seu valor.

Para o cálculo do volume deve ser usada a seguinte fórmula:

$$\text{Volume} = (4.0 / 3.0) * \text{PI} * \text{raio}^3$$

- 3) Faça um programa em Java em que o usuário entre com um valor de base e um valor de expoente. O programa deve calcular a potência. O programa deve ser modularizado.

Você deve decidir se a subrotina será um procedimento ou uma função.

A formula é: $\text{base}^{\text{expoente}}$

Introdução à Programação Orientada a Objeto (POO)



Objetivos de aprendizagem

- Conhecer a história do paradigma de programação orientada a objetos.
- Aprender o conceito de Classe e Objeto.
- Implementar uma Classe e criar Objetos a partir dela.



Seções de estudo

Seção 1 Paradigma de Programação Orientada a Objeto

Seção 2 Implementando uma Classe e criando Objetos a partir dela



Para início de conversa

Nessa unidade você será apresentado ao mundo da orientação a objetos. A orientação a objetos é uma maneira (paradigma) de se pensar e desenvolver programas. Ao desenvolver um programa orientado a objeto, você deve pensar em como a situação que o programa está automatizando funciona no mundo real, ou seja, quais os objetos envolvidos na situação e como eles se relacionam.

A partir disso você deve se preocupar em modelar e implementar o programa de maneira que ele espelhe a situação existente no mundo real, ou seja, deve criar e manipular as representações de objetos existentes no mundo real.

Nessa unidade, você aprenderá um pouco da história, dos objetivos da orientação a objetos e aprenderá dois conceitos mais importantes da orientação a objetos: Classe e Objeto.



Links Interessantes

<http://www.vivaolinux.com.br/artigos/impressora.php?codigo=2501>

Entenda paradigma de programação como um estilo, uma maneira de se desenvolver um programa. Existem outros paradigmas de programação como veremos adiante.

SEÇÃO 1 - Paradigma de Programação Orientada a Objeto



Paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver os problemas dentro dessas fronteiras. Os paradigmas influenciam nossa percepção; ajudam-nos a organizar e coordenar a maneira como olhamos para o mundo. (Definição literal)

Um pouco de história

O conceito de programação orientada a objeto não é algo novo. No final da década de 60, a linguagem Simula67, desenvolvida na Noruega, introduzia conceitos hoje encontrados nas linguagens orientadas a objetos. Em meados de 1970, o Centro de Pesquisa da Xerox (PARC) desenvolveu a linguagem Smalltalk, a primeira totalmente orientada a objetos. No início da década de 80, a AT&T lançou a Linguagem C++, uma evolução da linguagem de programação C em direção à orientação a objetos.

Atualmente, um grande número de linguagens incorpora características de orientação a objeto, tais como Java, Object Pascal, Python , etc.

Orientação a objeto – Conceitos básicos

O paradigma de programação orientada a objeto é baseado em alguns conceitos que definem uma forma de criar programas para computadores.

A filosofia principal desse paradigma de programação é solucionar um problema (via programação) através da representação computacional dos objetos existentes nesse problema (objetos do mundo real), usando para isso os conceitos mencionados acima. Com isso, essa maneira de se desenvolver programas fica mais próxima das situações, dos problemas como eles acontecem no mundo real.

Será um pouco difícil entender todas as vantagens de OO (Orientação a Objetos). Agora, portanto, vamos mencionar apenas duas:

- É mais fácil conversar com o cliente que pediu o software se falarmos com objetos que existem no mundo dele (o mundo do software fica mais perto do mundo real).
- O software feito com OO pode ser feito com maior qualidade e pode ser mais fácil de escrever e, principalmente, alterar no futuro.



Já que a programação orientada a objeto se baseia no conceito de objetos vamos entender um pouco mais sobre objetos?

O nosso mundo está repleto de objetos, sejam eles concretos ou abstratos. Qualquer tipo de objeto possui **atributos** (características) e **comportamentos** que são inerentes a esses objetos.



Exemplos de objetos **CONCRETOS**

1) **CANETA**

Atributos de qualquer Caneta

- altura
- espessura
- cor

Nesse caso, o nosso objeto **Caneta** tem os seguintes dados em cada característica:

- altura: 10 cm
- espessura: 2 cm
- cor: rosa

Comportamentos de qualquer Caneta

- desenhar
- etc.

2) **PESSOA**

Atributos de qualquer Pessoa

- altura
- peso
- idade

Nesse caso, o nosso objeto **Pessoa** tem os seguintes dados em cada característica:

- altura: 1,80
- peso: 70
- idade: 25

Comportamentos de qualquer Pessoa

- andar
- sorrir
- ler
- etc.



Exemplos de objetos que **NÃO SÃO CONCRETOS**:

1) RETÂNGULO

Atributos de qualquer Retângulo

- base
- altura

Nesse caso, o nosso objeto **Retângulo** tem os seguintes dados em cada característica:

- base: 10
- altura: 5

Comportamentos de qualquer Retângulo

- calcular área
- calcular perímetro
- etc.

2) DISCIPLINA

Atributos de qualquer Disciplina

- código da disciplina
- nome da disciplina

Nesse caso, o nosso objeto **Disciplina** tem os seguintes dados em cada característica:

- código da disciplina: 1
- nome da disciplina: Álgebra Linear

Comportamentos de qualquer Disciplina

- listar o nome da disciplina
- etc.

Disciplina nesse caso refere-se a uma disciplina oferecida numa universidade.

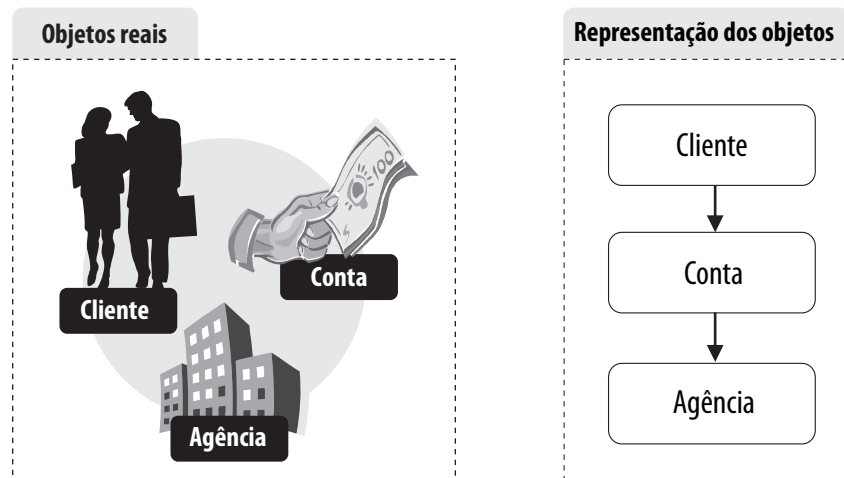
Essa operação ficará mais clara (concreta), quando avançarmos um pouco mais nos conceitos de orientação a objetos.

Para usar objetos na programação, primeiro precisamos distinguir entre um **objeto real** e a **representação de um objeto**. No desenvolvimento de programas, sempre trabalhamos com representações de objetos. Essa representação precisa refletir os objetos reais, ou seja, objetos existentes no problema do mundo real que queremos solucionar.

Por exemplo, um sistema de conta corrente não manipula diretamente os clientes, contas e cheques (esses são os objetos reais do problema).

Em vez disso, o software deve criar representações desses objetos, com os mesmos atributos e comportamentos dos objetos do mundo real. Essa representação é chamada de **ABSTRAÇÃO**.

Segundo o dicionário Aurélio **abstração** significa: "Ato de separar mentalmente um ou mais elementos de uma totalidade complexa (coisa, representação, fato), os quais só mentalmente podem subsistir fora dessa totalidade".



Vamos aprender mais sobre representação de objetos?

Para entender isso, vamos voltar ao exemplo do sistema de conta de corrente.

No mundo real existem vários objetos Cliente, vários objetos Conta e vários objetos Agência (pode ser somente um), ou seja, muito provavelmente não existirá somente um cliente, uma conta e uma agência.

Todos os objetos cliente possuem o mesmo conjunto de atributos, por exemplo, todos os objetos cliente possuem nome e endereço, assim como todos os objetos conta possuem um número, um saldo, um histórico de transações e todos os objetos agência possui um número e um nome.

Assim como os atributos, todos os objetos de um mesmo tipo compartilham do mesmo conjunto de **comportamentos**, por exemplo, todos os objetos Cliente fazem depósitos e saques.

Apesar de cada objeto ter o mesmo conjunto de atributos, cada objeto possui valores próprios para cada atributo, ou seja, cada objeto é único.

Essa explicação para objetos do mundo real vale para quando trabalhamos com a representação desses objetos no contexto de desenvolvimento desse sistema (programação).

Portanto, podemos perceber que o sistema de conta corrente (programa de conta corrente) será composto de vários objetos, cada qual com um conjunto de atributos e comportamento em comum (se forem do mesmo tipo) e com valores próprios nos seus atributos. A figura abaixo ilustra os vários objetos cliente, conta e agência.

Joao: Cliente	Ana: Cliente	
nome: Joao endereço: Rua J	nome: Ana endereço: Rua A	
9916:Conta	8761:Conta	Centro:Agencia
numero: 9916 saldo: 1000,00 transação: débito	numero: 8761 saldo: 500,00 transação: crédito	nome: Centro numero: 123

O termo **instância** é frequentemente usado para descrever um objeto com valores próprios em seus atributos. Por exemplo, observe a figura acima:

- Ana é uma instância de Cliente
- 9916 é uma instância de Conta

Todo objeto deve ter:

1. Estado
2. Comportamento
3. Identidade

1) Estado

É representado pelos valores dos atributos de um objeto em um determinado instante do tempo. O estado do objeto usualmente muda ao longo da existência do objeto.



O valor dos atributos nome e endereço do objeto João podem mudar ao longo da existência desse objeto. As figuras abaixo ilustram isso: no estado 1 do objeto João, o valor do atributo endereço era Rua J, no estado 2 o valor desse atributo é Rua XX.

Estado 1

Joao: Cliente
nome: Joao endereco: Rua J

Estado 2

Joao: Cliente
nome: Joao endereco: Rua J

2) Comportamento

Determina como um objeto age e reage: suas modificações de estado e interações com outros objetos.

O comportamento é determinado pelo *conjunto de operações* ou *métodos* que o objeto pode realizar. Operações ou métodos são algo que você pede para o objeto fazer, como fazer um depósito (comportamento de qualquer objeto do tipo Conta).



Você deve estar achando estranho: fazer um depósito é um comportamento de qualquer objeto do tipo Conta? Isso mesmo!

Fazer depósito é um comportamento de qualquer objeto do tipo Conta e não de qualquer objeto do tipo Cliente, como você deve ter pensado a princípio.

Isso acontece porque há diferenças grandes quando pensamos no comportamento de representações de objetos, ou seja, objetos do mundo do software.

Ao identificar comportamentos de objetos do mundo do software devemos levar em conta os seguintes princípios:

- **Primeiro princípio:** não vamos modelar todo o comportamento dos objetos. Exemplo: Clientes tomam café no mundo real, mas provavelmente não no software.
- **Segundo princípio:** os comportamentos freqüentemente são assumidos por objetos diferentes. Exemplo: No mundo real, quem faz um depósito? Um Cliente.

No software, quais objetos assumiriam a responsabilidade de fazer um depósito? Objetos do tipo Conta.



Por que isso?

A primeira grande regra de programação Orientada a Objeto é a seguinte:



“Comportamentos são associados aos objetos que possuem os atributos afetados pelo comportamento”.

Um depósito afeta duas coisas, do ponto de vista de um sistema bancário:

- O saldo de uma conta.
- O histórico de transações feitas a uma conta.

Como se pode ver, o depósito afeta apenas atributos de uma Conta. Portanto, esse comportamento está associado à Conta e não ao Cliente, embora, no mundo físico, seja o Cliente que faz o depósito.

Outros comportamentos associados aos objetos do tipo Conta: fazer um saque, informar seu saldo, etc.

3) Identidade

Refere-se à identificação do objeto. Como cada objeto é único, ou seja, tem seus próprios valores nos atributos, ele deve ser identificado por algum nome. Mesmo que seu estado seja idêntico ao de outro objeto ele tem identificação própria.

A identificação do objeto do tipo Agência mostrado na figura abaixo é Centro. Esse objeto poderia ter qualquer nome identificando-o.

Centro:Agencia
nome: Centro numero: 123

Agora você já sabe que qualquer programa orientado a objeto deve trabalhar com representações de objetos que reflitam as características e comportamento de objetos do mundo real e já aprendeu os principais conceitos relacionados a objetos.



Falta agora aprender como CRIAR a representação de um objeto dentro de um programa.

No sistema de conta corrente foram identificados, a princípio, 3 tipos ou grupos de objetos: *Cliente*, *Agência* e *Conta*.

Sabemos que poderão existir vários objetos de cada um desses tipos e todos eles irão possuir o mesmo conjunto de atributos e comportamentos do seu tipo.

Para criar uma representação de cada objeto dentro do programa, devemos antes criar uma estrutura, uma espécie de **molde** ou **template** que represente os atributos e comportamentos do tipo do objeto. Essa estrutura é chamada de **classe**.

Portanto, devemos ter uma classe Cliente, uma classe Agência e uma classe Conta.

É a partir dessa estrutura conhecida como classe que iremos criar as representações de objetos que precisamos para desenvolver o programa, ou seja, precisamos criar a estrutura (classe) apenas uma vez e, a partir dela, serão criadas as representações dos objetos.

**Atenção!**

Os conceitos de classe e objeto são os mais básicos e importantes da orientação a objetos. O correto entendimento desses conceitos é imprescindível para o entendimento dos demais conceitos da OO.

Vamos aprender mais conceitos sobre Classes e Objetos?

Uma **classe** é a descrição de atributos e comportamentos de um grupo de objetos com propriedades similares (atributos) e comportamento comum (operações ou métodos).

Por representar atributos e comportamento de um determinado tipo de objeto que existe no mundo real, uma classe é uma abstração da realidade, pois nela estão especificados somente parte dos atributos e comportamentos de um objeto do mundo real. Somente parte, porque não são todos os atributos e comportamentos dos objetos do mundo real que interessam ao sistema.

A figura a seguir, ilustra a representação gráfica das classes do sistema de conta corrente. Na primeira parte da figura está o nome da classe, na parte do meio, estão os atributos e, na parte final, está o comportamento (métodos ou operações). Observe que a classe Conta e Agencia ainda não estão com os comportamentos definidos. Faremos isso na medida que avançarmos nos conceitos de OO.

Cliente	Conta	Agencia
nome: String endereço: String cpf: int	numero: int saldo: double transação: String	codigo: int nome: String
alteraNome() alteraEndereco() alteraCpf() forneceNome() forneceEndereco() forneceCpf()		

Um **objeto** é uma instância de uma classe. Essa instância possui valores próprios em cada atributo definido na classe. As figuras a seguir ilustram a classe Cliente e os vários objetos (instâncias) da classe Cliente.

Classe Cliente

Cliente
nome: String endereço: String cpf: int
alteraNome() alteraEndereco() alteraCpf() forneceNome() forneceEndereco() forneceCpf()

Objetos (instâncias) de Cliente

Joao: Cliente
nome: Joao endereço: Rua J
Ana: Cliente
nome: Ana endereço: Rua A

A notação que estamos utilizando para representar classes e objetos é a utilizada pela **UML** (*Unified Modelling Language*) ou Linguagem de Modelagem Unificada. A UML não é uma linguagem de programação e sim, uma linguagem composta de vários diagramas (cada qual com seu conjunto de símbolos gráficos) que expressam como um sistema orientado a objetos deve funcionar. Ela é utilizada nas fases iniciais do desenvolvimento de um software, ou seja, antes da programação propriamente dita, com o objetivo de especificar e documentar o funcionamento do sistema. O diagrama que iremos utilizar é o **Diagrama de Classes** que especifica as classes do sistema e o relacionamento entre elas.



Agora que você já estudou a diferença entre CLASSE e OBJETO, vamos partir para a parte prática, ou seja, você vai aprender a implementar (programar) uma classe e a criar objetos a partir dela. Você continuará aprendendo outros conceitos nas próximas seções e unidades, porém, para que os conceitos não fiquem muito abstratos para você, vamos ver como isso funciona na prática de maneira paralela.

Para finalizar essa seção vamos fazer uma pequena revisão:

Um programa orientado a objeto é composto por vários objetos do mesmo tipo e de outros tipos que se relacionam. Chamamos esses objetos do mundo do software de representações de objetos.

Como num programa muito provavelmente existirão vários objetos do mesmo tipo, devemos criar uma estrutura ou *template* que represente os atributos e comportamentos desses objetos do mesmo tipo, essa estrutura é chamada de classe. Portanto classe é uma estrutura a partir da qual os objetos (do software) são criados.



Importante!

O entendimento dos conceitos de orientação a objetos que já vimos e, dos demais que estão por vir, são de vital importância para o desenvolvimento de programas orientado a objetos e sua utilização **independe** da linguagem utilizada para implementação dos programas. Quer dizer que, para desenvolver qualquer programa OO, é preciso aplicar esses conceitos na implementação do programa, utilizando para isso, qualquer linguagem de programação orientada a objetos.

SEÇÃO 2 - Implementando uma Classe e Criando Objetos a partir dela

A implementação de uma classe só tem sentido se, no contexto do desenvolvimento de um sistema orientado a objeto, for identificada a necessidade de existência de um conjunto de objetos com atributos e comportamentos em comum.

A partir do momento que esses objetos com atributos e comportamento em comum são identificados, já existe a necessidade de criar a estrutura (classe) que os defina.

Nessa seção, você irá aprender como implementar na linguagem Java uma classe e como criar objetos (instâncias) a partir dela. Pode-se falar também no termo “instanciar objetos” a partir de uma classe.

Não iremos neste momento demonstrar o processo de identificação de objetos num dado problema, isso será feito nas seções seguintes.

Veremos agora, como implementar a classe Cliente do Sistema de Conta Corrente citado anteriormente e como criar objetos cliente a partir dela.

A visão prática desses dois conceitos (Classe e Objeto) facilitará o entendimento dos demais conceitos.

Implementação da Classe Cliente

A seguir será apresentada uma implementação resumida e mais **didática** da classe Cliente.

.....
Didática porque tem como objetivo facilitar o entendimento nesse primeiro momento.

```
Linha 1  public class Cliente{
Linha 2      private String nome, endereco;
Linha 3      private int cpf;
Linha 4
Linha 5      public Cliente( ){
Linha 6          nome="";
Linha 7          endereco="";
Linha 8          cpf=0;
Linha 9      }
Linha 10     public void alteraNome(String snome){
Linha 11         nome=snome;
Linha 12     }
Linha 13
Linha 14     public void alteraEndereco(String sender){
Linha 15         endereco=sender;
Linha 16     }
Linha 17
Linha 18     public void alteraCpf(int icpf){
Linha 19         cpd=icpf;
Linha 20     }
Linha 21     public String forneceNome(){
Linha 22         return nome;
Linha 23     }
Linha 24
Linha 25     public String forneceEndereco(){
Linha 26         return endereco;
Linha 27     }
Linha 28
Linha 29     public int forneceCpf(){
Linha 30         return cpf;
Linha 31     }
Linha 32 }
```

Vamos analisar a implementação da classe Cliente linha a linha.



Relembrando: para desenvolver qualquer programa você deve editar o programa num editor de texto, salvá-lo num local conhecido e, logo após, compilá-lo. O mesmo processo deve ser feito com as classes que você desenvolver daqui para frente. Detalhe: classes que representam atributos e comportamento de objetos NÃO SÃO executadas.

- **Linha 1:** definição inicial da classe. Começa com a palavra **public** (veremos mais adiante qual o significado dessa palavra) seguida da palavra **class** e seguida do **nome da classe**.

Regras para Nomeação de Classes

Para se nomear classes, existem alguns requisitos que devem ser observados:

- Classes são nomeadas com um substantivo no singular
- O nome de uma classe deve iniciar com a **primeira letra maiúscula**;
- Não devem ser utilizados símbolos de sublinhado (" _ ") - nomes compostos por múltiplas palavras são organizados com todas as palavras juntas, onde a primeira letra de cada uma fica em maiúscula.

Exemplos: Aluno, Professor, FolhaPagamento.

- **Linhas 2 e 3:** definição dos atributos definidos para a classe. Na definição dos atributos deve-se utilizar a palavra **private** (veremos mais adiante o significado dessa palavra) seguido do tipo de dado de cada atributo e do nome do atributo. No caso da classe Cliente, os dois atributos nome e endereço devem armazenar dados do tipo String (literal) e o atributo cpf deve armazenar valores do tipo int (numéricos inteiros).

.....
Isso ficará mais claro no momento da explicação da criação do objeto a partir da Cliente.

Regras para Definição de Atributos

- Normalmente, atributos são colocados no início da definição da classe, depois do primeiro { . Também pode ser colocados bem no final, antes do último }.
- Devem começar com letra minúscula.

- **Linha 5:** depois da definição do nome da classe e dos atributos da classe, segue-se a definição dos comportamentos, no contexto da implementação, conhecidos como **métodos**. Por enquanto, os comportamentos ou métodos definidos para a classe `Cliente` são simples. Na medida em que formos avançando, mais comportamentos ou métodos poderão ser implementados. No momento, implementamos sete métodos: `Cliente()`, `alteraNome()`, `alteraEndereco()`, `alteraCpf()`, `forneceNome()`, `forneceEndereco()` e `forneceCpf`.

O primeiro método implementado se chama `Cliente()` e se enquadra na categoria dos **métodos construtores**. Um método construtor SEMPRE deve ter o mesmo nome da classe e não deve especificar no cabeçalho nenhum tipo de retorno, nem mesmo deve ser utilizada a palavra `void` (o que acontece nos demais métodos que não retornam nada – métodos do tipo procedimento). Todo o método construtor só é executado no momento em que um objeto dessa classe é criado (instanciado). Vamos saber mais sobre métodos construtores no momento da explicação da criação de objetos. Saiba por enquanto que é importante (não obrigatório) que cada classe tenha ao menos um método construtor implementado.

Métodos também são sinônimos de subrotinas que você aprendeu em Lógica de Programação II e foi revisado na unidade anterior. Lembre-se que existem dois tipos: procedimento e funções. Procedimento é o método que não retorna valor e função é o método que retorna valor.

- **Linhas 6 a 8:** linhas de código do método construtor. O que se pode programar dentro de um método construtor?

Qualquer coisa que se quer que aconteça quando um objeto do tipo `Cliente`, no caso, for criado, porque o código que está dentro de um método construtor é executado sempre que um objeto desse tipo for criado.

Porém, normalmente se programa para que os **atributos do objeto** sejam inicializados. No caso do nosso método construtor, os atributos `nome`, `endereco` foram inicializados com Strings vazias ("") e o atributo `cpf` com o valor 0.

Fala-se em **atributos do objeto** porque tudo o que é definido na classe será atributo de comportamento de qualquer objeto criado (instanciado) a partir dela.

- **Linha 10:** nessa linha está o comportamento ou método chamado `alteraNome()`. Esse tipo de comportamento é típico de qualquer objeto do tipo `Cliente`, pois todo

o cliente pode ter seu nome alterado. No caso de um sistema OO, quem executa esse comportamento é o próprio objeto Cliente.

A implementação desse método ou comportamento é feita através de um procedimento (veja que o tipo de retorno é void, ou seja, essa subrotina não retorna nenhum valor) que recebe como **parâmetro** o novo nome do cliente. Esse valor é recebido em uma variável de memória do tipo String que chamamos de `snome`. O nome da variável de memória que receberá o novo valor do atributo `nome` pode ser qualquer um, porém essa variável deve ser declarada como sendo do tipo String, já que o valor dela será atribuído ao atributo `nome` que também é do tipo String (próxima linha).

- **Linha 11:** atribuição do novo nome do cliente ao atributo `nome`. O atributo `nome` recebe o conteúdo da variável `snome`, que contém o novo nome do cliente.
- **Linha 12:** fim do método `alteraNome()`;
- **Linhas 14 a 16:** implementação do método ou comportamento `alteraEndereco()`. Esse método altera o endereço de qualquer objeto Cliente.
- **Linhas 18 a 20:** implementação do método ou comportamento `alteraCpf()`. Esse método altera o cpf de qualquer objeto Cliente.
- **Linha 21:** nessa linha está o comportamento ou método chamado `forneceNome()`. Esse tipo de comportamento é típico de qualquer objeto do tipo Cliente, pois todo o cliente pode ser capaz de fornecer seu nome. No caso de um sistema OO, quem executa esse comportamento é o próprio objeto Cliente.

A implementação desse método ou comportamento é feita através de uma **função** (veja que o tipo de retorno é String) que retorna o nome do cliente, ou seja, o valor armazenado no atributo `nome` do objeto Cliente.

Programas com procedimento, funções e passagem de parâmetros (valores) foram implementados na última unidade.

Qualquer subrotina que retorne algum valor é chamada de função.

- **Linha 22:** retorno (usar palavra `return`) do conteúdo do atributo `nome`.
- **Linhas 25 a 27:** implementação do método ou comportamento chamado `forneceEndereco()`.
- **Linhas 29 a 31:** implementação do método ou comportamento chamado `forneceCpf()`.
- **Linha 32:** final da implementação da classe.

Depois de implementar a classe, você deve compilar, mas não deve executar. Esse tipo de classe não é executável, ela não possui **método `main()`**. Essa classe só possui a finalidade de ser uma estrutura ou *template* para a criação de objetos desse tipo.

Você lembra que todos os programas (classes) desenvolvidos nas unidades anteriores possuíam método `main()`? Toda a classe que possui método `main()` é uma classe que pode ser executada.

Criar objetos a partir da classe `Cliente`

Depois que a classe `Cliente` for implementada e compilada, ela está pronta para ser “usada”, ou seja, é possível criar objetos do tipo `Cliente` a partir dela.

Abaixo está a implementação de uma classe que possui um método `main()`, ou seja, é uma classe executável. Dentro desse método estamos criando um objeto do tipo `Cliente` e manipulando esse objeto.



Mas o que é de fato “criar um objeto”? Lembre-se sempre: agora estamos falando de objetos no contexto de um software. Você entenderá isso nas linhas seguintes.

```

Linha 1  public class UsaCliente{
Linha 2      public static void main(String args[]) {
Linha 3          Cliente c=new Cliente();
Linha 4          c.alteraNome("Ana");
Linha 5          c.alteraEndereco("Rua A");
Linha 6          c.alteraCpf(1235);
Linha 7          System.out.println(c.forneceNome());
Linha 8      }
Linha 9  }
```

`System.out.println()` é uma instrução de saída de dados. Ela é uma alternativa a instrução `JOptionPane.showMessageDialog(null,...)` usadas nas unidades anteriores. Porém, essa instrução não imprime o dado dentro de uma caixa de diálogo e sim, na janela do MS-DOS.

Vamos analisar a implementação da classe UsaCliente linha a linha:



Relembrando: para desenvolver qualquer programa você deve editar o programa num editor de texto, salvá-lo num local conhecido e logo após compilá-lo. O mesmo processo deve ser feito com as classes que você desenvolver daqui para frente. Detalhe: classes que representam atributos e comportamento de objetos NÃO SÃO executadas.

- **Linha 1:** definição inicial da classe. Começa com a palavra **public** seguida da palavra **class** e seguida do **nome da classe**.
- **Linha 2:** definição do método `main()`. Essa definição é sempre a mesma utilizada nos programas anteriores. Todo o programa que iremos implementar estará dentro do método `main()`.
- **Linha 3:** essa linha cria um objeto do tipo Cliente.



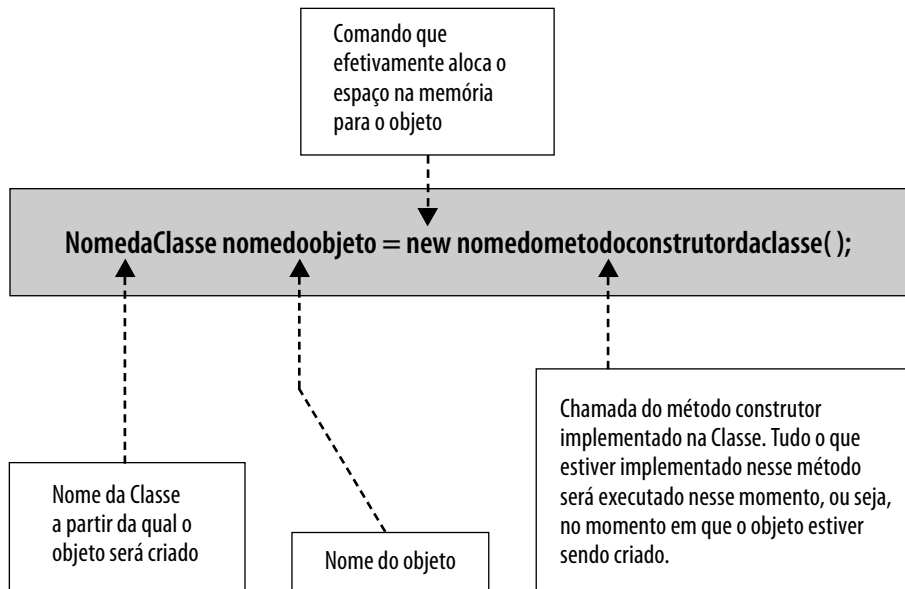
Criar um objeto é alocar (reservar) um espaço na memória do computador para armazenar valores para os atributos definidos na classe desse objeto.

A explicação de criar um objeto é similar à criação (declaração) de uma variável de memória. Quando declaramos uma variável de memória também estamos reservando espaço na memória para armazenar algum valor. Essa variável de memória deve ter um nome e deve ter um tipo de dados e nela só poderá ser armazenado valores desse tipo.

O mesmo acontece com a criação de um objeto. Ao criar um objeto, precisamos dar um nome (identificação) para ele e, nesse espaço de memória que estamos declarando, só poderão ser armazenados valores correspondentes aos atributos definidos no tipo (classe) desse objeto.

Essa explicação para criação de objeto é simples e será utilizada no momento para facilitar seu entendimento. Posteriormente, essa explicação será refinada.

Portanto, sempre que se quer criar um objeto deve-se usar a seguinte sintaxe:



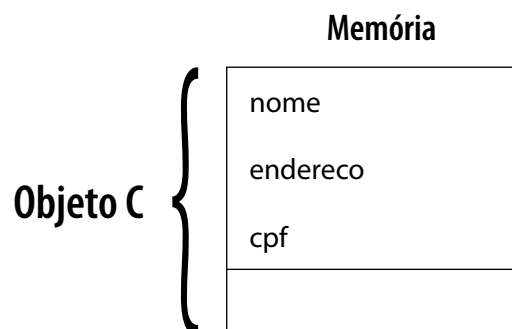
Após a execução dessa linha, um objeto, ou seja, uma área de memória é alocada.

Na instrução da linha 3, estamos criando um objeto chamado c. Note cuja a sintaxe é igual a definida acima para criação de objetos. Sempre que um objeto é criado, o método construtor definido na classe desse objeto é chamado (invocado) e, tudo o que está programado nesse método, é executado.

O que foi programado no método construtor da classe Cliente?

A inicialização de todos os atributos do objeto Cliente, portanto, ao criar o objeto cliente, todos os seus atributos já estarão inicializados.

A figura abaixo ilustra de maneira simples o objeto alocado na memória e seus atributos inicializados.



A partir desse momento, poderá ser chamado qualquer comportamento ou método definido na classe cliente. Isso será feito nas linhas seguintes.

- **Linha 4:** depois que o objeto é criado, podemos invocar seus comportamentos ou métodos. Nessa linha estamos chamando o método `alteraNome("Ana")` e passando como parâmetro o novo nome do cliente.

Observe que o método está **precedido** do nome do objeto e SEMPRE deve ser assim. Qualquer comportamento ou método deve ser chamado precedido do nome do objeto.

Quando se chama qualquer método definido em uma classe, o código implementando nesse método é executado. Nesse caso, o método irá alterar o nome do objeto `c`.

Qual o valor do atributo nome do objeto c ANTES da chamada desse método?

"" (String vazia) porque quando o objeto foi criado, esse valor foi armazenado lá. Isso aconteceu no momento que o método construtor foi chamado.

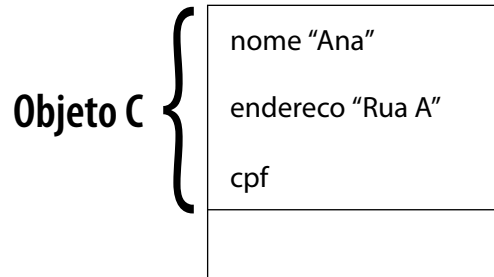
Qual o valor do atributo nome do objeto c DEPOIS da chamada desse método?

"Ana" porque foi esse o valor passado para o método. Lembre-se que o método `alteraNome(String snome)` recebe um nome como parâmetro e, esse nome, é armazenado no atributo `nome`.

Logo, após a execução dessa linha o objeto está com o seguinte estado:

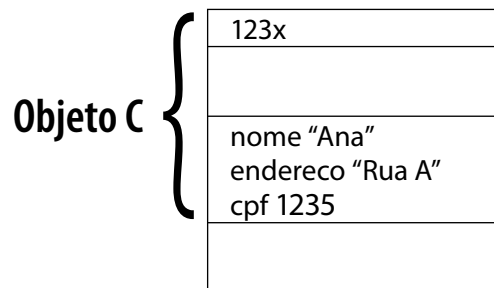
nome "Ana"
endereco
cpf

- **Linha 5:** chamada do método `alteraEndereco()` que irá alterar o endereço do objeto `c`. Logo após a execução dessa linha, o objeto está com o seguinte estado:



- **Linha 6:** chamada do método `alteraCpf()` que irá alterar o endereço do objeto `c`.

Logo, após a execução dessa linha, o objeto está com o seguinte estado:



- **Linha 7:** impressão do nome do cliente. Para isso chamamos o método `forneceNome()` do objeto `c`. Como esse método é uma função e toda a função retorna valor, ele não pode ser chamado isolado numa linha de instrução. Deve ser feito alguma coisa com o valor retornado do método, ou imprimir ou armazenar em outra variável de memória.
- **Linha 8:** fim do método `main()`.
- **Linha 9:** fim da classe `UsaCliente`.

Observação: Essa classe utiliza apenas 1 objeto, mas normalmente trabalharemos com programas que envolvem a criação de vários objetos. O procedimento explicado para criação e utilização de 1 objeto é o mesmo para vários objetos.



Síntese

Nessa unidade você, finalmente, foi apresentando ao “mundo” da orientação a objetos. Esse é um mundo cheio de conceitos que você terá que aprender para desenvolver programas orientado a objetos. Os primeiros conceitos apresentados nessa unidade são classe e objeto. Eles devem estar muito claros para você, pois são indispensáveis para a aprendizagem nas unidades seguintes.

Vamos partir para a **prática**?

.....

Revise os comando para compilação e execução de programas na linguagem Java nas unidades anteriores.



Atividades de auto-avaliação

- 1) Edite a classe Cliente no editor de texto (Bloco de Notas ou qualquer ambiente para desenvolvimento de programas Java) e salve na pasta CURSOWEBOO. Essa pasta deve ser criada anteriormente. Após a edição (digitação) compile a classe. O arquivo Cliente.class deve ter sido gerado.

- 2) Edite a classe `UsaCliente` no editor de texto. e salve na pasta `CURSOWEBOO`.

Após a edição, compile a classe. O arquivo `UsaCliente.classe` deve ter sido gerado. O ultimo passo é executar a classe. Dessa forma, você verá funcionando seu primeiro programa orientado a objeto desenvolvido na linguagem Java.

- 3) Cite exemplos de objetos do mundo real. Identifique atributos e comportamentos para esses objetos. Agrupe esses objetos nas suas respectivas classes. Faça isso usando a notação UML para classes e objetos.



Saiba mais

Deitel, H.M. e Deitel, P.J. **Java - Como programar**. Sexta Edição. Porto Alegre: Editora Pearson, 2005.

Lemay, Laura e Cadenhead, Rogers. **Aprenda em 21 dias Java 1.2**. Rio de Janeiro: Editora Campus, 1999.

Conceitos de Orientação a Objeto



Objetivos de aprendizagem

- Entender a diferença entre referência e Objeto.
- Criar programas com mais de um Objeto.
- Refinar a implementação de uma Classe.
- Aprender o conceito de Encapsulamento.



Seções de estudo

- Seção 1** Referência e Objeto
- Seção 2** Criando programas com mais de um Objeto
- Seção 3** Refinando a Criação de uma Classe
- Seção 4** Encapsulamento



Para início de conversa

Na unidade anterior você aprendeu dois conceitos muito importantes da OO: Classe e Objeto.

Nessa unidade você vai aprender a diferença entre **referência** e **objeto**, conceitos presentes no processo de criação de um objeto que precisam ser bem entendidos.

Estudará também como desenvolver outros programas (classes) que utilizam mais que um objeto e a refinar a implementação de uma classe, ou seja, a desenvolver uma classe que represente atributos e comportamentos de um objeto de maneira mais correta e, finalmente, aprenderá outro conceito muito importante da OO: o encapsulamento. Bons estudos!

SEÇÃO 1 - Referência e Objeto

Na última parte da unidade anterior, você aprendeu a criar um objeto a partir de uma classe. Isso foi feito na classe UsaCliente.java.

Esse foi o primeiro contato com o desenvolvimento completo de um programa orientado a objeto, ou seja, primeiramente foi modelada e implementada a classe Cliente e, posteriormente, utilizamos essa classe (na classe UsaCliente) para criar um objeto do tipo Cliente e manipular suas informações em memória.

Vamos revisar o código da unidade anterior e explicar dois conceitos importantes relacionados à criação de objetos.

Lembre-se que criar um objeto é alocar um espaço na memória que armazenará informações definidas no tipo (classe) desse objeto.

```

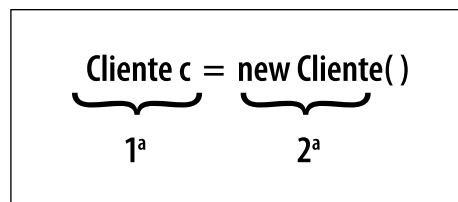
Linha 1  public class UsaCliente{
Linha 2      public static void main(String args[]) {
Linha 3          Cliente c=new Cliente( );
Linha 4          c.alteraNome("Ana");
Linha 5          c.alteraEndereco("Rua A");
Linha 6          c.alteraCpf(1235);
Linha 7          System.out.println(c.forneceNome( ));
Linha 8      }
Linha 9  }
```

Você lembra que a instrução da linha 3 cria um objeto (aloca espaço na memória) do tipo Cliente e que essa, é sintaxe para a criação de um objeto?

Isso está correto!

Porém, nessa linha de instrução existem dois conceitos importantes envolvidos na criação de um objeto: o conceito de **referência** e o próprio conceito de **objeto**.

Para entender isso, vamos separar essa instrução em duas partes, como mostra a figura abaixo:

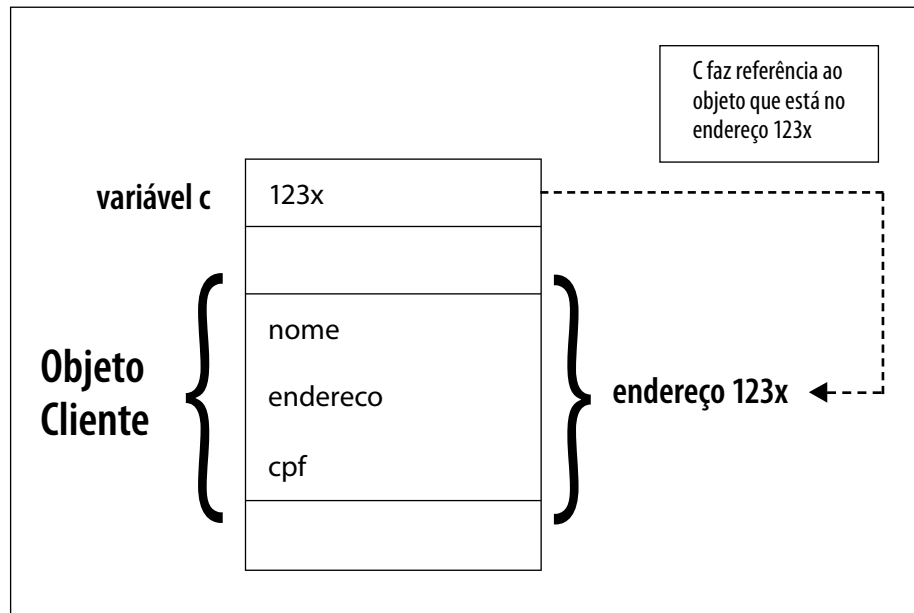


A 1ª parte está criando (declarando) uma variável de memória que pode armazenar uma referência (endereço de uma área de memória) para um objeto (área de memória) do tipo Cliente. Portanto, `c` armazena uma **referência** para um objeto do tipo Cliente.

A 2ª parte aloca uma área de memória para o objeto do tipo Cliente, isso quer dizer, que é reservada uma área de memória para armazenar os atributos definidos na classe Cliente. Quem faz essa alocação é a palavra **new**. Ela é responsável por identificar uma área livre na memória e reservar essa área. Essa área de memória reservada é o **objeto** do tipo Cliente.

Após reservar a área de memória do objeto o método construtor definido na classe desse objeto é chamado. No caso do exemplo acima, o método `Cliente()` está sendo chamado e dentro desse, foi implementado para que os atributos alocados dentro do objeto cliente sejam inicializados. Após, o endereço de memória da área alocada para o objeto Cliente é atribuído (armazenado) na variável `c`. A partir desse momento, `c` referencia um objeto Cliente.

A figura abaixo ilustra esse processo de criação de um objeto (linha 3) que na realidade envolve a criação de uma variável que fará referência a um objeto e o armazenamento nessa variável do endereço de memória do objeto criado.



As figuras abaixo ilustram o estado do objeto Cliente cuja identificação é c (o nome do objeto é c) à medida que as linhas do código da classe UsaCliente vão sendo executadas.

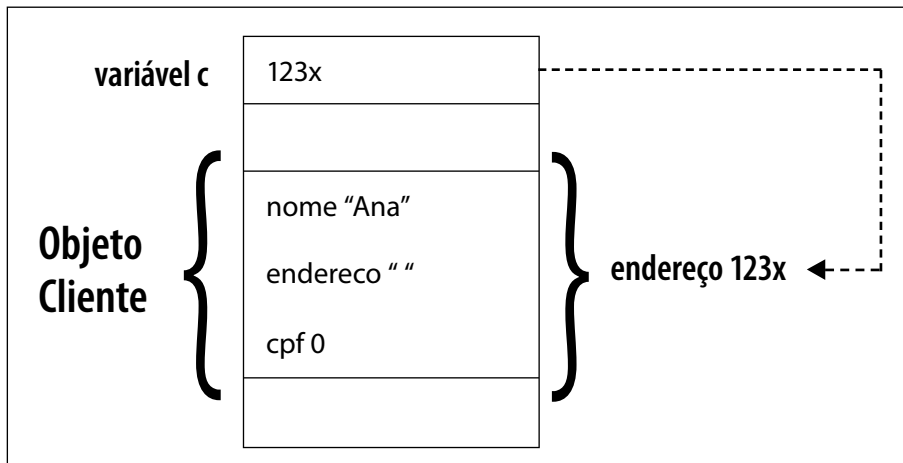
Linha 4: é chamado o método ou comportamento `alteraNome("Ana")` precedido da variável referência `c` e é passado um nome fixo como parâmetro. Como `c` "aponta" ou faz referência para o objeto (área de memória) que está no endereço `123x` e o método chamado armazena o valor recebido no atributo **nome** desse **objeto**, o atributo **nome** que será alterado é o existente nessa área de memória (objeto) que `c` referencia.

Veja a implementação dos métodos chamados na classe Cliente da unidade anterior.

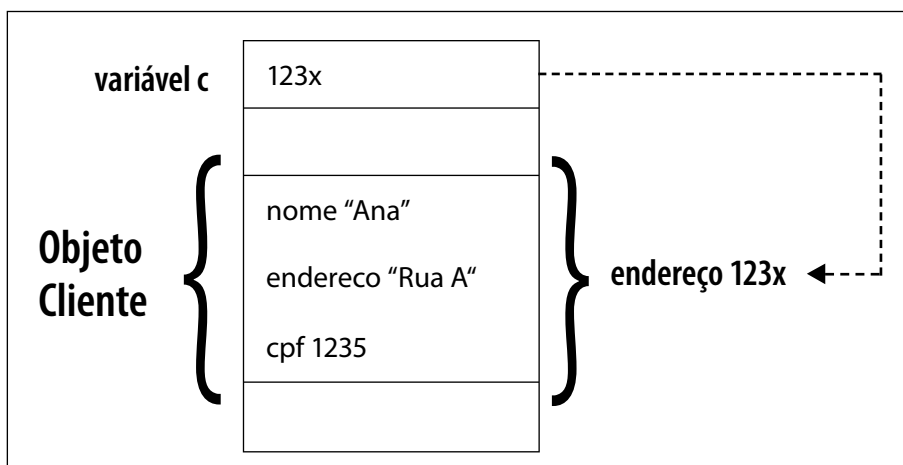
Vamos revisar essa chamada de método `alteraNome("Ana")`. A mesma explicação vale para os demais métodos `altera...()`.

Todo método `altera...()` implementado na classe Cliente espera receber um parâmetro. O método `alteraNome(String snome)` espera receber um valor (parâmetro) que será armazenado na variável local do método chamada `snome`. Por isso que, ao chamar esse método da classe UsaCliente, devemos passar um valor (parâmetro) para o método. No caso, "Ana" será armazenado na variável `snome` e logo em seguida será armazenado no atributo **nome** do objeto `c`.

O estado do objeto em memória fica assim:



Linhas 5 a 6: o mesmo se repete para as linhas 5 a 6.



Linha 7: nas linhas anteriores os atributos do objeto c foram todos preenchidos através da chamada dos métodos ou comportamentos altera....(). Agora queremos acessar, recuperar o valor de algum dos atributos do objeto c. Nessa linha, estamos mandando imprimir o conteúdo do atributo nome do objeto c. Para isso, chamamos o comportamento ou método desse objeto chamado forneceNome(). Como c faz referência para uma área de memória onde o objeto se encontra, o que será impresso é o valor do atributo nome desse objeto.

Todo o objeto (área de memória), para existir, deve estar associado a uma variável que armazene a sua referência (endereço da área de memória do objeto). Pode-se chamar essa variável de “**referência**”. No exemplo acima foi criada uma referência para um objeto do tipo Cliente.

Pode-se criar mais de uma referência para o mesmo objeto (se isso for necessário).

Vamos modificar o código da classe UsaCliente.java

```

Linha 1  public class UsaCliente{
Linha 2      public static void main(String args[]) {
Linha 3          Cliente c=new Cliente( );
Linha 4          Cliente b;
Linha 5          b=c;
Linha 6          c.alteraNome("Ana");
Linha 7          c.alteraEndereco("Rua A");
Linha 8          c.alteraCpf(75645);
Linha 9          System.out.println(b.forneceNome( ));
Linha 10      }
Linha 11  }

```

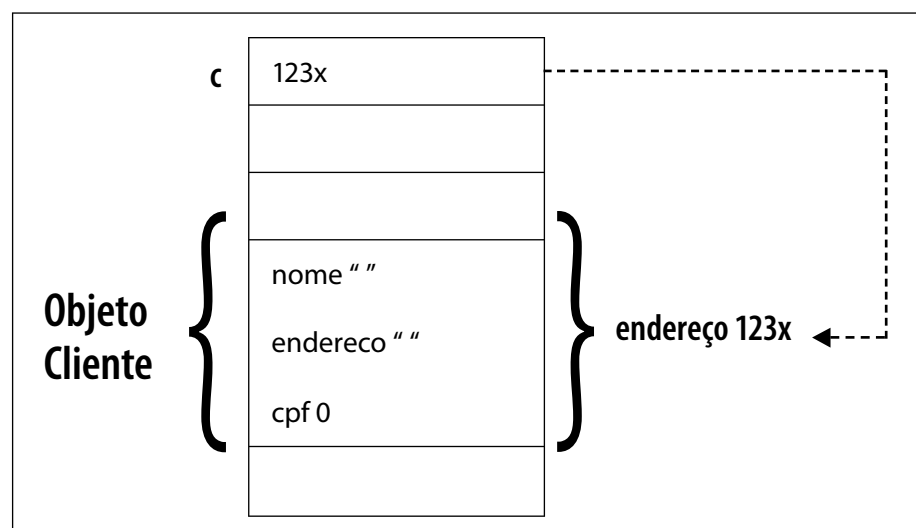
Será impresso na tela "Ana"



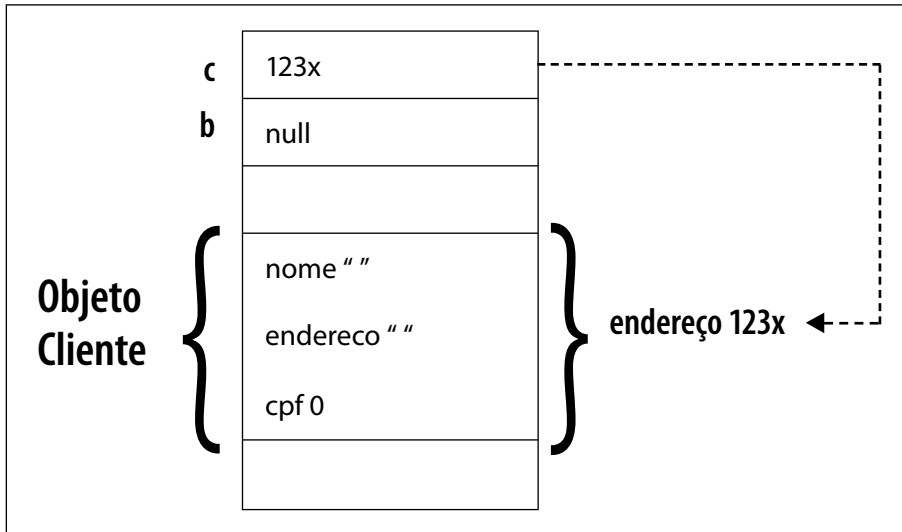
Por que será impresso o nome Ana na tela?

Vamos analisar linha a linha e fazer as ligações com a figura:

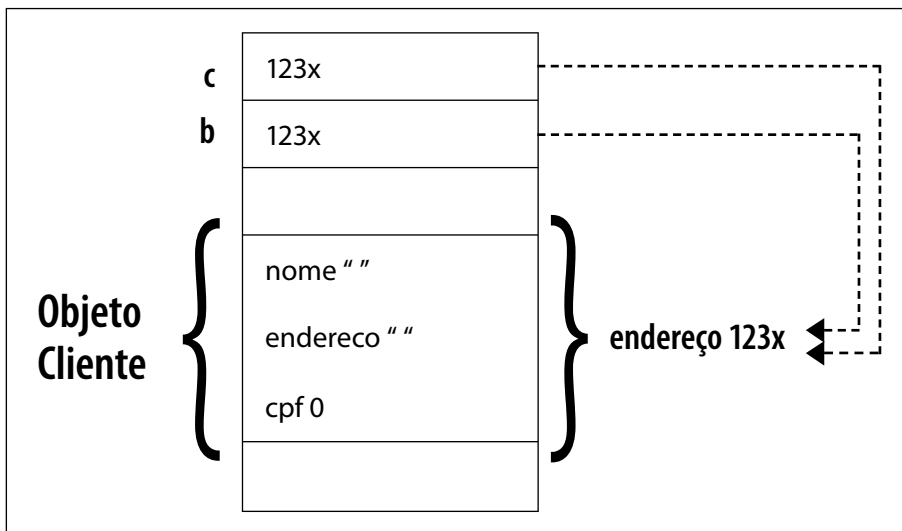
Linha 3: Uma variável referência chamada c é criada. À ela é armazenado o endereço do objeto do tipo Cliente criado com a palavra new. Note que na variável c é armazenado o endereço da área de memória alocada para o objeto Cliente, no caso 123x.



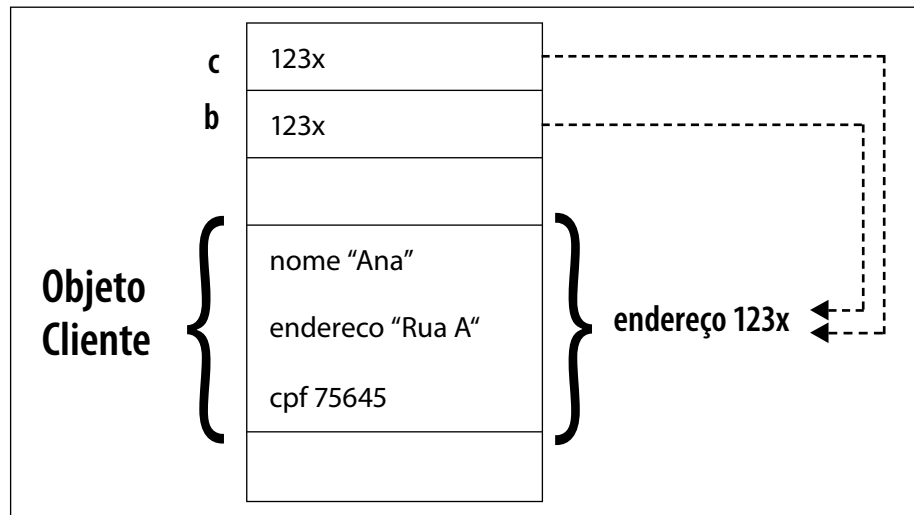
Linha 4: Outra variável referência chamada b, é criada. À ela não é armazenado nenhuma referência de objeto. Nesse caso, o valor dessa variável é null.



Linha 5: é atribuído (armazenado) na variável referência b o conteúdo da variável referência c. Portanto, o conteúdo de b agora é 123x e ela passa a referenciar o mesmo objeto Cliente. Agora c e b referenciam o mesmo objeto Cliente.



Linhas 6 a 8: os comportamentos ou métodos do objeto que a variável c referencia são invocados. C referencia o único objeto cliente criado, portanto, todos os métodos altera...() precedidos de c que são chamados nas linhas 6 a 8, irão manipular os atributos desse objeto.



Linha 9: nessa linha está sendo chamado o comportamento ou método `forneceNome()` precedido da nova referência `b`.

Quem `b` referencia?

O objeto `Cliente`.

Portanto, o método irá manipular atributos desse objeto. No caso, a implementação desse **método** faz retornar o conteúdo do atributo `nome`, assim sendo, é o conteúdo do atributo `nome` do objeto que será retornado e impresso na tela.

Veja a implementação desse e dos outros métodos na classe `Cliente` na unidade anterior.

SEÇÃO 2 - Criando programas com mais de um objeto

Até o momento criamos apenas uma classe (programa) que cria e manipula 1 objeto do tipo `Cliente` (`UsaCliente.java`).



Atenção!

Quando você encontrar alguma referência no texto para classe (programa), isso significa que não é uma classe que define os atributos e comportamentos de um objeto (essa também é conhecida como TAD – Tipo Abstrato de Dados) e sim, uma classe que cria e manipula (usa) objetos de um determinado tipo.

É preciso ficar claro que uma vez que a classe (TAD) que define os atributos e comportamentos de algum objeto está implementada e compilada, poderá ser usada para a criação e manipulação de objetos do tipo da classe em milhares de outras classes (programas).

Isso deve oportunizar você a entender mais visivelmente, um dos maiores benefícios da orientação a objetos, ou seja, uma vez que classe que define os atributos e comportamentos de algum objeto está pronta, ela pode ser usada para criar objetos e manipular seus atributos “por toda a vida”.

Você não terá mais que se preocupar em definir os atributos e comportamentos de algum objeto Cliente, por exemplo. Se isso foi feito uma vez, vai servir para ser usado para criar objetos do tipo Cliente para sempre, a não ser que os atributos de Cliente se alterem, diminuam ou aumentem. Nesse caso, a classe que representa esses atributos e comportamentos terá que ser alterada e compilada novamente.

Diante disso, vamos criar uma outra classe para manipular objetos do tipo Cliente.

Vamos chamar essa classe de UsaCliente2.java.

Logo abaixo está a implementação dessa classe.

O valor digitado pelo usuário nessa caixa de diálogo de entrada de dados será passado como parâmetro para o método alteraNome(). Isso quer dizer que o valor digitado pelo usuário será armazenado no atributo nome do objeto cliente1. O mesmo acontece para os demais métodos altera.().

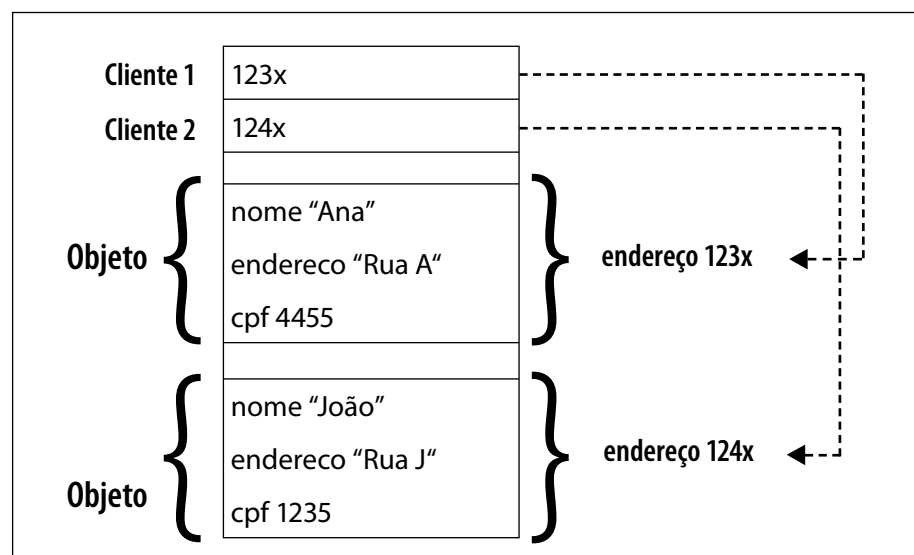
```

Linha 1  public class UsaCliente2{
Linha 2  public static void main(String args[]) {
Linha 3      Cliente cliente1=new Cliente();
Linha 4      Cliente cliente2=new Cliente();
Linha 5      cliente1.alteraNome(JOptionPane.showInputDialog("Entre com o nome"));
Linha 6      cliente1.alteraEndereco(JOptionPane.showInputDialog("Entre com o endereco"));
Linha 7      cliente1.alteraCpf(Integer.parseInt(JOptionPane.showInputDialog("Entre com o CPF")));
Linha 8      cliente2.alteraNome(JOptionPane.showInputDialog("Entre com o nome"));
Linha 9      cliente2.alteraEndereco(JOptionPane.showInputDialog("Entre com o endereco"));
Linha 10     cliente2.alteraCpf(1235);
Linha 11     System.out.println("Nome do Cliente 1"+ cliente1.forneceNome());
Linha 12     System.out.println("Nome do Cliente 2"+ cliente2.forneceNome());
Linha 13 }
Linha 14 }
```

Vamos analisar o código linha a linha:

- **Linha 3:** criação de uma variável chamada cliente1 que armazenará uma referência para o primeiro objeto Cliente. Atribuição a essa variável do endereço de memória do primeiro objeto Cliente.
- **Linha 4:** criação de uma variável chamada cliente2 que armazenará uma referência para o segundo objeto Cliente. Atribuição a essa variável do endereço de memória do segundo objeto Cliente.
- **Linhas 5 a 7:** alteração dos atributos do objeto cliente1 através da chamada de seus métodos altera...().
- **Linhas 8 a 10:** alteração dos atributos do objeto cliente2 através da chamada de seus métodos altera...().
- **Linha 11:** impressão na tela da frase “Nome do Cliente 1” seguido do nome que foi armazenado no atributo nome do objeto cliente1. O valor desse atributo é recuperado pelo método forneceNome() do objeto cliente1.
- **Linha 12:** idem à linha 11, só que para o objeto cliente2.

A representação gráfica desses dois objetos em memória é ilustrada na figura abaixo:



Ao término dessa seção, você já sabe que um objeto nada mais é do que uma área de memória que armazena valores para atributos definidos na classe (tipo) desse objeto, e que precisa ser referenciado por outra variável. Pode existir, também, mais de uma referência para um mesmo objeto, assim como podem existir (vamos programar assim) vários objetos alocados em memória.

SEÇÃO 3 - Refinando a criação de uma Classe

Nas duas seções anteriores você aprendeu a utilizar (usar) uma classe criando objetos a partir dela. Foi desenvolvido um programa que cria e manipula 1 objeto e, outro programa, que cria e manipula dois objetos do tipo Cliente.

Normalmente, os programas que você fará irão criar e manipular **muitos** objetos.

Antes disso, porém, você deve aprender mais alguns conceitos importantes relacionados ao desenvolvimento de uma classe.

Você já sabe o que é uma classe, pois, já implementou e utilizou uma para criar objetos. Contudo, existem mais conceitos relacionados à criação de classes. Agora, com um conhecimento um pouco mais maduro, você será apresentado a eles.

Para isso, você deve voltar à unidade anterior, mais especificamente onde a classe Cliente foi implementada.

O primeiro conceito importante que vamos rever é de **método construtor** de uma classe.

Método Construtor e Método Construtor Sobrecarregado

Quando um objeto é criado ou instanciado (alocado na memória através do operador new), um método definido na classe desse objeto sempre é automaticamente chamado, esse método é designado de construtor.



Esse método deve (não é obrigatório) ser definido na classe do objeto instanciado da seguinte forma:

```
public Cliente(){
    nome = "";
    endereço = "";
    cpf=0;
}
```

Cuidar letras maiúsculas e minúsculas. Se a classe de chamar cliente o método construtor também deve se chamar cliente().

Ele **SEMPRE** deve ter o mesmo nome da **classe** e não retornar valor (não deve nem ter a palavra void, que significa que nenhum valor está sendo retornado do método).

Quando o método construtor é chamado, o código que está dentro do método, é executado, portanto, devemos programar dentro de método construtor, qualquer instrução que queiramos que seja executada quando um objeto é criado. Normalmente, é programado dentro de um método construtor para que os atributos do objeto que está sendo criado sejam inicializados, ou seja, tenham valores consistentes (corretos).

É comum o desenvolvimento de mais de um método construtor com o mesmo nome, porém, com parâmetros diferentes, como mostra o trecho de código da classe Cliente a seguir:

```
public Cliente(){
    nome = "";
    endereço = "";
    cpf = 0;
}

public Cliente(String snome, String sendereco, int icpf){
    nome = snome;
    endereço = sendereco;
    cpf = icpf;
}
```

Quando isso acontece, se diz que o método construtor foi **sobrecarregado**.

Observe que o nome do método é igual, mas **um dos métodos não recebe parâmetro** e o **outro método recebe três parâmetros** que correspondem a valores que serão armazenados em cada atributo do objeto.

É importante observar que **os cabeçalhos dos métodos não devem ser iguais**. Deve haver uma diferença em nível de parâmetros, como no exemplo acima onde, um dos métodos não recebe parâmetros e o outro recebe.

O objetivo de **sobrecarregar** o método construtor é oferecer para quem for criar um objeto do tipo Cliente (um programador) outras maneiras de inicializar os atributos do objeto que foi instanciado.



1. Implemente o método sobrecarregado desenvolvido acima na classe Cliente.java
2. Compile a classe Cliente.java.
3. Edite e compile a classe abaixo: UsaCliente3.java

Criação do primeiro objeto e chamada do método construtor. Qual método construtor será chamado? O método construtor sem parâmetros implementado na classe Cliente. Na linha abaixo, o segundo objeto cliente está sendo criado e, nesse caso, o método construtor com parâmetros (o que implementado por último) está sendo chamado. Note que está sendo passado para o método três valores: Nome do cliente, endereço do cliente e o cpf.

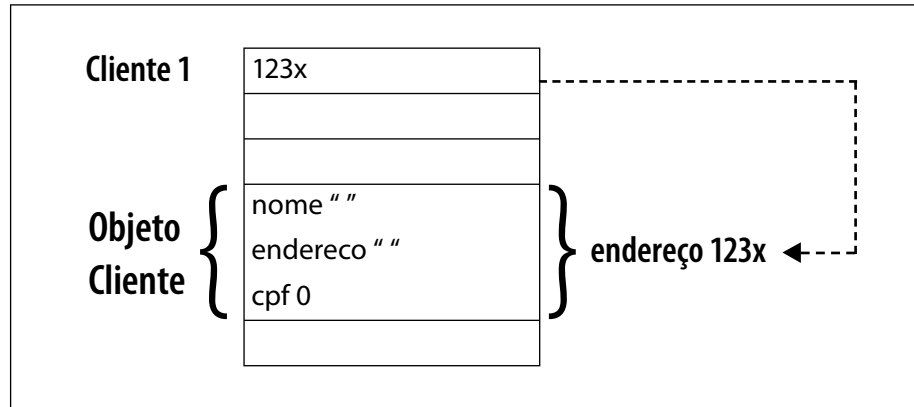
```

Linha 1  public class UsaCliente3{
Linha 2      public static void main(String args[]) {
Linha 3          Cliente cliente1=new Cliente( );
Linha 4          Cliente cliente2=new Cliente("Carlos", "Rua C", 123552244);
Linha 5          cliente1.alteraNome(JOptionPane.showInputDialog("Entre com o nome"));
Linha 6          cliente1.alteraEndereco(JOptionPane.showInputDialog("Entre com o endereco"));
Linha 7          cliente1.alteraCpf(12352241123);
Linha 11         System.out.println("Nome do Cliente 1"+ cliente1.forneceNome( ));
Linha 12         System.out.println("Nome do Cliente 2"+ cliente2.forneceNome( ));
Linha 13     }
Linha 14 }
```

Vamos analisar com mais detalhes:

Linha 3: o primeiro objeto cliente está sendo criado e o método construtor sem parâmetros Cliente () está sendo chamado. O que esse método faz? Inicializa cada atributo do objeto com string vazia para nome e endereço e 0 para cpf. Olhe a

implementação do método acima ou na classe `Cliente`. Nesse momento, o objeto representado em memória possui o seguinte estado:



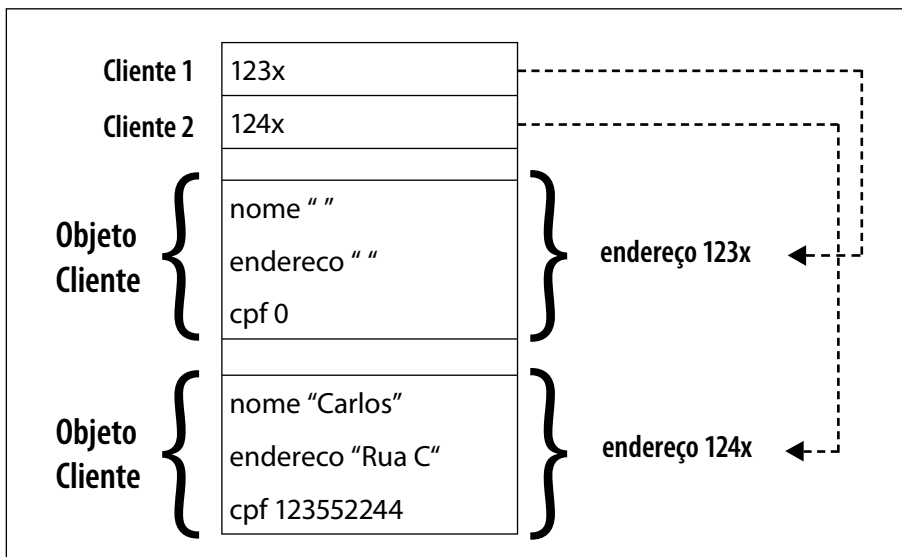
Observe que os atributos `nome`, `endereço` e `cpf` do objeto foram inicializados com os valores implementados no método construtor `Cliente()`.

Linha 4: o segundo objeto cliente está sendo criado e o método construtor com parâmetros `Cliente("Carlos", "Rua C", 123552244)` está sendo chamado. Cada valor (parâmetro) passado para o método é armazenado na respectiva variável de memória. Por exemplo: "Ana", será armazenado na variável de memória do método chamada `snome`; "Rua C", será armazenado na variável de memória do método chamada `sendereço`; o mesmo acontece com o valor `123552244`, que será armazenado na variável **cpf**.

Isso se chama passagem de parâmetros em subrotinas (aqui conhecida como método) e, você viu isso, em Lógica de Programação II.

Os valores passados serão armazenados, respectivamente, nessas variáveis de memória e, logo após, o conteúdo dessas variáveis será armazenado (atribuído) aos atributos do objeto que está sendo criado.

Portanto, esse método construtor sobrecarregado tem o objetivo de armazenar nos atributos do objeto, valores que são passados como parâmetro para o método. Nesse momento, o objeto representado em memória possui o seguinte estado:



Observe que, ao criar o segundo objeto e chamar o método construtor com parâmetros, os atributos nome, endereço e cpf, já possuem valores e, não precisam mais ser modificados através dos comportamentos ou métodos `alteraNome()`, `alteraEndereco()`, etc. como terá que acontecer com o objeto `cliente1`, nas linhas seguintes.



É uma boa prática de programação desenvolver pelo menos uma sobrecarga do método construtor de uma classe.

Importante!

Quando nenhum método construtor é especificado na classe, um construtor padrão é chamado automaticamente e inicializa atributos de tipo primitivos(int,double,etc.) com 0, atributos do tipo boolean para false e null para atributos do tipo **objeto**.

Outro conceito importante que vamos detalhar é o de **métodos modificadores (set) e recuperadores (get)** de uma classe.

Veremos o que é um atributo do tipo objeto mais adiante.

Métodos modificadores(set) e recuperadores(get)

Você implementou na classe Cliente uma série de métodos ou comportamentos para alterar o valor dos atributos de um objeto (métodos chamados altera...()) e para recuperar o valor desses atributos (métodos chamados fornece...()).

É importante você saber, e isso ficará mais claro na seção seguinte, que os atributos de um objeto só devem ser manipulados via comportamento ou métodos desse objeto e, nunca diretamente, como no exemplo:

```
cliente1.nome = "Ana";
```

Nesse caso, essa linha estaria na classe UsaCliente.

Essa instrução está tentando armazenar no atributo nome, do objeto cliente1, o nome "Ana". Está se tentando acessar o atributo diretamente e, não através de um método ou comportamento desse objeto. Isso não deve ser feito e, nem ao menos funcionará, pois, os atributos da classe Cliente estão com a palavra **private**.

Veremos com mais detalhes o significado e impacto dessa palavra nos atributos de uma classe mais tarde, por enquanto, é suficiente dizer que **essa palavra que impede que qualquer atributo de um objeto seja acessado diretamente em outra classe** (a linha acima estaria na classe UsaCliente), como no exemplo acima.

Portanto, para alterar o valor do atributo nome, **só nos resta chamar o método alteraNome("Ana") passando o nome "Ana" como parâmetro**. Isso só é possível porque todos os métodos implementados na classe Cliente têm a palavra **public** (isso também será explicado mais tarde).

Essa questão será revisada na próxima seção. O que nos interessa nesse momento está relacionado ao **nome dos métodos ou comportamentos** implementados na classe Cliente.

Na versão implementada da classe Cliente todos os métodos que alteram o valor dos atributos são chamados de altera...(). Esse nome foi utilizado para facilitar o entendimento da necessidade de existência desses métodos, o mesmo valendo para os métodos fornece...() que recuperam o valor dos atributos de um objeto.

O objetivo desse tópico é informar que existe um padrão de nomenclatura para nomes de métodos, classes e atributos.



Padrão de nomenclatura quer dizer que as pessoas que implementam classes usam o mesmo padrão de nomes para identificar (nomear) classes, atributos e métodos de uma classe.

Em relação aos métodos de uma classe, sempre que um método tem a função única e exclusiva de alterar (modificar) o atributo de uma classe ele deve começar com o a palavra **set** seguido do nome do atributo que ele altera.



Método na primeira versão da classe Cliente:

```
public void alteraNome(String snome){
    nome=snome;
}
```

Nome de método recomendado:

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

Note que o que muda é o início do nome do método. Todos os demais nomes de métodos devem ser alterados para começar com a palavra **set**.

O mesmo acontece com os métodos que tem a função de fornecer ou recuperar o valor dos atributos de um objeto, os métodos fornece...(). Esses métodos devem começar com a palavra **get** seguido do nome do atributo que ele recupera.



Método na primeira versão da classe Cliente:

```
public String forneceNome(){
    return nome;
}
```

Nome de método recomendado:

```
public String getNome(){
    nome=snome;
}
```

Note que o que muda é o início do nome do método. Todos os demais nomes de métodos devem ser alterados para começar com a palavra **get**.



Portanto, todo o método que se destina, exclusivamente, a alterar o valor de uma atributo deve começar com **set** e todo o método que se destina a recuperar o valor de um atributo com **get**.

Métodos **set** são denominados métodos **modificadores** porque modificam o valor dos atributos.

Métodos **get** são denominados métodos **recuperadores** porque recuperam o valor dos atributos.

Os métodos da classe `Cliente` devem ser todos alterados. A nova implementação da classe `Cliente` ficará como a seguir:

```

Linha 1  public class Cliente{
Linha 2      private String nome, endereco;
Linha 3      private int cpf;
Linha 4
Linha 5      public Cliente( ){
Linha 6          nome="";
Linha 7          endereco="";
Linha 8          cpf=0;
Linha 9      }
Linha 10     public void setNome(String snome){
Linha 11         nome=snome;
Linha 12     }
Linha 13
Linha 14     public void setEndereco(String sender){
Linha 15         endereco=sender;
Linha 16     }
Linha 17
Linha 18     public void setCpf(int icpf){
Linha 19         cpd=icpf;
Linha 20     }
Linha 21     public String getNome(){
Linha 22         return nome;
Linha 23     }
Linha 24
Linha 25     public String getEndereco(){
Linha 26         return endereco;
Linha 27     }
Linha 28
Linha 29     public int getCpf(){
Linha 30         return cpf;
Linha 31     }
Linha 32 }
```

SEÇÃO 4 - Encapsulamento

Na unidade anterior você estudou o conceito de ABSTRAÇÃO, que é empregado no momento em que você abstrai atributos e comportamentos de um objeto do mundo real para um objeto do mundo do software e representa isso através de uma classe.

Outro conceito muito importante da OO é o **ENCAPSULAMENTO**.

O encapsulamento nos diz que atributos e comportamentos em cima desses atributos devem ser encapsulados dentro do objeto, ou seja, os atributos de um objeto (nome, cpf, saldo...) devem ser escondidos de nós (programadores) e só devemos ter acesso a esses atributos através dos métodos ou comportamentos desse objeto.

Isso quer dizer que, não devemos ter acesso livre aos atributos de um objeto, como mostra o trecho de código abaixo:

```
.  
.   
Cliente c = new Cliente().  
c.nome="Ana";  
.   
.
```

No trecho acima, estamos tentando acessar (alterando o valor) o atributo nome do objeto c “diretamente”. Isso não deve acontecer, não é recomendado e burla o princípio do encapsulamento que nos diz que os atributos de um objeto devem estar encapsulados (escondidos dentro do próprio objeto) e só devem ser acessados por comportamentos ou métodos do próprio objeto.

O encapsulamento permite o **ocultamento das informações** de quem vai manipular o objeto, ou seja, não é preciso conhecer detalhes sobre os atributos de um objeto, só é preciso ter meios (métodos ou comportamentos) para manipular essas informações.

É por isso que no código que está logo no início dessa unidade, alteramos os valores dos atributos do objeto c através da chamada de seus métodos ou comportamentos **públicos**.

Pode-se dizer que um objeto é manipulado através da sua **interface**. A interface de um objeto é o conjunto de métodos públicos disponibilizados para manipular um objeto.



Estamos falando de métodos públicos. Mas o que são métodos públicos?

Observe a implementação da classe Cliente. Todo o método é definido com a palavra **public** e todo o atributo com a palavra **private**.

Para tornar um método público é preciso usar a palavra **public** na definição do método.

Isso significa que ele pode ser chamado (precedido da referência ao seu objeto) de qualquer outra classe. Dessa maneira, está se disponibilizando a interface do objeto para a sua manipulação.

Para tornar um atributo privado, é preciso usar a palavra **private** na definição do atributo. Ao tornar um atributo privado, estamos promovendo o encapsulamento, porque o atributo só poderá ser acessado por métodos do objeto.

Um dos objetivos do encapsulamento é a facilidade de manutenção do código.

Isso quer dizer que, se houve alterações na estrutura de um objeto, como por exemplo, o atributo cpf, não armazena mais dados do tipo int e sim, dados do tipo String.

Isso NÃO IRÁ ser refletido nas várias classes que criam e manipulam objetos do tipo Cliente, ou seja, não será necessário alterar (dar manutenção) nessas várias classes que manipulam objetos do tipo Cliente, porque essa mudança só ocorrerá dentro da classe Cliente, mais especificamente, na linha que define o atributo cpf.

As demais classes que criam e manipulam objetos do tipo Cliente continuarão chamando os métodos `alteraCpf()` e `forneceCpf()` para ter acesso ao atributo cpf que agora é do tipo String.

Se o encapsulamento não for mantido nesse exemplo, o atributo `cpf` nessas classes poderia ser acessado diretamente. Por exemplo:

```
Cliente c=new Cliente();
c.cpf=123525222;
```

Valor inteiro sendo armazenado diretamente no atributo `cpf` do objeto `c`.

Com a mudança no tipo do atributo, **TODAS** as classes que criam e manipulam objetos do tipo `Cliente` teriam que ser alteradas para o seguinte código:

```
Cliente c=new Cliente();
c.cpf="123525222";
```

O atributo `cpf` agora só armazena valor do tipo `String`. Como ele está sendo acessado diretamente nessa e em outras classes, todas elas precisarão ser alteradas para que `cpf` receba um dado do tipo `String`. Note que o valor está entre "" (aspas).



Síntese

Nessa unidade você aprendeu a diferença entre dois termos usados na criação de um objeto. O termo referência e o termo objeto. Referência é o endereço físico de um objeto em memória. Essa referência aponta para uma determinada área de memória que é chamada de objeto.

Aprendeu a criar programas que envolviam a criação de mais de um objeto e também o conceito muito importante da OO que é o Encapsulamento.



Atividades de auto-avaliação

- 1) Represente graficamente em memória, tal como foi feito nessa unidade, a seguinte instrução. Identifique na representação gráfica o que é referência e o que é objeto.

```
Cliente cliente = new Cliente ("Rita", "Rua R", 215215211);
```

- 2) Quando um método construtor é chamado?

- 3) Qual a função de um método modificador numa classe? E de um método recuperador?

Aplicando os conceitos de OO através de exemplos práticos



Objetivos de aprendizagem

- Fortalecer a aprendizagem dos conceitos estudados anteriormente através de exemplos.
- Desenvolver programas com vetor de objetos.
- Utilizar como recurso o Método sobrecarregado (overload).



Seções de estudo

Seção 1 Colocando em Prática os conceitos de OO

Seção 2 Método sobrecarregado



Para início de conversa

Na unidade anterior, você foi apresentado a mais alguns conceitos de OO, dentre eles, o de **encapsulamento**. Você estudou a diferença entre referência e objeto, aprendeu a desenvolver programas que exigiam a criação de mais de um objeto e a implementar uma classe com método construtor, modificador (set) e recuperador (get).

Nessa unidade, você irá reforçar a maioria dos conceitos apresentados anteriormente, através de uma série de exemplos concretos, diferentes do Sistema de Conta Corrente, descrito nas unidades anteriores. Aprenderá o conceito de método sobrecarregado e mais alguns conceitos novos, mas tudo através de exemplos práticos.

A partir da descrição de um problema iremos primeiramente implementar a classe que representa os atributos e comportamentos do(s) objeto(s) existente no problema e, finalmente, criar, a partir dessa classe, o(s) objetos(s) para solucionar o problema.

Tudo isso será feito colocando a “mão na massa”, ou seja, codificando (programando) na linguagem Java. Vamos começar?

Então, bom estudo!

SEÇÃO 1 - Colocando em prática os conceitos de OO

Problema 1

Uma empresa necessita de um programa para armazenar as informações sobre seus 50 funcionários. As informações que deseja armazenar são: nome, endereço e salário. Desenvolva um programa dentro do paradigma orientado a objetos.

Diante do enunciado desse problema, como você pretende resolvê-lo? Ou melhor, implementá-lo?

Pense em uma resposta para as seguintes questões. (Não olhe para as respostas sugeridas, pense nas suas e depois confira!)

1. Qual o(s) **objeto(s)** desse problema do mundo real?
2. Quais os atributos (características) e comportamentos desse(s) objeto(s)?
3. Todos os objetos são do mesmo tipo, ou seja, possuem o mesmo conjunto de atributos e comportamentos?
4. Esse problema trata com 1 ou vários objetos?

Armazenar informações sobre os funcionários de uma empresa é um problema do mundo real.

Respostas:

- 1) Funcionário. Esse problema trata com objetos do tipo Funcionário.
- 2) Todo funcionário tem como atributo nome, endereço, salário, estado civil, sexo, cor dos olhos, etc. Todo funcionário tem comportamentos do tipo digitar documentos, caminhar, fornecer seu nome, fornecer seu endereço, etc.
- 3) Sim, todos os objetos são do mesmo tipo: Funcionário, logo possuem o mesmo conjunto de atributos e comportamento.
- 4) Esse problema trata com vários objetos, mais precisamente, 50 objetos.

De acordo com as respostas acima, chegamos à conclusão que existe somente um tipo de objeto (Funcionário) e que o problema trata com vários objetos desse mesmo tipo (50). Esses objetos possuem um conjunto de atributos e comportamento no mundo real (exemplos foram listados acima).



Mas será que nos interessa todos os atributos e comportamentos desses objetos no contexto de um sistema (software) orientado a objetos? Muito provavelmente não!

No nosso sistema de cadastro de funcionários, não interessa atributos do tipo cor do cabelo e comportamento do tipo digitar documento. Portanto, vamos aplicar o processo de **abstração**, ou seja, desse conjunto de atributos e comportamentos de funcionários no mundo real, vamos apanhar somente os atributos e comportamentos que interessam ao sistema (software) de cadastro de funcionários.

Os atributos que escolhemos são: nome, endereço e salário



“Abstrair é extrair tudo o que for essencial e nada mais”.

Aaron Walsh

“A abstração é o processo de filtragem de detalhes sem importância do objeto, para que apenas as características apropriadas que o descrevem permaneçam”.

Peter Van Der Linden

Para cada **atributo** definido, deve-se ter um *comportamento* ou *método* para armazenar algum valor no atributo (método set) e para recuperar o valor do atributo (método get).

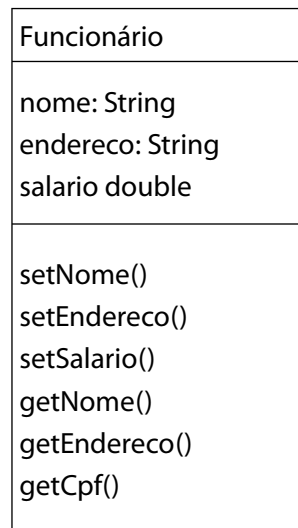
Feito a abstração, primeiramente devemos criar uma classe que represente os atributos e comportamentos dos vários objetos Funcionário.

Pode existir mais de uma classe se existir mais de um tipo de objeto no problema e elas ainda podem se relacionar. Veremos isso nas unidades seguintes!

Essa classe deve se chamar Funcionário. Ela deve ter como atributos: nome, endereço e salário e deve ser um conjunto de comportamentos (métodos) que manipulem esses atributos. Você Lembra do **encapsulamento**?

Identificada a(s) **classes(s)** que representam os objetos do problema, devemos criar um modelo gráfico dessa(s) classe(s). Esse modelo gráfico é chamado de Diagrama de Classes (já falamos dele antes, lembram?).

Como nesse problema só identificamos a classe Funcionário, nosso diagrama será composto de apenas 1 classe.



Observe que o símbolo gráfico que representa a classe possui três divisões:

- a primeira corresponde ao nome da classe;
- a segunda aos atributos da classe; e
- a terceira corresponde ao comportamento da classe.

É importante que você saiba que quando falamos em atributos e comportamentos da classe, na realidade, estamos querendo dizer, atributos e comportamento dos objetos que a classe representa.



Em relação aos comportamentos, você lembra por que chamamos os comportamentos ou métodos de **setNome()** e **getNome()**?

Você estudou na unidade oito, na seção sobre Encapsulamento, que os atributos de um objeto devem estar protegidos, encapsulados dentro do objeto (dentro de uma área de memória) e que só podem ser acessados por comportamentos ou métodos do próprio objeto.

Portanto, devemos ter um comportamento ou método público (public) para modificar (set) e recuperar (get) cada atributo de um objeto Funcionário. Por isso, temos que planejar e implementar

um método `setNome()` para modificar o valor do atributo `nome` de um objeto `Funcionario` na memória e, devemos planejar e implementar um método `getNome()` para recuperar o valor desse **atributo**.

Lembre-se que qualquer método que modifica o valor do atributo de um objeto deve começar pela palavra `set` e qualquer método que recuperar o valor do atributo deve começar pela palavra `get`. Isso é um padrão de nomenclatura.

private: para promover o princípio do encapsulamento devemos colocar os atributos de classe como privados (`private`)

É recomendado que toda classe tenha pelo menos 1 método construtor. Método construtor sempre tem o mesmo nome da classe e é executado quando um objeto do tipo `Funcionario` é criado.

Método `setEndereco(String sender)` recebe como parâmetro o endereço do funcionario na variável `sender` e armazena o conteúdo dessa variável no atributo `endereco` do objeto `Funcionario`.

Método `getEndereco()` retorna o conteúdo do atributo `endereco` do objeto `Funcionario`.

Depois de identificar a classe que representa o objeto do problema e de representar graficamente essa classe, estamos prontos para a implementação (programação).

```

Linha 1  public class Funcionario{
Linha 2      private String nome, endereco;
Linha 3      private double salario;
Linha 4
Linha 5      public Funcionario() {
Linha 6          nome="";
Linha 7          endereco="";
Linha 8          salario=0;
Linha 9      }
Linha 10     public void setNome(String snome){
Linha 11         nome=snome;
Linha 12     }
Linha 13
Linha 14     public void setEndereco(String sender){
Linha 15         endereco=sender;
Linha 16     }
Linha 17
Linha 18     public void setSalario(double dsalario){
Linha 19         salario=dsalario;
Linha 20     }
Linha 21     public String getNome(){
Linha 22         return nome;
Linha 23     }
Linha 24
Linha 25     public String getEndereco(){
Linha 26         return endereco;
Linha 27     }
Linha 28
Linha 29     public double getSalario(){
Linha 30         return salario;
Linha 31     }
Linha 32 }
```

Procedimentos Práticos:

1. Digitar o código acima e salvar o arquivo com o nome `Funcionario.java` na pasta `CURSOWEB`. (Lembre-se que você pode usar o Bloco de Notas ou uma IDE como JCreator, JBuilder, etc)
2. Compilar a classe `Funcionario.java`. Se você usou o Bloco de Notas para digitar a classe, entre na janela do MS-DOS, vá para a pasta `CURSOWEB` e digite o comando de compilação:

```
javac Funcionario.java
```

Se a sua classe não contiver erros, será gerado o arquivo `Funcionario.class`. É desse arquivo que você precisa para criar objetos do tipo `Funcionario` no próximo programa e em qualquer programa que exija a criação de objetos do tipo `Funcionario`.



Essa é uma grande vantagem da orientação a objetos. Uma vez que você cria um tipo abstrato de dados (classe), pode usar sempre que tiver necessidade, sem precisar escrever a classe novamente. Isso se chama **reutilização de código**.

Se a classe contiver erros, você deve voltar ao Bloco de Notas, arrumar o código da classe e compilar novamente. Esse processo só estará terminado quando o arquivo `Funcionario.class` for gerado com sucesso.

Até o momento, podemos dizer que 50% do nosso trabalho está realizado. Temos, no entanto, um problema. Identificamos o objeto desse problema, desenhamos e implementamos uma classe para representar esse objeto ou o conjunto desses objetos. O que precisamos fazer é usar essa classe para resolver o problema.

Releia o problema novamente.

Devemos criar um programa OO para armazenar informações de 50 funcionários de uma empresa. Nesse programa OO, para armazenar a informação de 1 funcionário, devemos criar 1 objeto do tipo Funcionário, como mostra o código abaixo:

```

Linha 0  import javax.swing.*;
Linha 1  public class CadastroFuncionario{
Linha 2      public static void main(String args[]) {
Linha 3          Funcionario f=new Funcionario( );
Linha 4          f.setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 5          f.setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 6          f.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 7          JOptionPane.showMessageDialog(null,"Nome do Funcionário Cadastrado " + f.getNome());
Linha 8          System.exit(0);
Linha 9      }
Linha 10 }
```

Classe CadastroFuncionario que cria um objeto do tipo Funcionário (linha 3) para poder armazenar nome, endereço e salário de um funcionário.

O programa acima, depois de compilado e executado irá armazenar as informações de 1 funcionário. Ao executar a linha 3 será criado um objeto do tipo Funcionário, portanto, será alocada somente uma área de memória onde poderá ser armazenado nome, endereço e salário de um funcionário. Logo após, nas linhas 4 a 5 os métodos `setNome()`, `setEndereco()` e `setSalario()` serão chamados. Cada método chamado passa como parâmetro um valor digitado pelo usuário. Esse valor é armazenado no respectivo atributo do objeto `f`. No caso da linha 4, o método `setNome()` passa como parâmetro o nome que o usuário irá digitar na caixa de entrada de dados `JOptionPane.showInputDialog("Entre com o Nome")`.



Essa é a maneira que programamos qualquer entrada de dados do usuário. Ela faz aparecer uma caixa no centro da tela, com um campo onde o usuário digita algum valor.

A linha 7 imprime na tela a frase “Nome do Funcionário Cadastrado “ seguido do nome digitado do funcionario. Para recuperar o nome digitado e armazenado no atributo nome do objeto `f`, é invocado na linha 4 o método `getNome()`.

Procedimentos Práticos:

1. Digitar o código acima e salvar o arquivo com o nome CadastroFuncionario.Java na pasta CURSOWEB.
2. Compilar a classe CadastroFuncionario.java. A compilação deve gerar o arquivo CadastroFuncionario.class. Comando de compilação:

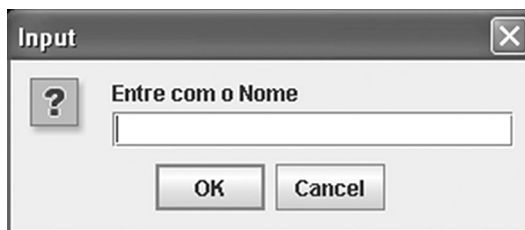
```
javac CadastroFuncionario.java
```

3. Executar o arquivo CadastroFuncionario.class. Comando para execução:

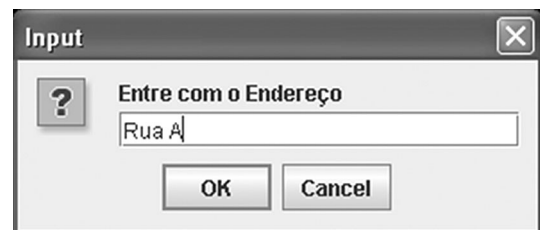
```
java CadastroFuncionario
```

Obs.: Nunca usar .class no comando de execução.

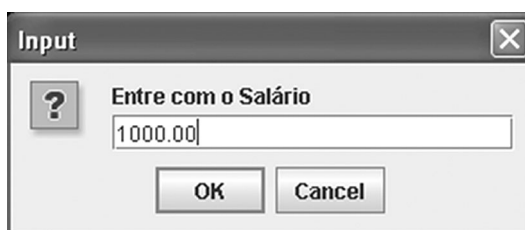
A seguir, é apresentada a sequência de telas do programa exemplificado mais acima:



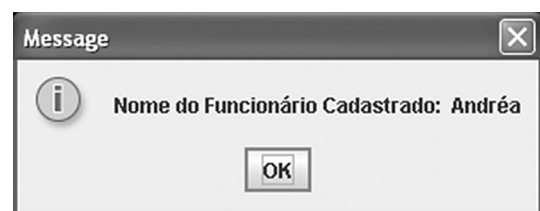
Tela 1



Tela 2



Tela 3



Tela 4

Como você deve ter observado, o programa acima armazena os dados de 1 funcionário. Porém, o problema inicial pede um programa para armazenar os dados de 50 funcionários. O que devemos alterar no nosso programa, ou seja, na classe `CadastroFuncionario.java` para permitir que sejam armazenados os dados de 50 funcionários?

Devemos criar 50 objetos! Isso mesmo, 50 objetos.

Vamos começar a desenvolver o código para criar 50 objetos.

```
Linha 0  import javax.swing.*;  
Linha 1  public class CadastroFuncionario{  
Linha 2      public static void main(String args[]) {  
Linha 3          Funcionario f1=new Funcionario( );  
Linha 4          Funcionario f2=new Funcionario( );  
Linha 5          Funcionario f3=new Funcionario( );  
Linha 6          Funcionario f4=new Funcionario( );  
Linha 7          Funcionario f5=new Funcionario( );  
Linha 8              ...  
Linha 9              ....  
Linha 10             ....  
Linha 11      }  
Linha 12 }
```

Quantas linhas de código teremos que implementar para criar os 50 objetos? 50 linhas!

E para armazenar as informações nos 50 objetos, 50 linhas para cada atributo que iríamos querer armazenar algum valor. Multiplique isso por três atributos e, então teremos 150 linhas.

É muita linha de código com a mesma instrução. O que muda é só o nome do objeto. Para evitar esse excesso de linhas de código, você vai aprender a trabalhar com **VETOR DE OBJETOS**.

Você já aprendeu a trabalhar com vetor em Lógica de Programação II e a implementar um vetor na linguagem Java nas unidades anteriores.



Um **vetor** é uma variável de memória que pode armazenar vários valores, todos do mesmo tipo de dado sendo que, cada posição dessa variável, armazena somente um valor.

Vamos revisar como trabalhar com vetor?

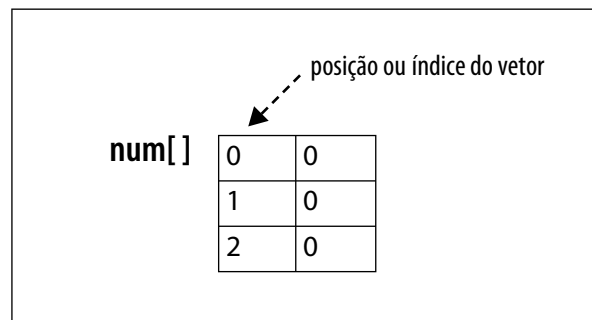
```

Linha 0  import javax.swing.*;
Linha 1  public class CriarVetor{
Linha 2      public static void main(String args[]) {
Linha 3          int num[]=new int[3];
Linha 4          num[0] = 5;
Linha 5          num[1] = Integer.parseInt(JOptionPane.showInputDialog("Entre com o Número"));
Linha 6          num[2] = 6;
Linha 7          System.exit(0);
Linha 8      }
Linha 9  }

```

Vamos analisar o código linha a linha:

- **Linha 3:** Criação de um vetor chamado num de 3 posições. Esse vetor é do tipo int, ou seja, só pode armazenar valores do tipo int (inteiro). Lembre-se que, cada valor dentro do vetor é acessado através de uma posição ou índice. Quando um vetor do tipo int é criado, cada posição do vetor é inicializada com 0.



- **Linha 4:** Na primeira posição do vetor, posição 0, é armazenado o valor 5.

- **Linha 5:** Na segunda posição do vetor, posição 1, é armazenado o valor digitado pelo usuário.
- **Linha 6:** Na terceira posição do vetor, posição 2, é armazenado o valor 6.

Outra forma de armazenar dados num vetor é usar uma **estrutura de repetição** (para-faça ou for na linguagem Java) e programar para o usuário digitar os dados que serão armazenados no vetor.

Vejamos na próxima versão do programa.

```
Linha 0  import javax.swing.*;  
Linha 1  public class CriarVetor{  
Linha 2      public static void main(String args[]) {  
Linha 3          int num[]=new int[3];  
Linha 4          for (int i=0;i<3;i++)  
Linha 5              num[i] = Integer.parseInt(JOptionPane.showInputDialog("Entre com o Número"));  
Linha 6  
Linha 7          System.exit(0);  
Linha 8      }  
Linha 9 }
```

Nessa versão, em cada posição do vetor num (de 0 até 2) é armazenado um valor digitado pelo usuário. Note que não foi preciso programar 3 instruções de entrada de dados para armazenar o valor em cada posição do vetor num. Usamos uma estrutura de repetição 'for' e mandamos ela repetir 3 vezes, ou seja, executar a instrução de atribuição programada dentro dela 3 vezes.

Na última versão do programa (classe) CadastroFuncionario.java, vamos usar um vetor para armazenar os 50 objetos que precisamos. Só que esse vetor armazenará objetos do tipo Funcionario. Vamos usar também uma estrutura de repetição **for** para trabalhar melhor sobre esse vetor de objetos.

```

Linha 0  import javax.swing.*;
Linha 1  public class CadastroFuncionario{
Linha 2      public static void main(String args[]) {
Linha 3          Funcionario f[ ]=new Funcionario[50];
Linha 4          for (int i=0;i<50;i++){
Linha 5              f[i]=new Funcionario( );
Linha 6              f[i].setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 7              f[i].setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 8              f[i].setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 9              JOptionPane.showMessageDialog(null,"Nome do Funcionário Cadastrado " + f[i].getNome( ));
Linha 10     }
Linha 11     System.exit(0);
Linha 12     }
Linha 13 }

```

Vamos analisar o código linha a linha:

Linha 3: Criação de um vetor de 50 posições chamado f. Esse vetor é do tipo Funcionario, ou seja, somente pode armazenar objetos do tipo Funcionário. Quando um vetor de um tipo abstrato de dados (no caso classe Funcionário) é criado, cada posição do vetor é inicializada com o valor null.

f[]	0	null
	1	null
	2	null
	3	null
	4	null
	..	null
	..	null
	49	null

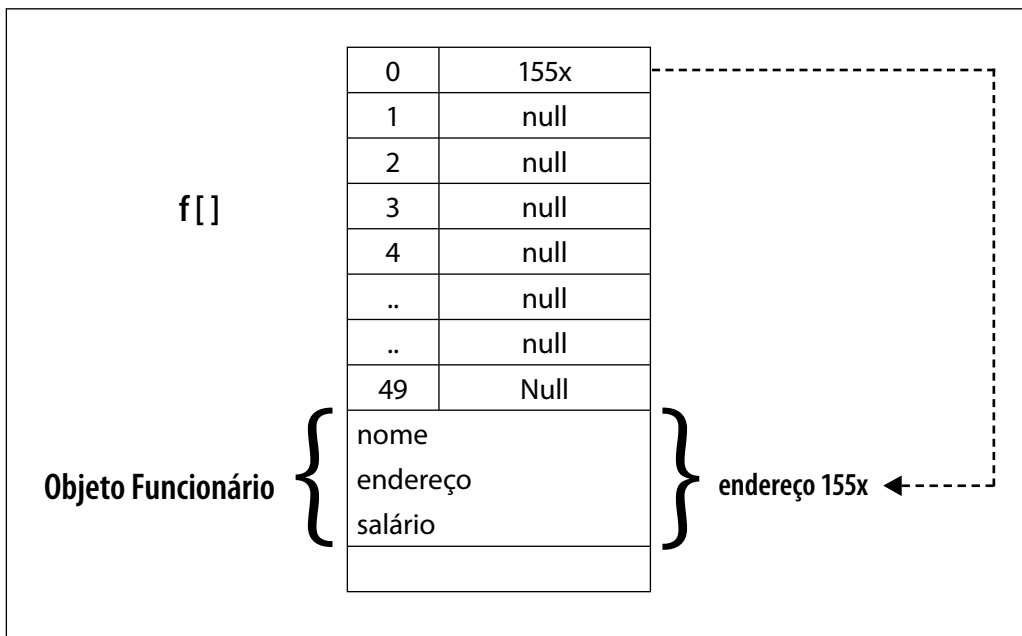
Linha 4: Observe que as posições do vetor não referenciam nenhum objeto do tipo Funcionário, elas contêm o valor null. Precisamos então criar, instanciar um objeto do tipo Funcionário em cada posição do vetor f. Poderíamos fazer isso sequencialmente e repetidamente da seguinte forma:

```
..  
..  
f[0]=new Funcionario( );  
f[1]= new Funcionario( );  
f[1]= new Funcionario( );  
f[1]= new Funcionario( );  
..  
f[49]= new Funcionario( );  
..
```

Isso está correto, mas vamos ter que programar 50 linhas de código para armazenar a referência de um objeto em cada posição do vetor. Ao invés de fazer assim, vamos usar uma estrutura de repetição **for** que repetirá 50 vezes as instruções das linhas 5 a 9.

Linha 5: Nessa linha está sendo criado um objeto do tipo Funcionário e sua referência está sendo armazenada no vetor f[] em alguma das suas posições. Por exemplo, quando a estrutura de repetição for iniciar a variável de controle i, vai estar com o valor inicial 0, logo, na linha 5, será armazenado na posição 0 do vetor f[] uma referência a um objeto do tipo Funcionário, em palavras mais simples, será criado um objeto do tipo Funcionário.

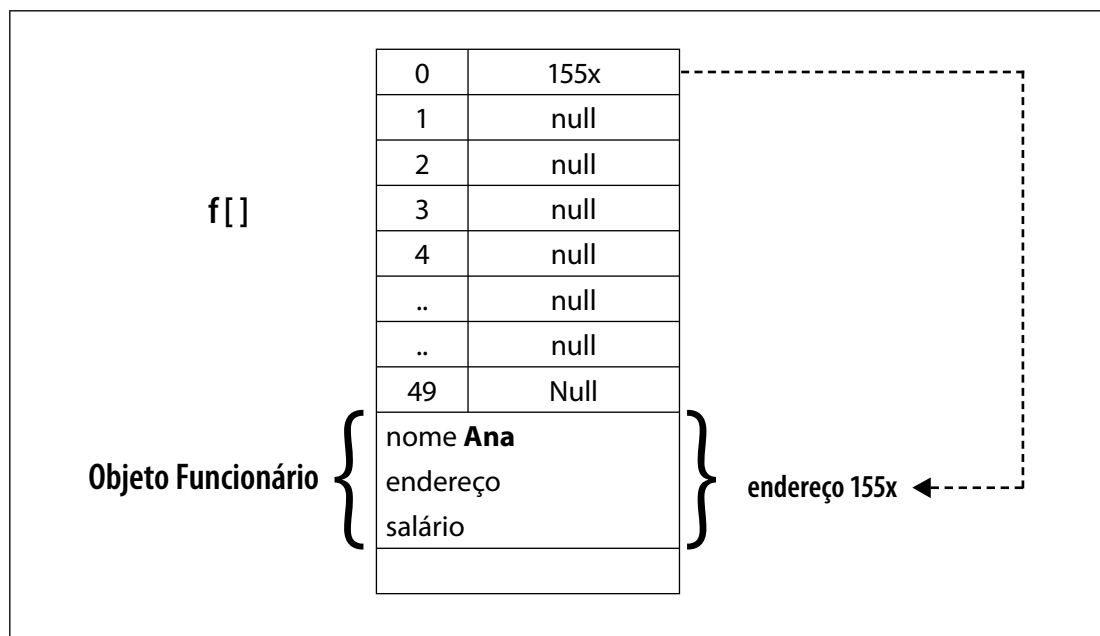
A figura a seguir ajuda a entender essa explicação.



Sempre que a posição `f[0]` for acessada, estará sendo acessado o objeto Funcionário do endereço 155x de memória.

O mesmo irá acontecer para as demais posições do vetor `f[]`, pois essa instrução da linha 5, está dentro de uma estrutura de repetição que irá executar essa instrução 50 vezes, ou seja, serão criados 49 objetos do tipo Funcionário e a referência de cada um será armazenada em cada posição do vetor `f[]`.

Linha 6: Depois de criar o objeto em determinada posição do vetor, por exemplo na posição 0, é chamado o método `setNome()` desse objeto (note que a maneira de fazer referência ao objeto é através do nome do vetor e a posição – `f[i]` – onde `i` tem valor 0, no momento) e passado como parâmetro o nome do funcionário digitado pelo usuário. Isso fará com que o nome digitado seja armazenado no atributo `nome` do objeto que a posição 0 do vetor `f[]` faz referência. A figura a seguir ilustra essa instrução:

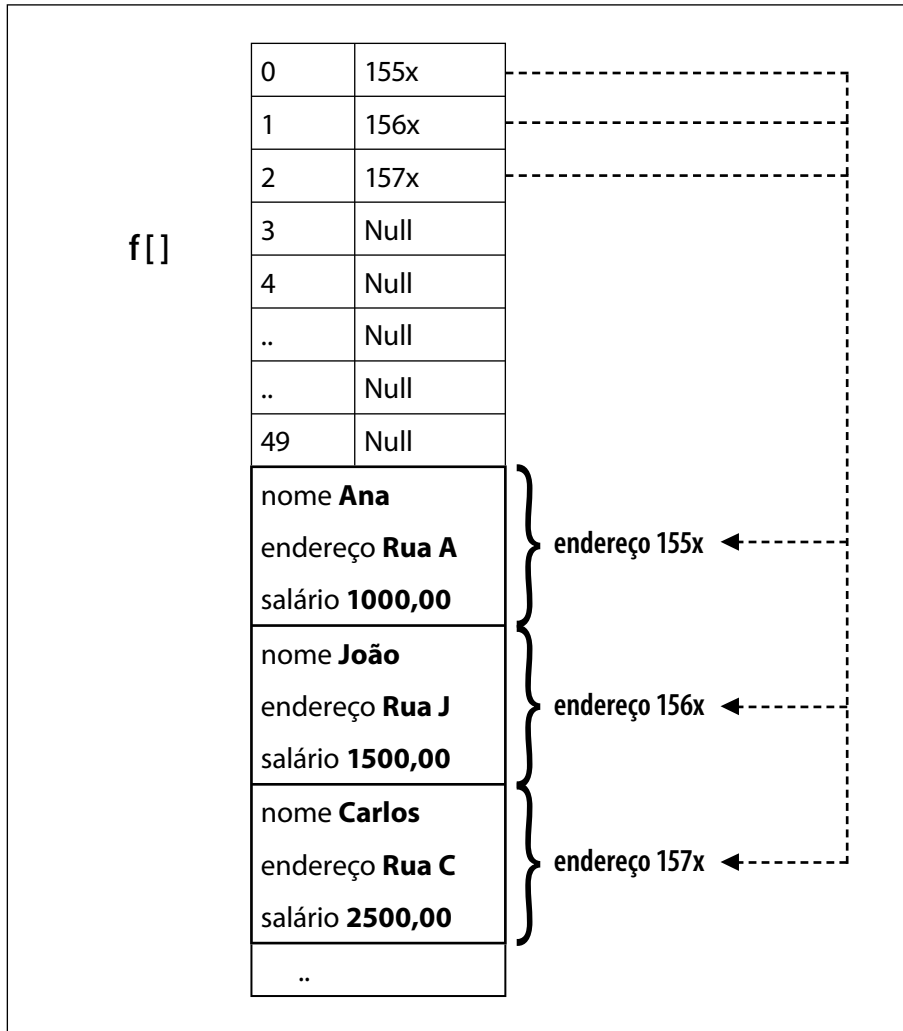


Linhas 7 a 8: Idem para as linhas 7 e 8 que chamam respectivamente os métodos `setEndereco()` e `setSalario()` e passam como parâmetro, o endereço e o salário do funcionário digitados pelo usuário.

Linha 9: Depois de criar o objeto e armazenar a referência em determinada posição do vetor, armazenar os valores de nome, endereço e salário no objeto `Funcionário`, essa linha imprime na tela o nome do funcionário que acabou de ser armazenado no atributo `nome` desse objeto. Para isso, chama o método `getNome()` desse objeto. Note que sempre precedido da referência ao objeto que está armazenada no vetor `f[i]`, onde `i` possui um valor de 0 a 49 no momento.

Linha 10: Final da estrutura de repetição **for**. Observe que as cinco instruções anteriores serão executadas 50 vezes, pois o intervalo de repetição estabelecido foi 50.

A figura a seguir ilustra alguns objetos Funcionários criados, suas referências armazenadas no vetor `f[]` e com valores dentro dos atributos.



SEÇÃO 2 - Método sobrecarregado

Quando você implementa os comportamentos (métodos) de uma classe, pode utilizar o recurso de **método sobrecarregado**. Um método sobrecarregado, como já vimos um pouco na unidade anterior, **é um método com o mesmo nome de outro método que já existe na classe**. Mas ele deve ter apenas o nome igual ao nome de outro método que já existe. Isso quer dizer que, alguma diferença em nível de parâmetros que o método recebe ou do tipo de retorno do método, deve existir.

Observe o trecho de código da classe Funcionario:

```
public void setSalario(double dsalario){  
    salario=dsalario;  
}  
  
public void setSalario(String ssalario){  
    salario = Double.parseDouble(ssalario);  
}
```

Existem dois métodos com o mesmo nome setSalario, mas o que muda é o tipo do parâmetro desse método. O primeiro método setSalario recebe como parâmetro um valor (salário) do tipo double e armazena na variável. O segundo método setSalario recebe como parâmetro um valor (salário) do tipo String. Veja que o tipo de retorno dos métodos é igual, ou seja, void (não retorna nada).

Nesse caso, diz-se que o método setSalario está sobrecarregado (*overloading*).



Mas, para que serve isso? Por que devo **sobrecarregar** algum método da minha classe?

Quando projetamos e implementamos uma classe, devemos ter em mente que ela é um molde, um template, uma estrutura que não será usada somente para resolver o problema em questão inicial para o qual ela foi projetada, no nosso caso, cadastrar informações sobre funcionários. Uma vez que a classe foi implementada e compilada, ela pode ser usada para criar objetos a partir dela em milhares de outros programas (classes).

Nesse sentido, quanto mais métodos ou variáveis do mesmo método (sobrecarga) se tiver implementado, melhor será para as futuras utilizações dessa classe.

No nosso exemplo, resolvemos sobrecarregar somente o método `setSalario`. Uma versão dele recebe um valor de salário do tipo `double` e, outra versão, recebe um valor do tipo `String`. Isso quer dizer que, quando for necessário chamar o método `setSalario` para setar, modificar o valor do atributo salário, o programador que está utilizando a classe `Funcionário`, poderá escolher se deseja chamar o método `setSalario` e passar um valor do tipo `double` ou passar um valor do tipo `String`. A sobrecarga de método dá mais opções ao programador na hora de programar a chamada de métodos de uma classe.

Abaixo está o código da classe `Funcionário` com o método construtor sobrecarregado (vimos isso na unidade anterior) e com o método `setSalario` sobrecarregado.

```

Linha 1  public class Funcionario{
Linha 2      private String nome, endereco;
Linha 3      private double salario;
Linha 4
Linha 5      public Funcionario( ){
Linha 6          nome="";
Linha 7          endereco="";
Linha 8          salario=0;
Linha 9      }
Linha 10
Linha 11      public Funcionario( String snome,String sendere, double ssal){

```

Não existe limite para o número de vezes que um método pode ser sobrecarregado. Poderíamos sobrecarregar o método `setSalario` 2 ou 3 vezes se isso fosse necessário.

Método construtor sobrecarregado `Funcionario (String snome, String sendere, double ssal)`. Recebe como parâmetro o nome do funcionário na variável do tipo `String snome`, o endereço na variável `String sendere` e o salário na variável `double ssal` e armazena o conteúdo dessas variáveis nos respectivos atributos. Note que já existe outro método construtor, mas esse não recebe parâmetros. Isso quer dizer que o método construtor está sobrecarregado.

Método `setSalario(String ssalario)` sobrecarregado. Recebe como parâmetro o salário do funcionário na variável do tipo `String ssalario` e armazena o conteúdo dessa variável transformada para `double` no atributo `salario` do objeto. É necessário transformar porque o atributo `salario` só aceita valores do tipo `double`. Veja que já existe outro método `setSalario` só que este recebe como parâmetro um valor do tipo `double`.

```

Linha 12         nome = snome;
Linha 13         endereço = sendere;
Linha 14         salario = ssal;
Linha 15         }
Linha 16
Linha 17         public void setNome(String snome){
Linha 18             nome=snome;
Linha 19         }
Linha 20
Linha 21         public void setEndereco(String sender){
Linha 22             endereco=sender;
Linha 23         }
Linha 24
Linha 25         public void setSalario(double dsalario){
Linha 26             salario=dsalario;
Linha 27         }
Linha 28
Linha 29         public void setSalario(String ssalario){
Linha 30             salario = Double.parseDouble(ssalario);
Linha 31         }
Linha 32
Linha 33         public String getNome(){
Linha 34             return nome;
Linha 35         }
Linha 36
Linha 37         public String getEndereco(){
Linha 38             return endereco;
Linha 39         }
Linha 40
Linha 41         public double getSalario(){
Linha 42             return salario;
Linha 43         }
Linha 44     }

```

Depois de implementar a nova versão da classe `Funcionário`, compile-a.

Após ter sido compilada, vamos utilizar esse classe para criar objetos do tipo funcionário em outra classe chamada `InserFuncionario.java`.

Observe que **não é** a classe `CadastraFuncionario.java`

```

Linha 0  import javax.swing.*;
Linha 1  public class InserFuncionario{
Linha 2      public static void main(String args[]) {
Linha 3          Funcionario f=new Funcionario( "Ana","Rua A", 1000.00 );
Linha 4          JOptionPane.showMessageDialog(null,"Nome do Funcionário Cadastrado " + f.getNome());
Linha 5          System.exit(0);
Linha 6      }
Linha 7 }

```

Nessa classe, estamos criando um objeto do tipo `Funcionário`, na linha 3. Para isso, estamos usando um dos métodos construtores disponíveis na classe `Funcionário`. Qual deles está sendo chamado? O método construtor sem parâmetros ou o com parâmetros? O que espera três parâmetros!

E o outro método? Está lá na classe, disponível para ser usado em outra criação de objeto (porque é só nesse momento que um método construtor é chamado).

No caso dessa classe `InserFuncionario`, note que não precisamos chamar os métodos `setNome`, `setEndereco` porque os atributos do objeto `funcionário` criado já foram armazenados com a chamada desse método construtor, ou seja, na chamada do método construtor já foram passados os valores para o nome, endereço e salário do funcionário. A linha 4 só imprime o conteúdo do atributo nome do objeto criado. O nome impresso será "Ana".

A seguir temos a classe `CadastroFuncionario.java`, a que criamos no início dessa unidade. Note que ela foi modificada. Na linha 6, o método `setSalario` está sendo chamado e está sendo passado como parâmetro, um valor de salário do tipo `String`. Veja que o valor da entrada do usuário não está sendo transformado para `double` como nas versões anteriores.

Portanto, qual o método `setSalario` que está sendo chamado? O `setSalario` que espera um valor do tipo `double` ou o `setSalario` que espera um valor do tipo `String`?

O método `setSalario` que espera um valor do tipo `String`, porque esse valor é o que está sendo passado como parâmetro.

```

Linha 0    import javax.swing.*;
Linha 1    public class CadastroFuncionario{
Linha 2        public static void main(String args[]) {
Linha 3            Funcionario f=new Funcionario( );
Linha 4            f.setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 5            f.setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 6            f.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 7            JOptionPane.showMessageDialog(null,"Nome do Funcionário Cadastrado " + f.getNome( ));
Linha 8            System.exit(0);
Linha 9        }
Linha 10   }
```



Síntese

Nessa unidade você reforçou os conceitos de classe e objeto, bem como, a diferença entre referência e objeto, aprendidos nas unidades anteriores, porém, através de um novo exemplo.

Aprendeu passo a passo como modelar um problema do mundo real para o mundo orientado a objeto, identificando os objetos do problema, seus atributos e comportamentos e representando esses objetos através de uma classe.

Aprendeu também a desenvolver programas com a criação de mais de um objeto e manipulação desses objetos dentro de um vetor. Trabalhar com vetor de objeto será muito necessário daqui para frente, pois vamos começar a desenvolver programas que armazenam informações para vários objetos.



Atividades de auto-avaliação

- 1) Implemente o código explicado nessa unidade.

Modificadores



Objetivo de aprendizagem

- Aprender o conceito e a forma como se faz a aplicação dos modificadores de acesso e dos modificadores `static` e `final`.



Seção de estudo

Seção 1 Modificadores



Para início de conversa

Na unidade anterior você sedimentou os conceitos das unidades 7 e 8 (bases da OO) e aprendeu a criar programas que envolviam a criação de vários objetos usando para isso vetor de objetos. Aprendeu também o conceito de método sobrecarregado e a utilidade de utilizar esse tipo de recursos ao implementar uma classe.

Nessa unidade você irá aprender o conceito de modificadores de acesso (`public`, `private`, `protected` e sem modificador) e modificadores `static` e `final`.

Modificadores estão presentes desde a primeira classe que você desenvolveu, inclusive antes de aprender os conceitos de OO. Portanto, é importante que você saiba o significado de cada um deles e como utilizá-los.

SEÇÃO 1 - Modificadores

Na linguagem Java, podemos utilizar algumas palavras-chave (keyword) para modificar o modo como são declaradas classes, métodos e variáveis (atributos). Veremos como aplicar cada um desses modificadores no texto a seguir.

Modificadores de acesso a membros (`public`, `private`, `protected` e acesso de pacote)

São considerados **membros de uma classe** as variáveis de instância (atributos) e os **métodos** da classe.

Atributos também são conhecidos como variáveis de instância. Instância é o termo usado para designar um objeto.

Quando implementamos uma classe, sempre declaramos os seus membros (atributos ou métodos). No momento dessa declaração, utilizamos algum modificador de acesso. Os modificadores de acesso que podem ser utilizados são: **public**, **private**, **protected** e, quando nenhum modificador é especificado, se diz que o modificador de acesso é de **pacote** (**package**).

Veja o trecho de código da classe `Funcionário` da unidade anterior.


```

Linha 1  public class Funcionario{
Linha 2      private String nome, endereco;
Linha 3      private double salario;
Linha 4      ..
Linha 5      ..
Linha 6      ..
Linha 10     public void setNome(String snome){
Linha 11         nome=snome;
Linha 12     }

```

Declaração de todos os atributos com o modificador de acesso **private**.

Declaração de método com o modificador de acesso **public**.



Mas afinal, para que servem **os modificadores de acesso**?

É através dos modificadores de acesso, também conhecidos como **qualificadores de acesso**, que se controla o acesso aos membros da classe. Isso quer dizer que, os modificadores determinam **como**, ou melhor, de que lugar, esses membros (atributos e métodos) poderão ser acessados.



Importante!

Lembre-se que para acessar os membros de uma classe em outra classe, é necessário precedê-los com o nome da referência do objeto dessa classe.

Modificador private

No exemplo acima, todos os atributos declarados possuem o modificador **private**, o que determina que eles são privados à classe, ou seja, só podem ser acessados por métodos da própria classe Funcionário. Esses atributos não podem ser acessados diretamente pelo nome em outras classes, como mostra o código abaixo:

```

Linha 1  public class CadastroFuncionario{
Linha 2      public static void main(String args[]) {
Linha 3          Funcionario f=new Funcionario();
Linha 4          f.nome = "Ana";
Linha 5      }
Linha 6  }

```

Essa instrução está errada. Não podemos acessar o atributo nome de qualquer objeto do tipo Funcionário porque ele foi definido com o modificador de acesso **private**. Esse atributo nome só pode ser acessado diretamente pelo nome dentro da classe Funcionário.

Nesse caso, a única maneira de armazenar um valor dentro do atributo nome desse objeto é chamar o método público (com modificador **public**) `setNome()`.

A seguinte linha corrige a instrução acima:

Linha 4

`f.setNome("Ana");`



É através do modificador **private**, por exemplo, que se implementa um importante conceito da OO - o **ENCAPSULAMENTO**.

Uma boa prática de programação é tornar todos os atributos de uma classe **private**, pois isso encapsula, protege os atributos dos objetos que se originam da classe, permitindo que, só os métodos (geralmente os métodos `set`) da classe possam acessar e modificar os valores dessas variáveis.

Modificador **public**

Quando um membro de uma classe (atributo ou método) for declarado como modificador **public** ele poderá ser acessado, diretamente pelo nome, dentro da própria classe e por outras classes também.



Qual membro de uma classe nós queremos que seja acessado por outras classes? Atributos ou métodos?

Métodos, porque os atributos devem ser privados para promover o encapsulamento. E se os atributos devem ser todos privados, devemos ter métodos públicos para acessar esses atributos externamente, ou seja, em outras classes.

Portanto, todos os métodos `set` e `get` de uma classe devem ser **public** para que possam ser acessados por outras classes.

Diz-se que os métodos **public** de uma classe apresentam aos **clientes** de uma classe uma visualização dos **serviços** que a classe oferece (interface pública da classe).

Os **clientes** de uma classe são as pessoas (programadores) que usam uma classe para criar objetos a partir dela.

São chamados de **serviços** os métodos de uma classe.

Modificador de acesso de pacote

Chama-se de acesso de pacote quando não é explicitamente especificado um modificador para algum atributo ou método. Isso faz com que, outras classes do mesmo pacote (entendam pacote como uma pasta ou diretório) possam acessar os membros da classe em questão.

O termo **pacote** está relacionado a um conjunto de classes agrupadas em determinada estrutura de pastas ou diretórios.

Agrupam-se classes em pacotes com o objetivo de organizar essas classes em categorias.

Por exemplo, vamos imaginar que temos as seguintes classes: Retângulo, Triângulo e Trapézio. Cada uma dessas classes representa atributos e comportamentos de objetos: Retângulo, Triângulo e Trapézio, respectivamente.

Como essas três classes representam atributos e comportamentos de figuras da geometria plana, podemos organizá-las em um pacote (pasta) chamado GeometriaPlana.

A criação de pacotes não é meramente uma ação de criar uma pasta e copiar as classes para lá, mas envolve um método que será abordado oportunamente, mais à frente.

O que importa agora é você saber que, se não for especificado nenhum modificador em um atributo ou método da classe, ele terá modificador de **pacote**. Isso quer dizer que, somente as classes que estão no mesmo pacote dessa classe poderão acessar diretamente pelo nome os atributos e métodos dessa classe.

No trecho abaixo, os atributos declarados estão com modificador de acesso de pacote.

Linha 1	public class Funcionario{
Linha 2	String nome, endereco;
Linha 3	double salario;
..	..
..	..

.....

Atributos declarados sem modificador de acesso. São atributos com acesso de pacote.

O código abaixo ilustra o acesso direto (não recomendado) aos atributos de um objeto, criado a partir da classe acima.

```
Linha 1 public class CadastroFuncionario{  
Linha 2     public static void main(String args[]) {  
Linha 3         Funcionario f=new Funcionario( );  
Linha 4         f.nome = "Ana";  
  
Linha 5     }  
Linha 6 }
```

O atributo nome do objeto pode ser acessado diretamente nessa outra classe, mas isso não é recomendado!

Modificador protected

Esse modificador será abordado quando você aprender o conceito de **herança**.

Modificador final

O modificador **final** não é um modificador de acesso, ou seja, seu objetivo não é controlar o acesso a atributos e métodos de um objeto.

Na maioria das vezes, é desejado que os atributos (variáveis de instância) de um objeto tenham seus valores modificados. Porém, existem situações em que um atributo tem um valor fixo e esse valor não deve ser mudado. O atributo terá esse valor para todos os objetos que forem instanciados (criados) a partir dessa classe.

Quando essa situação ocorrer, devemos definir o atributo com o modificador **final**.

Vamos mostrar o uso desse modificador em outra classe que representa atributos e comportamento de objetos Círculo: a classe Círculo.

Todo objeto Círculo tem como atributos o raio e o pi. Porém, o valor de pi é fixo, constante, não deve mudar para cada objeto do tipo Círculo. Logo, o atributo pi é um candidato em potencial a ter um modificador **final**.

```
Linha 1    public class Circulo{  
Linha 2        private final double pi=3.14;  
Linha 3        private double raio;  
Linha 4  
Linha 5        public Circulo( ){  
                raio=0;  
            }  
            public Circulo(double draio){  
                raio=drario;  
            }  
            public void setRaio(double draio){  
                raio=draio;  
            }  
            public double getRaio(){  
                return raio;  
            }  
            public double getPi(){  
                return pi;  
            }  
            public double calculaArea( ){  
                return pi*raio*raio;  
            }  
        }  
    }
```

Note que não foi implementado o método `setPi()` pois não podemos alterar o valor do atributo `pi` já que ele foi definido como **final**.

Na linha 2, o atributo **pi** foi declarado com o modificador **final** e imediatamente inicializado com o seu valor fixo.

Qualquer atributo definido como **final**, deve ter seu valor atribuído na linha de definição do atributo, é obrigatório!

Uma tentativa dentro de algum método da própria classe `Circulo` de modificar o valor de um atributo **final** é capturada em tempo de compilação.

Quando um atributo de uma classe é definido como **final**, **todos** os objetos criados a partir daquela classe terão nesse atributo o valor fixo inicializado no momento da definição do atributo e, esse valor não poderá mudar.

Vejamos o código da classe a seguir que cria objetos do tipo `Circulo`.

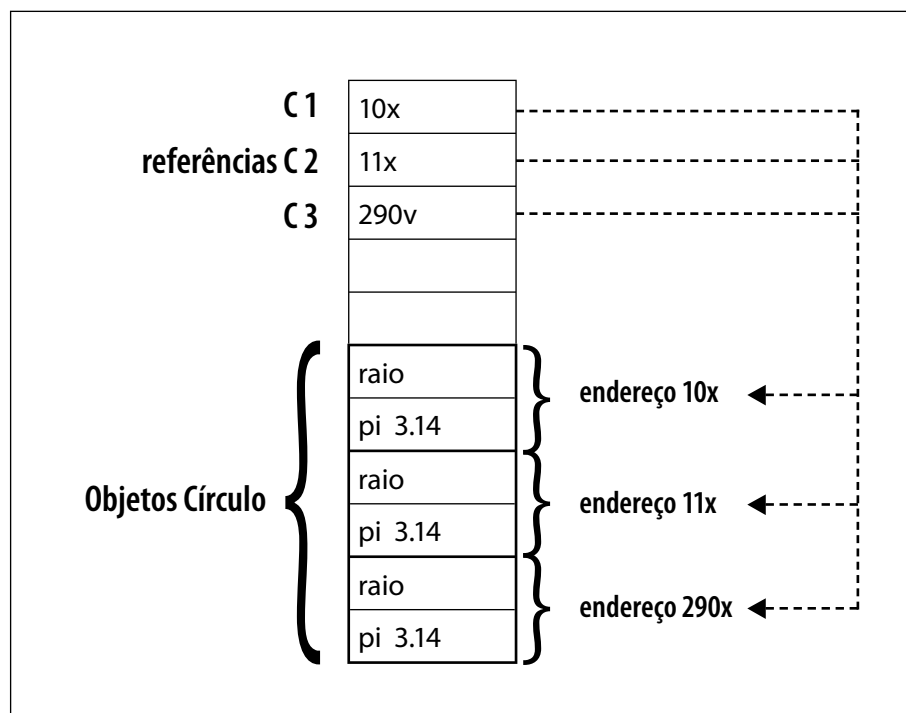
```

Linha 1  public class CalculaAreaCirculos{
Linha 2      public static void main(String args[]) {
Linha 3          Circulo c1 = new Circulo( );
Linha 4          Circulo c2 = new Circulo( );
Linha 5          Circulo c3 = new Circulo( );
Linha 6          c1.setRaio(5.0);
Linha 7          c2.setRaio(6.0);
Linha 8          c3.setRaio(7.0);
Linha 9      }
Linha 10 }

```

Vamos analisar o código:

Linhas 3 a 5: são criados objetos do tipo `Circulo`. Nesse momento, é criada a seguinte representação em memória de cada objeto.



Observe que o atributo `pi` de cada objeto já está com o valor 3.14.

Linhas 6 a 8: chamada ao método público `setRaio()` para cada objeto `Circulo` criado. Isso fará com que os valores de raio passado como parâmetro sejam armazenados no atributo `raio` de cada objeto `Circulo`.

O modificador **final** também pode ser aplicado aos métodos de uma classe, mas para entender o funcionamento, nesse contexto, é necessário que você aprenda o conceito de **herança**.

Modificador `static`

O modificador **static** não é um modificador de acesso, ou seja, seu objetivo não é controlar o acesso a atributos e métodos de um objeto. Sabemos que cada objeto instanciado a partir de uma classe tem sua própria cópia de todos os atributos (variáveis de instância) da classe.

Quando existir uma situação em que o valor de uma variável de instância puder ser compartilhado por todos os objetos instanciados a partir de uma classe, essa variável pode ser definida como **static**.

Declarar um atributo como **static** economiza memória, uma vez que esse valor (espaço na memória) é compartilhado por todos os objetos criados a partir da classe onde o atributo **static** foi criado.



Mas em qual situação vamos desejar que um atributo seja compartilhado por todos os objetos de uma classe?

Resposta: quando o valor desse atributo puder ser compartilhado por todos os objetos criados.

Na classe `Circulo`, implementada no tópico anterior, temos uma situação onde o valor de um atributo pode ser compartilhado por todos os objetos `Circulo`.

Que atributo é esse? - O atributo `pi`!

Veja a representação gráfica dos objetos criados (última figura).

O valor do atributo `pi` é o mesmo para todos os objetos `Circulo`, logo, ele pode ser compartilhado por todos os objetos `Circulo`. Portanto `pi`, além de **final**, pode ser também **static**.

Observe o trecho de código da classe `Circulo` onde o atributo `pi` é definido:

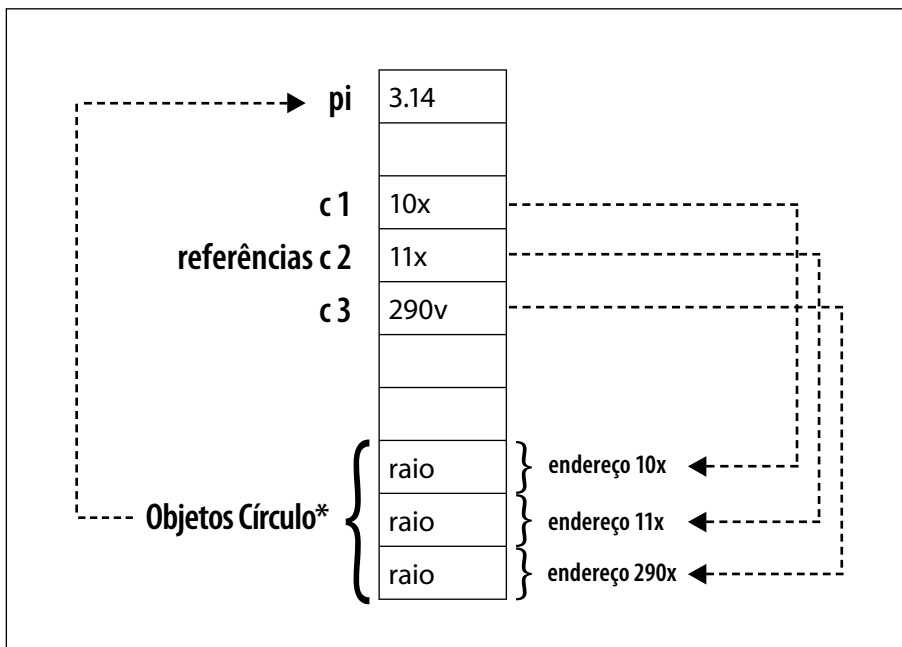
```
Linha 1  public class Circulo{  
Linha 2      private static final double pi=3.14;  
Linha 3      private double raio;  
Linha 4      ..
```

Veja agora como fica a representação em memória dos objetos `Circulo` criados com o código abaixo:

```
Linha 1  public class CalculaAreaCirculos{  
Linha 2      public static void main(String args[]) {  
Linha 3          Circulo c1 = new Circulo( );  
Linha 4          Circulo c2 = new Circulo( );  
Linha 5          Circulo c3 = new Circulo( );  
Linha 6          c1.setRaio(5.0);  
Linha 7          c2.setRaio(6.0);  
Linha 8          c3.setRaio(7.0);  
Linha 9          System.out.println(c2.getPi( ));  
Linha 10         System.out.println(c3.getPi( ));  
Linha 11     }  
Linha 12 }
```

Vamos analisar o código:

Linhas 3 a 5: são criados objetos do tipo `Circulo`. Nesse momento é criada a seguinte representação em memória de cada objeto.



Linhas 6 a 8: chamada ao método `setRaio()` para armazenar o valor de raio em cada objeto.

Linhas 9 e 10: para comprovar que qualquer objeto `Circulo` tem acesso ao atributo `static pi`, observe essa instrução. Através do método `getPi()` do objeto `c1`, temos acesso ao conteúdo do atributo `pi`. Ele é impresso na tela.

Na linha seguinte o mesmo acontece, só que o acesso se dá através do objeto `c2`. Isso comprova que, todos os objetos têm acesso indistinto ao conteúdo de `pi`, que é **static** e, nesse caso, também é **final**.

Um atributo que é **final** pode ser **static**, porém o inverso não se aplica, um atributo que é **static** nem sempre será **final**.

* Note que todos os objetos `Circulo` têm acesso ao atributo `pi`, que é **static**. É como se `pi` estivesse dentro do domínio de cada objeto. Tornar `pi` **static** é interessante porque ele não ocupa espaço em memória em cada objeto.

Métodos static

Métodos de uma classe também podem ser declarados com o modificador **static**, porém, se isso acontecer, esses somente poderão acessar atributos e outros métodos também **static**.

Isso acontece porque qualquer membro da classe (atributo ou método) declarado como **static**, pertence à classe e não a nenhuma instância (objeto) em particular dessa classe.

Vamos transformar o método `getPi()` da classe `Circulo` em **static**.

```
public static double getPi(){  
    return pi;  
}
```

Isso só é possível porque esse método acessa o atributo `pi`, que também é **static**.



Revisando: método **static** só pode manipular atributos e métodos também **static**.

Como os membros de uma classe (atributos ou métodos) declarados como **static** pertencem a classe, e não a nenhuma instância (objeto) em particular, eles podem ser acessados em outras classes da seguinte maneira:

```
Nomedaclassa.membrodaclassa
```

O código a seguir ilustra essa explicação:

```
Linha 1 public class CalculaAreaCirculos{  
Linha 2     public static void main(String args[]) {  
Linha 3         System.out.println ( Circulo.getPi ( ) );  
Linha 4     }  
Linha 5 }
```

Observe que, na linha 5, o método `getPi()` está sendo chamado precedido do nome da classe `Circulo` e não precedido da referência de um objeto, como sempre foi feito. Veja que não existe nenhum objeto `Circulo` criado anteriormente. Como o atributo `pi` é **static**, ele já está alocado em memória, mesmo que nenhum objeto `Circulo` tenha sido criado.

A instrução da linha 3 só pode acontecer porque o método `getPi()` é **static**.

Qual o valor que o método `getPi()` irá retornar?

3.14

É esse valor que será impresso na tela com o comando `System.out.println`.

Isso não impede que se chame os membros da classe declarados como `static` e que são `public` da maneira convencional, ou seja, através de uma referência de objeto.

nomedareferenciadeobjeto.membrodaclasse

O código abaixo ilustra essa explicação:

```
Linha 1 public class CalculaAreaCirculos{
Linha 2     public static void main(String args[]) {
Linha 3         Circulo c=new Circulo();
Linha 4         System.out.println ( c.getPi());
Linha 5     }
Linha 6 }
```

Note que o método `static getPi()` está sendo chamado precedido da referência ao objeto `Circulo`. Nós já usamos várias vezes, em nossos programas, métodos `static` de classes da biblioteca de classes da linguagem Java (API Java).

Exemplos:

```
JOptionPane.showInputDialog("Entre com o valor");
```

Observe a sintaxe da linha acima. Temos o nome de uma classe (JOptionPane) seguido de um . (ponto) seguido de método (showInputDialog()).

Essa é a maneira como métodos static são chamados, precedidos do nome da sua classe. Portanto, o método showInputDialog() é um método static da classe JOptionPane.

Observe que nunca precisamos criar uma objeto do tipo JOptionPane para chamar o método showInputDialog().

Outro exemplo são os métodos da classe Math. Essa também é uma classe da biblioteca de classes da linguagem Java. Essa classe possui um conjunto de métodos matemáticos que podemos usar da seguinte maneira:

```
Math.pow(2,3).
```

Como pow(..) é um método **public static** da classe Math, não precisa ser acessado via uma referência de objeto, ou seja, não é preciso instanciar um objeto dessa classe para utilizar o método pow(..). Esse método calcula a potência de um número.

O primeiro argumento passado é o 2 que se refere à base e, o segundo argumento passado é o 3, que se refere ao expoente. Logo, esse método irá retornar o valor de 2^3 .

Os membros de uma classe que são declarados como static existem, mesmo que nenhum objeto dessa classe seja instanciado. Eles estão disponíveis em tempo de execução quando a classe é carregada para a memória.

Para fortalecer o conceito de modificador **static**, vamos trabalhar com outro exemplo. Modifique a classe Funcionário e insira o código a seguir:

```

Linha 1      public class Funcionario{
Linha 2          private String nome, endereco;
Linha 3          private double salario;
Linha 4          private static int cont;
Linha 5
Linha 6          public Funcionario(){
Linha 7              nome="";
Linha 8              endereco="";
Linha 9              salario=0;
Linha 10         cont=cont + 1;
Linha 11         }
Linha 12
Linha 13         public void setNome(String snome){
Linha 14             nome=snome;
Linha 15         }
Linha 16
Linha 17         public void setEndereco(String sender){
Linha 18             endereco=sender;
Linha 19         }
Linha 20
Linha 21         public void setSalario(double dsalario){
Linha 22             salario=dsalario;
Linha 23         }
Linha 24         public String getNome(){
Linha 25             return nome;
Linha 26         }
Linha 27
Linha 28         public String getEndereco(){
Linha 29             return endereco;
Linha 30         }
Linha 31
Linha 32         public double getSalario(){
Linha 33             return salario;
Linha 34         }
Linha 35
Linha 36         public static int getCont(){
Linha 37             return cont;
Linha 38         }
Linha 39     }

```

Temos um atributo static chamado cont.

Cada vez que o construtor dessa classe for chamado, o atributo cont irá incrementar em 1. Quando um método construtor é chamado? Sempre que um objeto do tipo Funcionário for criado.

Método static getCont() retorna o conteúdo do atributo static cont.



Você entendeu a função do atributo `cont` na classe `Funcionário`?

Ele é `static`, logo, ele não estará “dentro” de cada objeto `Funcionário` criado. Ele existirá apenas uma vez na memória, mas todos os objetos poderão acessá-lo, modificando ou recuperando seu valor.



Existe algum método `setCont ()` para alterar o valor do atributo `cont`?

Não! O valor do atributo `cont` é alterado (incrementado em 1) somente dentro do método construtor.

Com essas características concluímos que o atributo `cont` tem a função de armazenar o número de objetos `Funcionário` criados.

No código abaixo estamos criando vários objetos `Funcionário`:

```

Linha 1      public class ContaFuncionario{
Linha 2          public static void main(String args[]){
Linha 3              System.out.println(Funcionário.getCont( ));

Linha 4              Funcionario f1= new Funcionario( );
Linha 5              Funcionario f2= new Funcionario( );
Linha 6              System.out.println(Funcionario.getCont());

Linha 7              System.out.println(f1.getCont());

Linha 8              System.out.println(f2.getCont());

Linha 9          }
Linha 10     }
```

Vamos analisar o código linha a linha:

Linha 3: chamada ao método `public static getCont()` que retornará o valor 0 porque, nesse momento, o atributo `static cont` está em memória com o valor 0. Note que o método é chamado precedido com o nome da classe. Isso só pode acontecer porque o método é `static`.

A figura abaixo ilustra a representação em memória depois da execução dessa linha. O atributo static cont está em memória com valor 0.

0

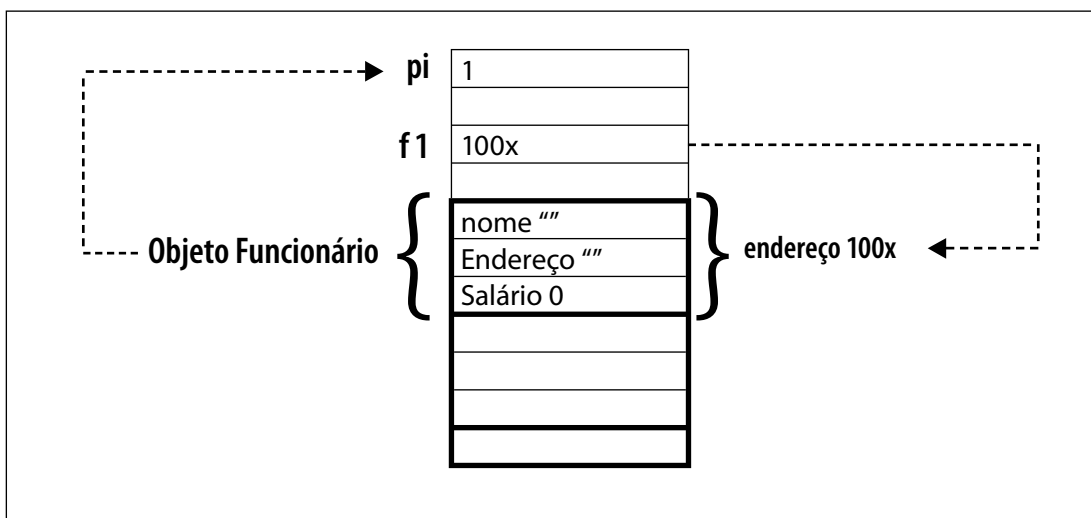
Linha 4: Criação do objeto Funcionário f1. No momento da criação de um objeto, sempre um método construtor é chamado, nesse caso, o método construtor Funcionário () é chamado e é executado o código que está dentro dele. Uma instrução que está dentro do construtor é a seguinte:

<pre>cont = cont + 1;</pre>

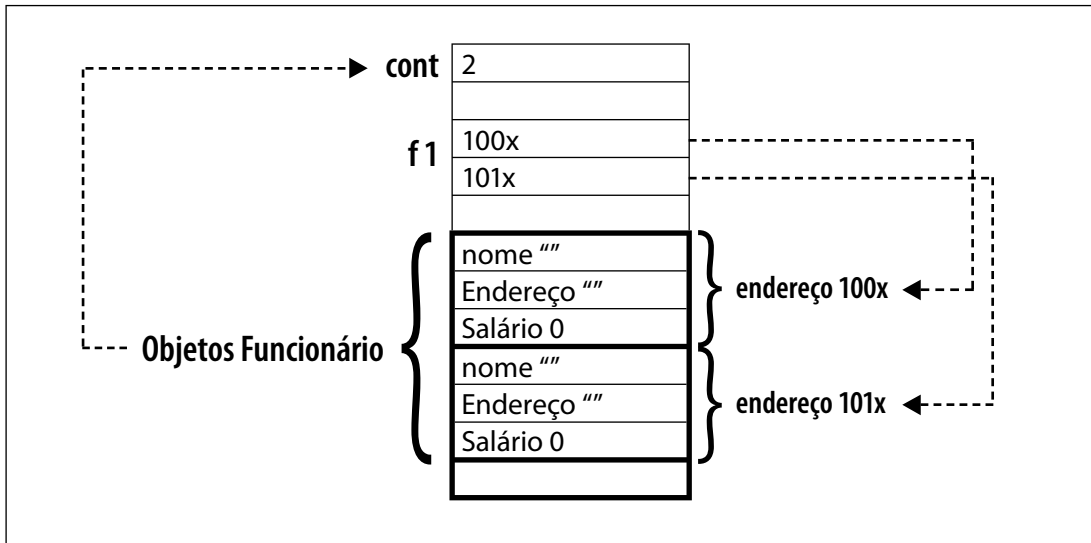
Isso faz com que o atributo cont armazene + 1. Logo, o seu valor no momento é 1.

A figura ilustra a representação em memória depois da execução dessa instrução.

O objeto f1 é criado, seu método construtor é chamado e incrementa o valor do atributo cont.



Linha 5: criação do objeto Funcionário f2. No momento da criação de um objeto, sempre um método construtor é chamado e, assim, é executado o que foi programado dentro dele. Logo, a instrução que incrementa `cont` é executada e seu valor do momento é 2.



Linha 6: chamada ao método public static `getCont()` que retornará o valor 2 porque, nesse momento, o atributo static `cont` está em memória com esse valor, depois de ter criado dois objetos Funcionário.

Linha 7: um método static também pode ser chamado precedido da referência a um objeto. Nessa linha, o método static `getCont()` é chamado precedido da referência `f1` e retorna o valor 2, que é o valor do atributo `cont`.

Linha 8: chamada do método static `getCont()` precedido da referência `f2`. Lembre-se que todos os objetos podem acessar o atributo `pi` através do método `getCont()`. O valor retornado também será 2.



Síntese

Nessa unidade você aprendeu sobre modificadores. Viu que existem os chamados modificadores de acesso (`public`, `private`, `protected` e `pacote`) e que existem os modificadores `static` e `final`. Modificadores podem ser aplicados a uma classe, atributo ou método.

É importante que você entenda porquê usava a palavra `public`, `private`, `static` nos programas anteriores.

Agora isso se tornou possível.

Vimos nessa unidade as aplicações mais importantes desses modificadores, contudo, existem outras possibilidades de aplicação dos modificadores.



Atividades de auto-avaliação

- 1) Analise o seguinte trecho de código e identifique se existe algum erro. Caso exista, explique qual o erro e como solucioná-lo.

```
//classe Retangulo
public class Retângulo{
    private double base, altura;

    //métodos

}

//classe calcula que cria objeto retangulo
public class CalculaRetangulo{

    public static void main(String args[]){
        Retangulo r = new Retangulo( )
        r.base = 5.0;
        r.altura = 6.0
    }
}
```

- 2) A seguinte classe tem dois métodos static. Implemente outra classe chamada UsaCalculos.java e demonstre como esses métodos static podem ser chamados dessa classe.

```
public class Calculos{  
    public static int potencia(int base, int exp){  
        int pote=1;  
        for (int i=1;i<=exp;i++){  
            pote = pote * base;  
        }  
        return pote;  
    }  
  
    public static int fatorial(int nu){  
        int fat=1;  
        for (int i=1;i<=nu;i++){  
            fat = fat * nu;  
        }  
        return fat;  
    }  
}
```

UNIDADE 11

11

Objetos como atributos de outros objetos



Objetivo de aprendizagem

- Entender o conceito de associação.



Seção de estudo

Seção 1 Associação



Para início de conversa

Na unidade anterior você estudou o conceito de modificador e viu que existem vários tipos de modificadores: os de acesso, `static` e `final` e as implicações de se definirem atributos e métodos com esses modificadores. Nessa unidade, você irá estudar que, ao representar atributos e comportamentos de um determinado tipo de objeto através de uma classe, podem existir atributos (nessa classe) que são de um outro tipo de objeto que também está representado através de uma classe. Quando isso acontece, existe um relacionamento entre esses tipos de objetos diferentes (no mínimo dois), conseqüentemente, entre as classes que os representam, que será chamado, nessa unidade, de associação.

O exemplo prático que daremos a seguir irá facilitar seu entendimento sobre o conceito de associação.

SEÇÃO 1 – Associação

Ao modelar os atributos da classe `Funcionário` (lembre-se que ela representa os atributos e comportamentos de objetos `Funcionário`) podemos definir que o setor em que o funcionário trabalha é uma informação importante.



Poderíamos definir essa informação como um atributo do tipo **String**.

Também podemos pensar que essa informação é um objeto do problema (cadastrar informações sobre funcionários). Esse objeto possui atributos como código e nome do setor ao ser representado através de uma classe, possivelmente chamada de `Setor`, pode ser usado como atributo de outros objetos/Classe ou em outros programas.

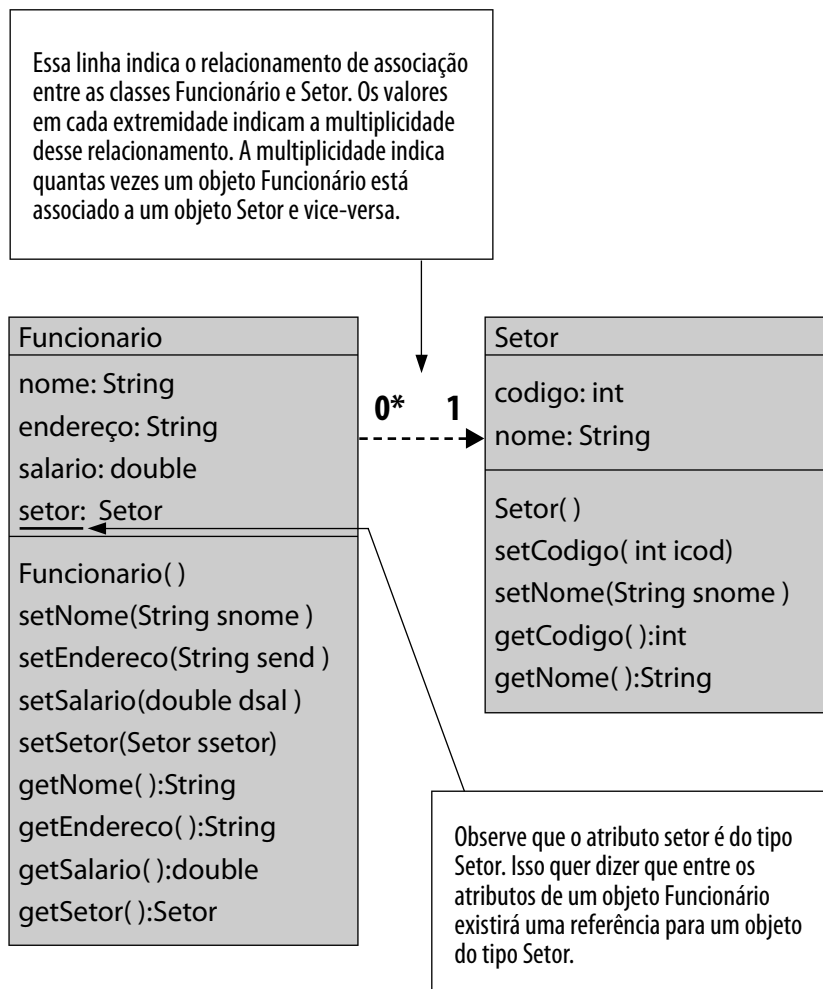
Ao pensar em setor como um objeto, devemos representar esse objeto através de uma classe logo teremos mais uma classe no

nosso problema. Ela pode ser chamada de Setor, pois representa os atributos e comportamentos de qualquer objeto Setor.

Teremos, portanto, duas classes para representar os objetos do problema: Funcionário e Setor.

O que ocasionou a criação da classe Setor foi pensarmos no atributo setor como um objeto. Portanto, se setor é um atributo de Funcionário e ele é do tipo Setor, a classe Funcionário possui um relacionamento com a classe Setor. Vamos chamar esse relacionamento de **Associação**.

A figura abaixo ilustra, em UML, o relacionamento de associação entre a classe Funcionário e a classe Setor.



Como observado na figura acima, a **multiplicidade** indica o número de vezes que um objeto se relaciona com outro objeto, ou o número referência ou é referenciado por outro objeto.

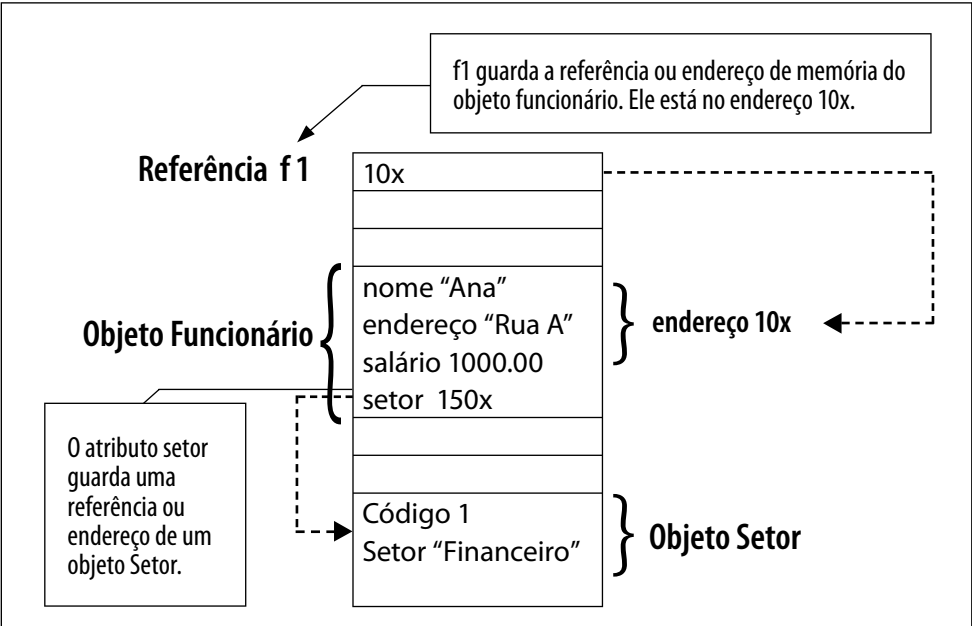
Por exemplo: na figura anterior, um objeto Funcionário está associado a somente um objeto Setor (veja cardinalidade 1 na extremidade direita). Isso quer dizer que um objeto Funcionário fará referência a, no máximo, 1 objeto Setor. Por outro lado, um objeto Setor pode ser referenciado por vários objetos do tipo Funcionário.

A seta no sentido da direita indica que somente o objeto Funcionário tem conhecimento do objeto Setor.

A tabela abaixo indica os diversos tipos de indicadores de multiplicidade.

Indicadores de multiplicidade	
Muitos	*
Exatamente um	1
Zero ou mais	0..*
Um ou mais	1..*
Zero ou um	0..1
Faixa especificada	2..4, 6..8

Antes de demonstrar como ficará o código da classe Funcionário com o novo atributo setor, vamos representar graficamente um objeto f1 do tipo Funcionário em memória, para melhorar a sua compreensão sobre esse conceito. O objeto está com todos os atributos preenchidos.



Primeiramente, vamos implementar a classe Setor.

Posteriormente, vamos implementar a nova versão da classe Funcionário, agora com o atributo setor que é do tipo Setor.

Quando implementamos uma classe que possui algum atributo ou faz referência à outra classe, essa outra classe já deve ter sido implementada, por isso, vamos implementar primeiro, a classe Setor.

É uma implementação de classe simples, como você já fez antes.

```
Linha 1      public class Setor{
Linha 2          private int codigo;
Linha 3          private String nomesetor;
Linha 4
Linha 5          public Setor( ){
Linha 6              codigo=0;
Linha 7              nomesetor="";
Linha 8          }
Linha 9
Linha 10         public Setor(int icod, String snomeset ){
Linha 11             codigo=icod;
Linha 12             nomesetor=snomeset;
Linha 13         }
Linha 14         public void setCodigo(int icod){
Linha 15             codigo=icod;
Linha 16         }
Linha 17
Linha 18         public void setNomeSetor(String snomeset){
Linha 19             nomesetor= snomeset;
Linha 20         }
Linha 21
Linha 22         public int getCodigo(){
Linha 23             return codigo;
Linha 24         }
Linha 25
Linha 26         public String getNomeSetor(){
Linha 27             return nomesetor;
Linha 28         }
Linha 29
Linha 30     }
```

Implementação da classe Funcionário, agora com o atributo setor.

O atributo setor que é do tipo Setor, ou seja, fará referência a um objeto do tipo Setor em memória deve ser private como os demais atributos.

o atributo setor é inicializado com o valor null. Todo atributo/objeto deve ser inicializado com esse valor. Isso quer dizer que no momento ele não faz referência a nenhum objeto.

Método setSetor(Setor ssetor) recebe como parâmetro a referência ou endereço de um objeto Setor. Essa referência é armazenada no atributo setor.

Método getSetor() retorna o valor do atributo setor que é uma referência ou endereço do objeto Setor armazenado nela. Como a referência retornada é do tipo Setor o tipo retornado é Setor

Esse método retorna o valor do atributo nomesetor do objeto Setor a que o atributo setor faz referência. Isso irá ficar mais claro em seguida.

Implementação da classe Funcionário.java

```

Linha 1      public class Funcionario
Linha 2      {
Linha 3          private String nome, endereco;
Linha 4          private double salario;
Linha 5          private Setor setor;
Linha 6
Linha 7      public Funcionario( )
Linha 8      {
Linha 9          nome="";
Linha 10         endereco="";
Linha 11         salario=0;
Linha 12         setor = null;
Linha 13     }
Linha 14     public void setNome(String snome)
Linha 15     {
Linha 16         nome=snome;
Linha 17     }
Linha 18
Linha 19     public void setEndereco(String sender)
Linha 20     {
Linha 21         endereco=sender;
Linha 22     }
Linha 23
Linha 24     public void setSalario(double dsalario)
Linha 25     {
Linha 26         salario=dsalario;
Linha 27     }
Linha 28     public String getNome()
Linha 29     {
Linha 30         return nome;
Linha 31     }
Linha 32
Linha 33     public String getEndereco()
Linha 34     {
Linha 35         return endereco;
Linha 36     }
Linha 37
Linha 38     public double getSalario()
Linha 39     {
Linha 40         return salario;
Linha 41     }
Linha 42
Linha 43     public void setSetor (Setor ssetor)
Linha 44     {
Linha 45         setor = ssetor;
Linha 46     }
Linha 47
Linha 48     public Setor getSetor( )
Linha 49     {
Linha 50         return setor;
Linha 51     }
Linha 52
Linha 53     public String getNomeSetorFunc( )
Linha 54         return setor.getNomeSetor();
Linha 55     }
Linha 56 }
```

Depois de implementar a classe Funcionário e a classe Setor, vamos desenvolver um programa de ‘cadastre as informações’ de alguns Funcionários que agora possuem um setor associado.

Implementação da classe CadastroFuncionarioSetor.java

```

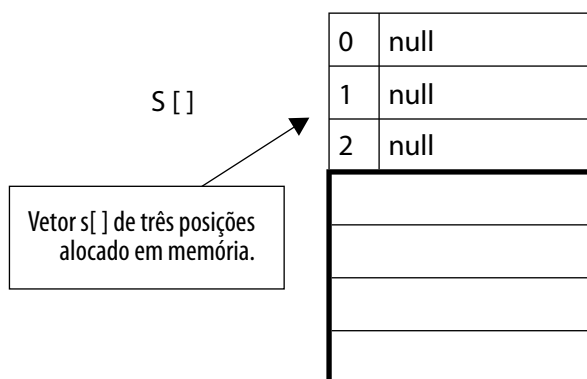
Linha 0  import javax.swing.*;
Linha 1  public class CadastroFuncionarioSetor{
Linha 2  public static void main(String args[]) {
Linha 3      Setor s[]=new Setor[3];
Linha 4      s[0] = new Setor(1,"Financeiro");
Linha 5      s[1] = new Setor(2,"Compras");
Linha 6      s[2] = new Setor(3,"Almoxarifado");
Linha 7      Funcionario f=new Funcionario( );
Linha 8      f.setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 9      f.setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 10     f.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 11     int codsetor=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Código do Setor"));
Linha 12     for (int i=0;i<3;i++){
Linha 13         if (codsetor == s[i].getCodigo( ))
Linha 14             f.setSetor(s[i]);
Linha 15 }
Linha 16     Funcionario f2=new Funcionario( );
Linha 17     f2.setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 18     f2.setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 19     f2.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 20     codsetor=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Código do Setor"));
Linha 21     for (int i=0;i<3;i++){
Linha 22         if (codsetor == s[i].getCodigo( ))
Linha 23             f2.setSetor(s[i]);
Linha 24 }
Linha 25     Funcionario f3=new Funcionario( );
Linha 26     f3.setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 27     f3.setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 28     f3.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salário")));
Linha 29     codsetor=Integer.parseInt(JOptionPane.showInputDialog("Entre com o Código do Setor"));
Linha 30     for (int i=0;i<3;i++){
Linha 31         if (codsetor == s[i].getCod( ))
Linha 32             f3.setSetor(s[i]);
Linha 34 }
Linha 35     JOptionPane.showMessageDialog(null,"Nome e Setor dos Funcionários \n" + f.getNome( )
        + " " + f.getNomeSetorFunc( ) + "\n" + f2.getNome( ) + " " + f2.getNomeSetorFunc( )
        + "\n" + f3.getNome( ) + " " + f3.getNomeSetorFunc( ) );
Linha 37     System.exit(0);
Linha 38 }
Linha 39 }

```

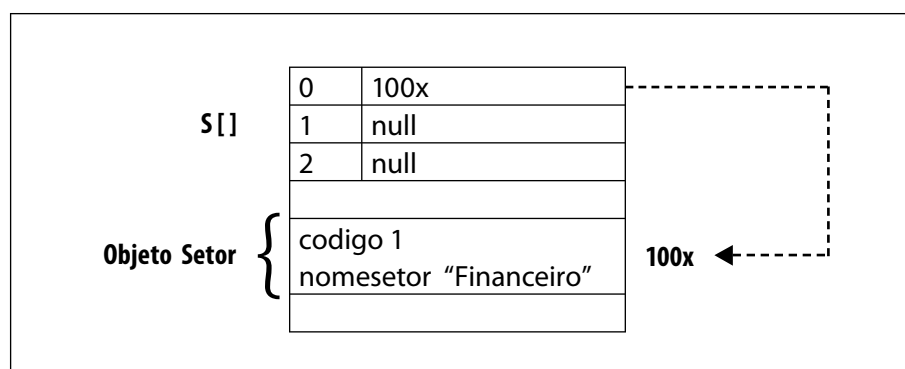
Nesse trecho de código, o código do setor digitado pelo usuário é procurado no vetor s[] que contém três objetos Setor. Quando encontrar um objeto Setor, com o atributo código igual ao código digitado pelo usuário, a referência (endereço) desse objeto é passada como parâmetro para o método setSetor() de Funcionario. Essa referência é armazenada no atributo setor do objeto Funcionario f.

Vamos analisar o código acima, linha a linha:

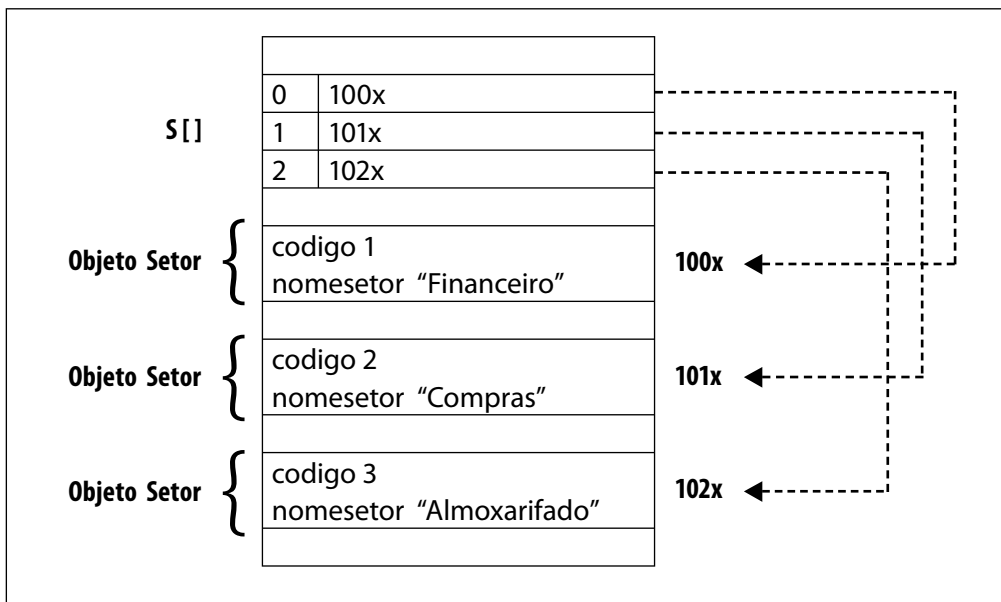
Linha 3: Criação de um vetor chamado `s[]` para armazenar endereços de objetos do tipo `Setor`. Após a criação do vetor, cada uma das três posições estará inicializada com o valor `null`.



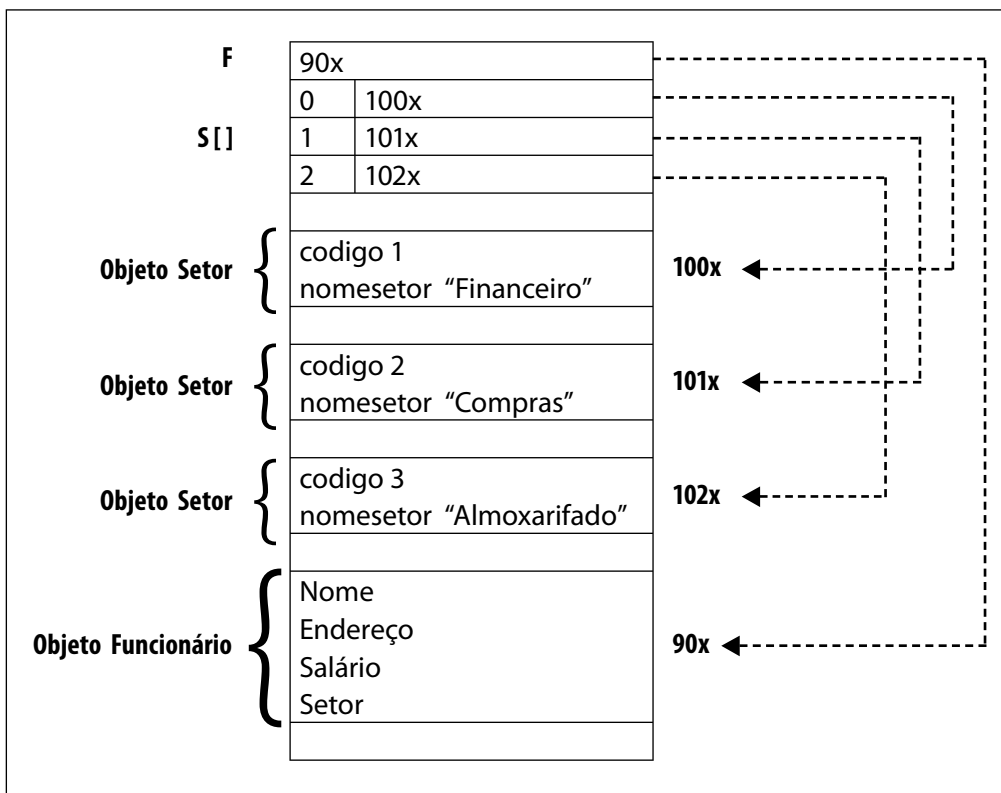
Linha 4: armazena na posição 0 do vetor `s[]` a referência (endereço) do objeto `Setor` que está sendo criado com a instrução: `new Setor(1, "Financeiro")`. Observe que o objeto `Setor` está sendo criado, e o seu método construtor com parâmetros está sendo chamado. Conforme implementação desse método construtor os dois valores passados como parâmetros estão sendo armazenados nos respectivos atributos `código` e `nomesetor`.



Linhas 5 e 6: armazena na posição 1 e 2 do vetor `s[]` a referência (endereço) de mais dois objetos `Setor` que estão sendo criados com as respectivas instruções: `new Setor(2, "Compras")` e `new Setor(3,"Almoxarifado")`. Na criação desses dois objetos `Setor`, novamente o método construtor com parâmetros está sendo chamado. Chamar o método construtor com parâmetro é interessante quando se quer criar um objeto em memória e já inicializar seus atributos com valores.



Linha 7: Depois dos objetos Setor terem sido criados, ou seja, estarem alocados em memória, vamos começar a criar os objetos funcionários do nosso problema para, finalmente, podermos armazenar as informações de funcionário. Nessa linha é criado um objeto f do tipo Funcionário.



Linhas 8 a 10: Depois que o objeto `f` for criado, são chamados seus métodos `setNome()`, `setEndereco()` e `setSalário()` para armazenarem os valores digitados pelo usuário nos atributos `nome`, `endereco` e `salário`.

Linha 11: o usuário digita o código do setor em que o funcionário trabalha e ele é armazenado na variável de memória `codsetor`. Esse código de setor não pode ser armazenado no atributo `setor` do objeto `funcionário`, porque esse atributo não é do tipo `int` e sim, do tipo `Setor`.

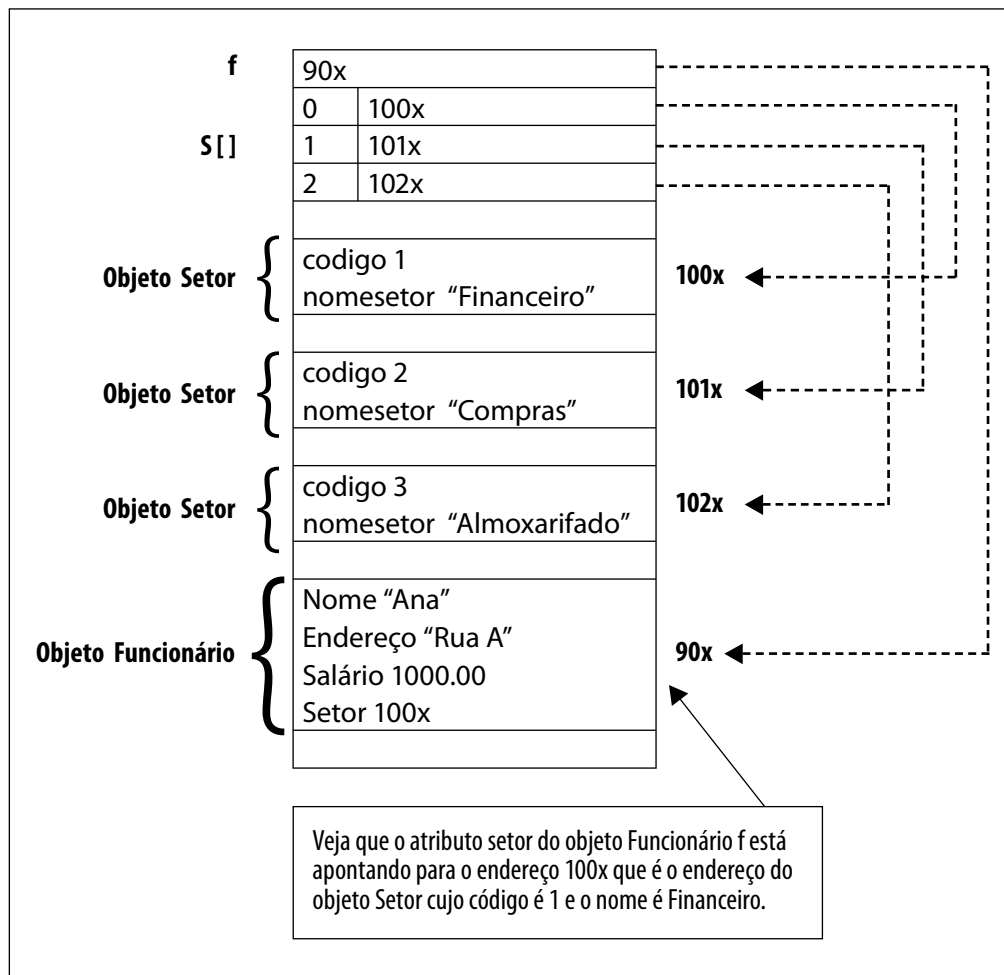
Linha 12: Depois do código do setor digitado e armazenado na variável `codsetor`, iremos procurar um objeto `Setor` no vetor `s[]` que tenha no atributo `código` um valor igual ao que o usuário digitou e que está armazenado na variável `codsetor`. Para isso, usaremos uma estrutura de repetição `for` para percorrer todos os objetos `Setor` armazenados no vetor `s[]`. Note que a condição de repetição começa em 0 e vai até um valor menor que 3, ou seja, 2. Isso porque vetor `s[]` só possui três posições.

Linha 13: é verificado, testado, se o atributo `código` de cada objeto `Setor` que está no vetor `s[]` é igual ao conteúdo da variável `codsetor` (aqui está o código de setor que o usuário digitou). O código de cada objeto `Setor` é recuperado com seu método `getCodigo()`.

Linha 14: se for encontrado um objeto `Setor` com código igual ao código digitado pelo usuário e que está armazenado na variável `codsetor`, a referência desse objeto `Setor` é passada como parâmetro para o método `setSetor` do objeto `Funcionário` que está sendo cadastrado no momento. Note o que está em `s[i]`, sendo que `i` pode ser 0, 1 ou 2 é uma referência para um objeto `Setor`, por isso que `s[i]` está sendo passado como parâmetro para o método `setSetor`. Na implementação desse método (ver linha 35 da classe `Funcionário`) essa referência é armazenada no atributo `setor` do objeto `Funcionário`.

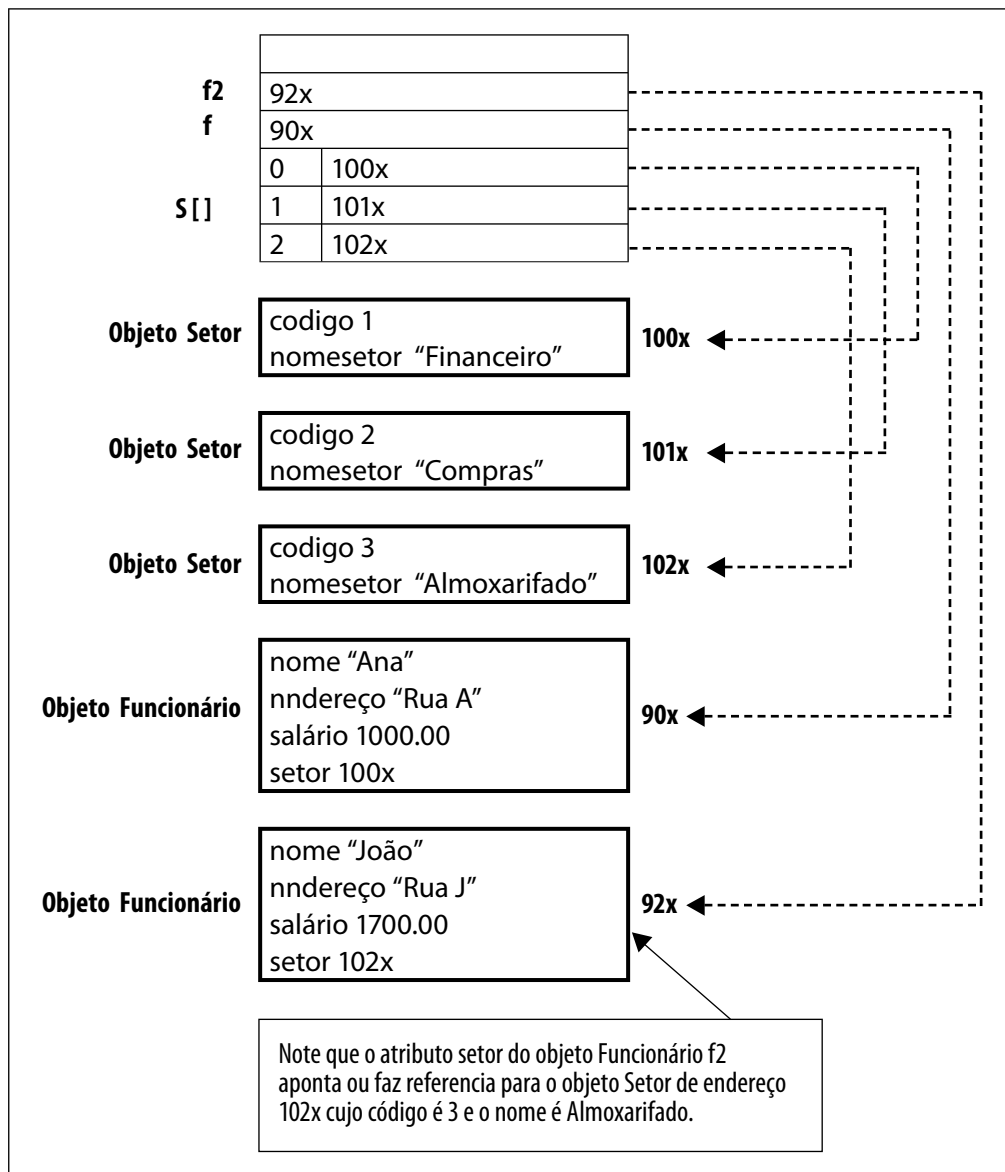
No nosso exemplo, o usuário irá digitar o código de setor 1 para o primeiro funcionário cadastrado (significa que o funcionário trabalha no setor Financeiro). A rotina das linhas 12 a 14 irá procurar esse código em algum objeto `Setor` do vetor `s[]`. Ele será encontrado no objeto `Setor` que está na primeira posição do vetor `s[]`. Observem na figura abaixo que a primeira posição

do vetor `s[]` aponta para um objeto Setor, cujo código é 1 logo, a referência ou endereço que está nessa posição, será passado como parâmetro para o método `setSetor()` e será armazenada no atributo `setor`. Veja isso na figura abaixo: o atributo `setor` do objeto `Funcionário` está com o valor de referência (100x) para o objeto Setor cujo código é 1 (Financeiro).



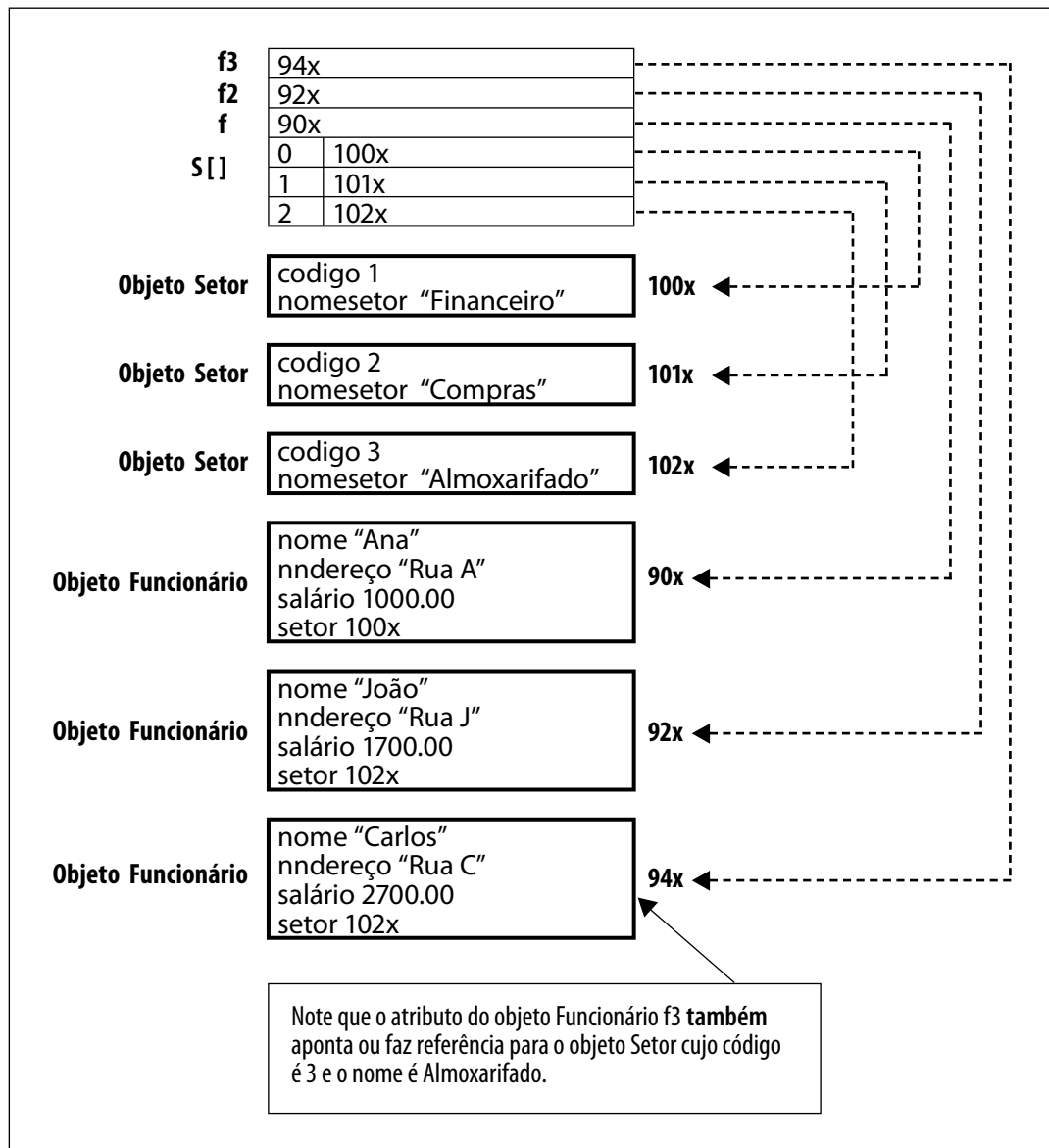
Linhas 16 e 24: é criado um outro objeto `Funcionário` `f2` e o mesmo procedimento descrito para o objeto `Funcionario` `f`, é executado. O funcionário `f2` trabalha no setor de Almoxarifado, logo será digitado o código 2, e as rotinas das linhas 21 a 24 irão procurar no vetor `s[]` se existe algum objeto Setor cujo código é igual ao código 2 digitado pelo usuário e que foi armazenado na variável `codsetor`. Esse código será encontrado no objeto Setor, que está sendo referenciado na posição 2 do vetor `s[]` logo, a referência para o objeto Setor ali armazenada será passada como parâmetro para o método `setSetor()` e será armazenada no

atributo setor do objeto Funcionário f2, caracterizando, assim, que esse funcionário trabalha no setor de Almoxarifado. A figura abaixo ilustra esse processo.



Linhas 25 e 34: é criado o terceiro objeto Funcionário f3 e o mesmo procedimento descrito para os objetos Funcionario f e f2 é executado. O funcionário f3 também trabalha no setor de Almoxarifado logo será digitado o código 2, e a rotina das linhas 30 a 34 irá procurar no vetor `s[]` se existe algum objeto Setor, cujo código é igual ao código 2 digitado pelo usuário e que foi armazenado na variável `codsetor`. Esse código também

será encontrado no objeto Setor que está sendo referenciado na posição 2 do vetor `s[]` logo a referência para o objeto Setor ali armazenada será passada como parâmetro para o método `setSetor()` e será armazenada no atributo `setor` do objeto Funcionário `f3`, estabelecendo assim, que esse funcionário também trabalha no setor de Almoxarifado. Como você deve ter notado, teremos dois funcionários que trabalham no mesmo setor, portanto, que compartilham do mesmo objeto Setor. A figura abaixo ilustra esse processo.



Esse último cadastro de funcionário explicita multiplicidade 0..* (nenhum ou muitos) da classe Setor com a classe Funcionario.



Vamos revisar: essa multiplicidade indica que um objeto Setor pode ser referenciado por nenhum ou por vários objetos Funcionário.

E é isso que acabamos de demonstrar com o cadastro do funcionário f3, já que o seu atributo setor também faz referência ao objeto Setor, cujo código é 3 e que já tinha sido referenciado pelo segundo funcionário, já que ambos trabalham no setor Almoxarifado.

Linha 35: nessa linha, é impresso na tela no nome e setor dos três funcionários cadastrados. Note que o nome é recuperado através do método `getNome()` do objeto Funcionário e o nome do setor é recuperado através do método `getNomeSetorFunc()`, que também foi implementado na classe Funcionário. Vamos revisar a implementação desse método aqui.

```
public String getNomeSetorFunc()
    return setor.getNomeSetor()
}
```

Observe que quando esse método é chamado para um determinado objeto Funcionário, será acessado o valor do atributo setor desse objeto Funcionário.

Exemplo: se esse método for chamado para o objeto Funcionário f2 o valor do atributo setor será 102x. Essa é a referência ou endereço de um objeto Setor. Logo, a instrução `setor.getNomeSetor()` recupera o nome do setor do objeto que está no endereço 102x, no nosso caso, Almoxarifado.

A saída em tela será como a listagem a seguir:

Nome e Setor dos Funcionários	
Ana	Financeiro
João	Almoxarifado
Carlos	Almoxarifado



Síntese

Nesta unidade você aprendeu o conceito de relacionamento entre classes. O relacionamento demonstrado nessa unidade se chama Associação.

Na maioria dos sistemas orientados a objetos que você irá construir, existirão diversas classes representando atributos e comportamentos dos objetos do problema. Esses objetos, muito provavelmente, não estarão isolados, ou seja, eles irão se relacionar. Uma classe poderá se relacionar com várias outras classes.

Uma das formas de relacionamento é a Associação.

Nela, um dos objetos, ou ambos, apenas conhecem o outro objeto.



Atividades de auto-avaliação

- 1) Faça uma nova versão da classe Funcionário e inclua um novo atributo: cep.

O cep irá indicar a rua (logradouro) e bairro do funcionário. O atributo endereço que já existe para o funcionário pode armazenar o complemento do endereço como o número, bloco, apto, etc.

Você deve pensar e modelar cep com um objeto do tipo Cep. Esse objeto tem os seguintes atributos: código (cep), rua (logradouro) e o bairro.

Um funcionário está associado a somente um (1) cep, mas um cep pode estar associado a vários funcionários.

Portanto, existirá um relacionamento de Associação entre Funcionário e Cep, semelhante ao relacionamento trabalhado nessa unidade entre Funcionário e Setor.

Modele as classes do sistema em UML.

Crie um programa para cadastrar 5 funcionários e após o cadastro exibir nome, rua e bairro de todos os funcionários.

UNIDADE 12

12

Associação na prática



Objetivo de aprendizagem

- Fortalecer o conceito de associação.



Seção de estudo

Seção 1 Associação por exemplos



Para início de conversa

Na unidade anterior você aprendeu que, ao representar atributos e comportamentos de um determinado tipo de objeto através de uma classe, podem existir atributos (nessa classe) que são de um outro tipo de objeto, que também está representado através de uma classe.

Isso caracteriza um relacionamento entre esses tipos de objetos diferentes (no mínimo dois), conseqüentemente, entre as classes que os representam, esse relacionamento é chamado de associação.

Nessa unidade iremos fortalecer esse conceito através de outro exemplo prático.

SEÇÃO 1 - Associação por exemplos

Vamos relembrar o problema do sistema de contas correntes, apresentado nas unidades 7 e 8, em que você começou a aprender os primeiros conceitos de O.O.

Nesse sistema tínhamos a classe Cliente e a classe Conta.

Obviamente, elas representam objetos do tipo Cliente e objetos do tipo Conta nesse sistema.

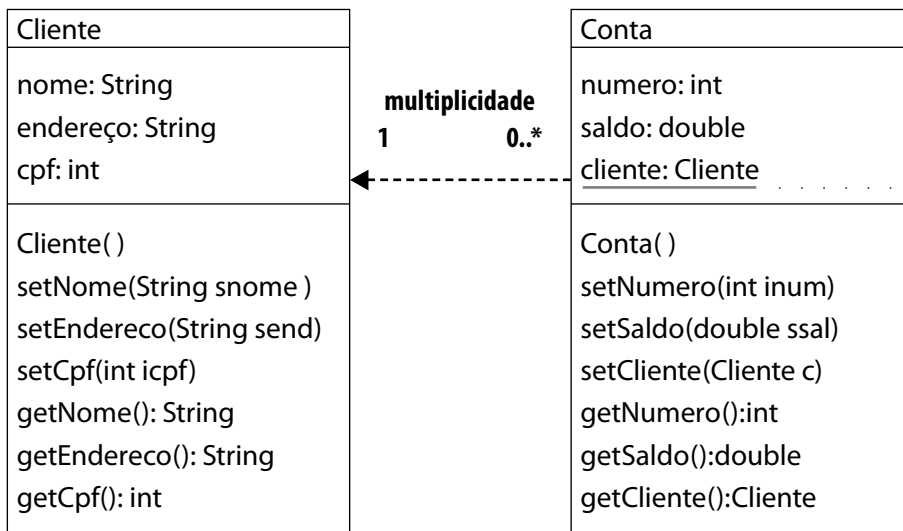


Existe alguma relação entre estes dois tipos de objetos: Cliente e Conta?

Sim! Um cliente poder possuir várias contas e, (no nosso sistema) uma conta, pode pertencer a, no máximo, 1 cliente. Isso indica a multiplicidade desse relacionamento.

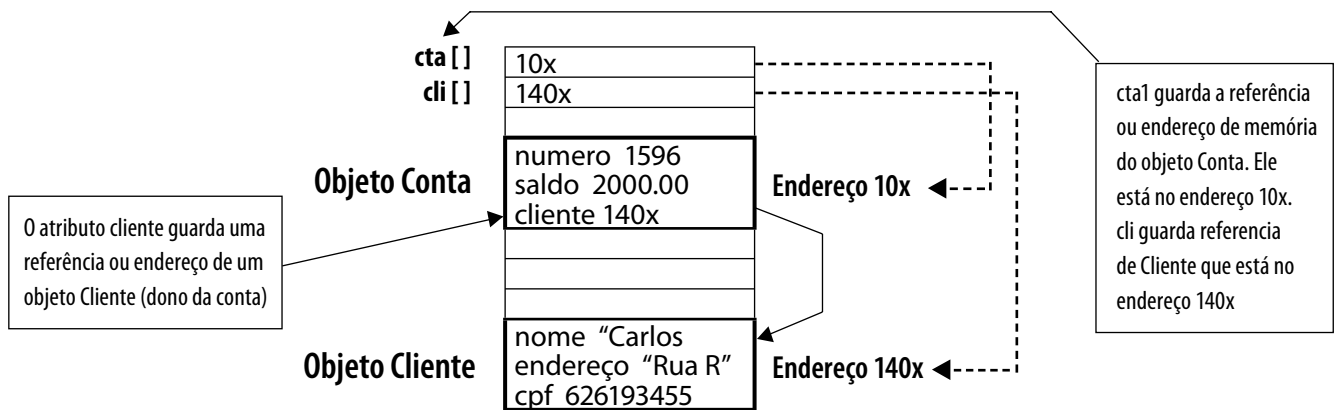
O objeto conta, apenas conhece o seu cliente. Esse é um relacionamento de associação.

A figura a seguir representa em UML, as classes desse sistema e como elas se relacionam.



Observe que o atributo cliente é do tipo Cliente. Isso caracteriza a ligação do objeto conta com o objeto Cliente. Cada conta faz referência a um determinado Cliente.

Antes de demonstrar como ficará o código das classes Conta e Cliente, vamos representar graficamente um objeto c1 do tipo Conta em memória, fazendo referência através do seu atributo cliente a um objeto Cliente.



Primeiramente, vamos implementar a classe Cliente.

Posteriormente, vamos implementar a classe Conta com o atributo cliente que é do tipo Cliente.



Lembre-se!

Quando implementamos uma classe que possui algum atributo ou faz referência a outra classe, essa outra classe, já deve ter sido implementada, por isso, vamos implementar primeiro a classe Cliente.

Implementação da Classe Cliente.java

```
Linha 1      public class Cliente{
Linha 2          private String nome;
Linha 3          private String endereco;
Linha 4          private int cpf;
Linha 5
Linha 6          public Cliente( ){
Linha 7              nome=" ";
Linha 8              endereco="";
Linha 9              cpf=0;
Linha 10         }
Linha 11
Linha 12         public Cliente(String snom, String send, int icpf){
Linha 13             nome=snom;
Linha 14             endereco=send;
Linha 15             cpf=icpf;
Linha 16         }
Linha 17
Linha 18         public void setNome(String snom){
Linha 19             nome=snom;
Linha 20         }
Linha 21
Linha 22         public void setEndereco(String sender){
Linha 23             endereco= sender;
Linha 24         }
Linha 25
Linha 26         public void setCpf(int icpf){
Linha 27             cpf= icpf;
Linha 28         }
Linha 29
Linha 30         public String getNome(){
Linha 31             return nome;
Linha 32         }
Linha 33
Linha 34         public String getEndereco(){
Linha 35             return endereco;
Linha 36         }
Linha 37
Linha 38     }
Linha 39         public int getCpf( ){
Linha 40             return cpf;
Linha 41         }
Linha 42 } //fim da classe Cliente
```


Implementação da Classe Conta.java

```

Linha 1  public class Conta{
Linha 2      private int numero;
Linha 3      private double saldo;
Linha 4      private Cliente cliente;
Linha 5
Linha 6      public Conta ( ){
Linha 7          numero =0;
Linha 8          saldo =0;
Linha 9          cliente = null;
Linha 10
Linha 11      }
Linha 12      public void setNumero(int inum){
Linha 13          numero=inum;
Linha 14      }
Linha 15
Linha 16      public void setSaldo(double ssaldo){
Linha 17          saldo=ssaldo;
Linha 18      }
Linha 19
Linha 20      public void setCliente(Cliente c){
Linha 21          cliente = c;
Linha 22      }
Linha 23      public int getNumero( ){
Linha 24          return numero;
Linha 25      }
Linha 26
Linha 27      public double getSaldo(){
Linha 28          return saldo;
Linha 29      }
Linha 30
Linha 31      public Cliente getCliente(){
Linha 32          return cliente;
Linha 33      }
Linha 34  }
```

O atributo cliente é do tipo Cliente, ou seja, fará referência a um objeto do tipo Cliente em memória.

O atributo cliente é inicializado com o valor null. Todo atributo/objeto deve ser inicializado com esse valor. Isso quer dizer que, no momento, ele não faz referência a nenhum objeto.

Método setCliente(Cliente c) recebe como parâmetro a referência ou endereço de um objeto Cliente. Essa referência é armazenada no atributo cliente. Isso indicará que o objeto Conta está ligado a um objeto Cliente.

Esse método retorna a referência para um objeto Cliente armazenada no atributo cliente. Por isso o tipo de retorno é Cliente.

Depois de implementar a classe Cliente e a classe Conta, vamos desenvolver um sistema de contas correntes que cadastre Clientes e suas Contas.

Como o sistema tem fins didáticos, cadastrar 3 contas e 2 clientes. Você deve expandir esse número depois, nas atividades de auto-avaliação.

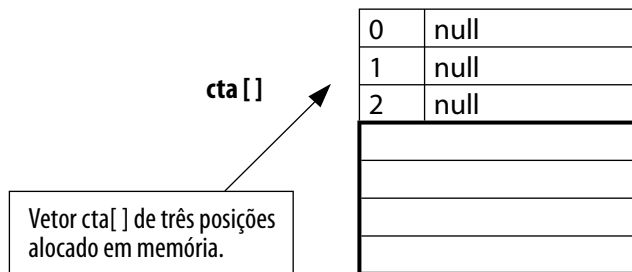
Implementação da Classe Cliente.java

```

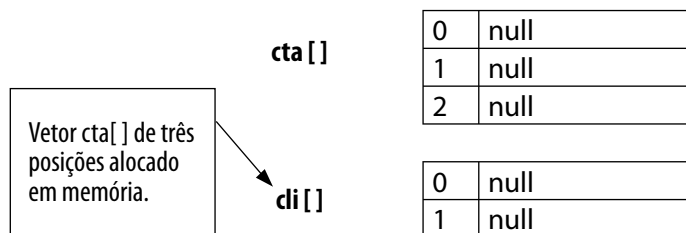
Linha 0 import javax.swing.*;
Linha 1 public class SistemaContaCorrente{
Linha 2     public static void main(String args[]) {
Linha 3         Conta cta[ ]=new Conta[3];
Linha 4         Cliente cli[ ]= new Cliente[2];
Linha 5         //Cadastrar o primeiro cliente com uma conta corrente
Linha 6         cli[0] = new Cliente( );
Linha 7         cli[0].setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 8         cli[0].setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 9         cli[0].setCpf(Integer.parseInt(JOptionPane.showInputDialog("Entre com o CPF")));
Linha 10        cta[0] = new Conta();
Linha 11        cta[0].setNumero(Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero")));
Linha 12        cta[0].setSaldo(Double.parseDouble(JOptionPane.showInputDialog("Entre com o saldo")));
Linha 13        cta[0].setCliente(cli[0]);
Linha 14        //cadastrar o segundo cliente e sua conta
Linha 15        cli[1] = new Cliente( );
Linha 16        cli[1].setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 17        cli[1].setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 18        cli[1].setCpf(Integer.parseInt(JOptionPane.showInputDialog("Entre com o CPF")));
Linha 19        cta[1] = new Conta();
Linha 20        cta[1].setNumero(Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero")));
Linha 21        cta[1].setSaldo(Double.parseDouble(JOptionPane.showInputDialog("Entre com o saldo")));
Linha 22        cta[1].setCliente(cli[1]);
Linha 23        //cadastrar outra conta e associar ao primeiro cliente.
Linha 24        cta[2] = new Conta();
Linha 25        cta[2].setNumero(Integer.parseInt(JOptionPane.showInputDialog("Entre com o Numero")));
Linha 26        cta[2].setSaldo(Double.parseDouble(JOptionPane.showInputDialog("Entre com o saldo")));
Linha 27        cta[2].setCliente(cli[0]);
Linha 28        //listar o número da conta, saldo e nome de todos os clientes cadastrados
Linha 29        JOptionPane.showMessageDialog(null," Número da conta – Saldo - Nome dos Clientes \n" +
Linha 30        cta[0].getNumero( ) + " - " + cta[0].getSaldo( ) + " - " + cta[0].getCliente( ).getNome( ) + "\n" +
Linha 31        cta[1].getNumero( ) + " - " + cta[1].getSaldo( ) + " - " + cta[1].getCliente( ).getNome( ) + "\n" +
Linha 32        cta[2].getNumero( ) + " - " + cta[2].getSaldo( ) + " - " + cta[2].getCliente( ).getNome( ) );
Linha 36
Linha 37        System.exit(0);
Linha 38    }
Linha 39 }
```

Vamos analisar o código anterior, linha a linha:

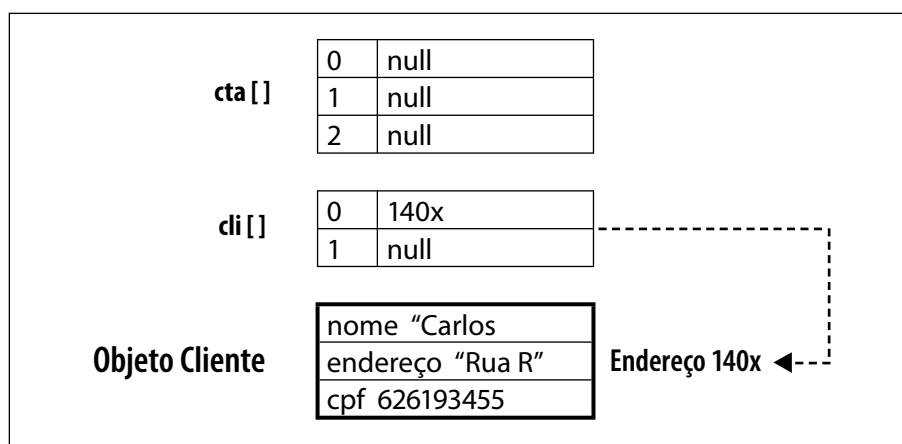
Linha 3: Criação de um vetor chamado `cta[]` para armazenar objetos do tipo `Conta`. Após a criação do vetor, todas as posições estão com o valor `null`.



Linha 4: Criação de um vetor chamado `cli[]` de 2 posições para armazenar objetos do tipo `Cliente`. Após a criação do vetor todas as posições estão com o valor `null`.

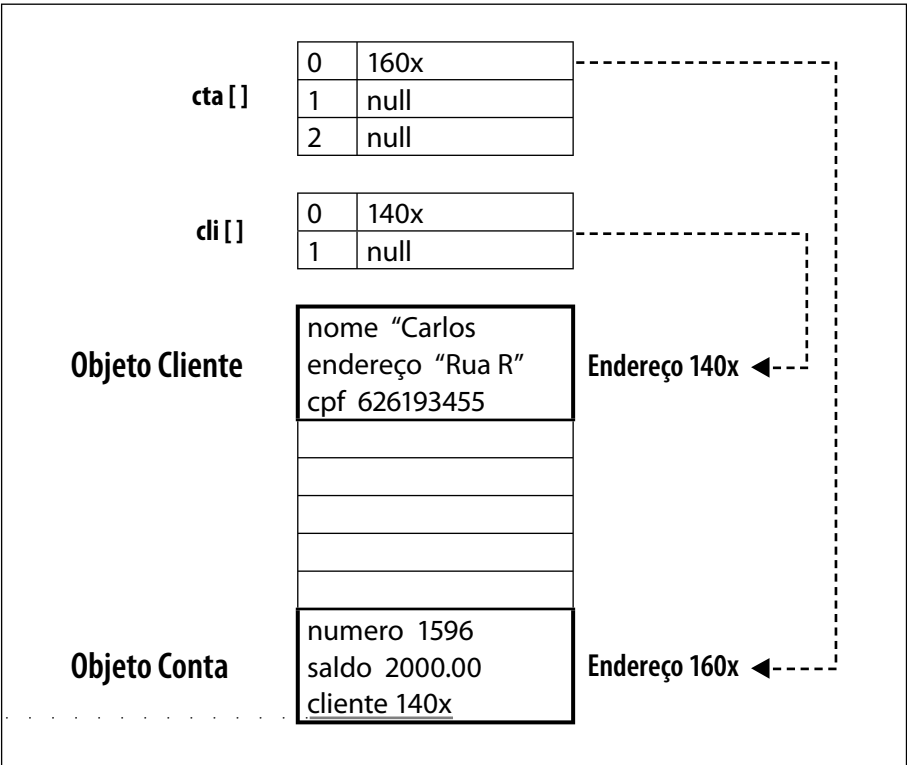


Linhas 6 a 9: Criação de um objeto `Cliente` e atribuição da sua referência na posição 0 do vetor `cli`. Posteriormente, armazenamento de nome, endereço e cpf nos respectivos atributos desse objeto.



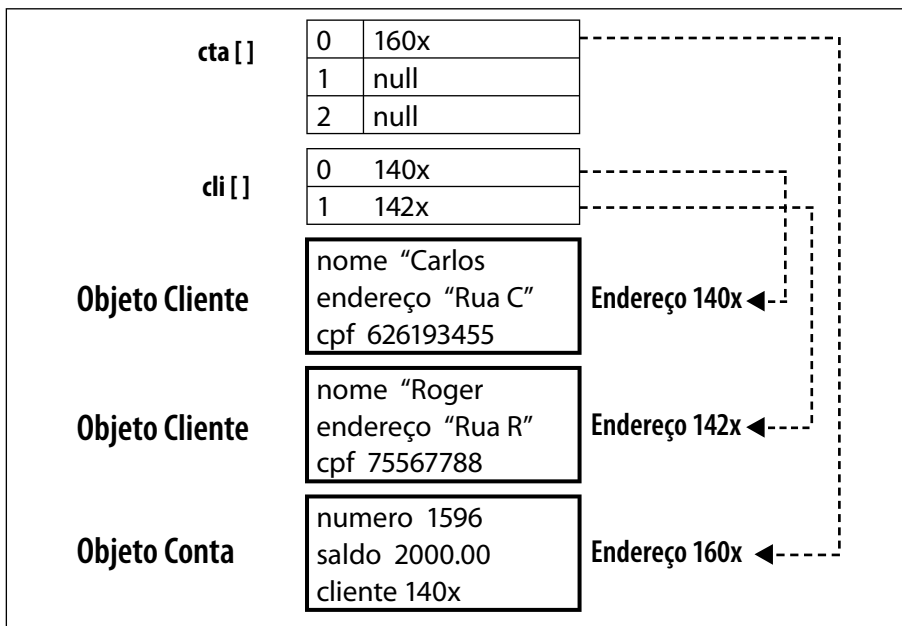
Linhas 10, 11 e 12: Criação de um objeto Conta e atribuição da sua referência na posição 0 do vetor cta. Posteriormente, armazenamento de número e saldo desse objeto.

Linha 13: passagem da referência que está armazenada na posição 0 do vetor cli[] para o método setCliente() do objeto Conta. A referência que está nessa posição corresponde ao endereço do primeiro objeto Cliente cadastrado. No método setCliente, essa referência é armazenada no atributo cliente do objeto Conta que está sendo cadastrado no momento (cta[0]).

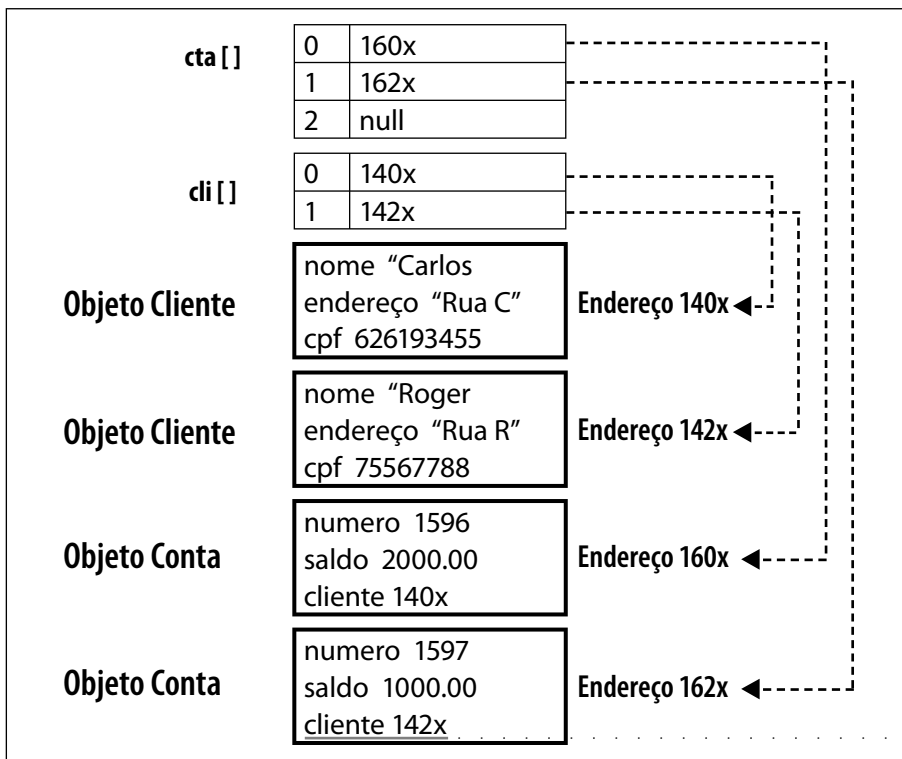


O atributo cliente do objeto Conta aponta ou faz referência para o endereço de um objeto Cliente. 140x é o endereço do primeiro objeto Cliente cadastrado.

Linhas 15 a 18: Criação do segundo objeto Cliente e atribuição da sua referência na posição 1 do vetor cli. Posteriormente, armazenamento de nome, endereço e cpf nos respectivos atributos desse objeto.



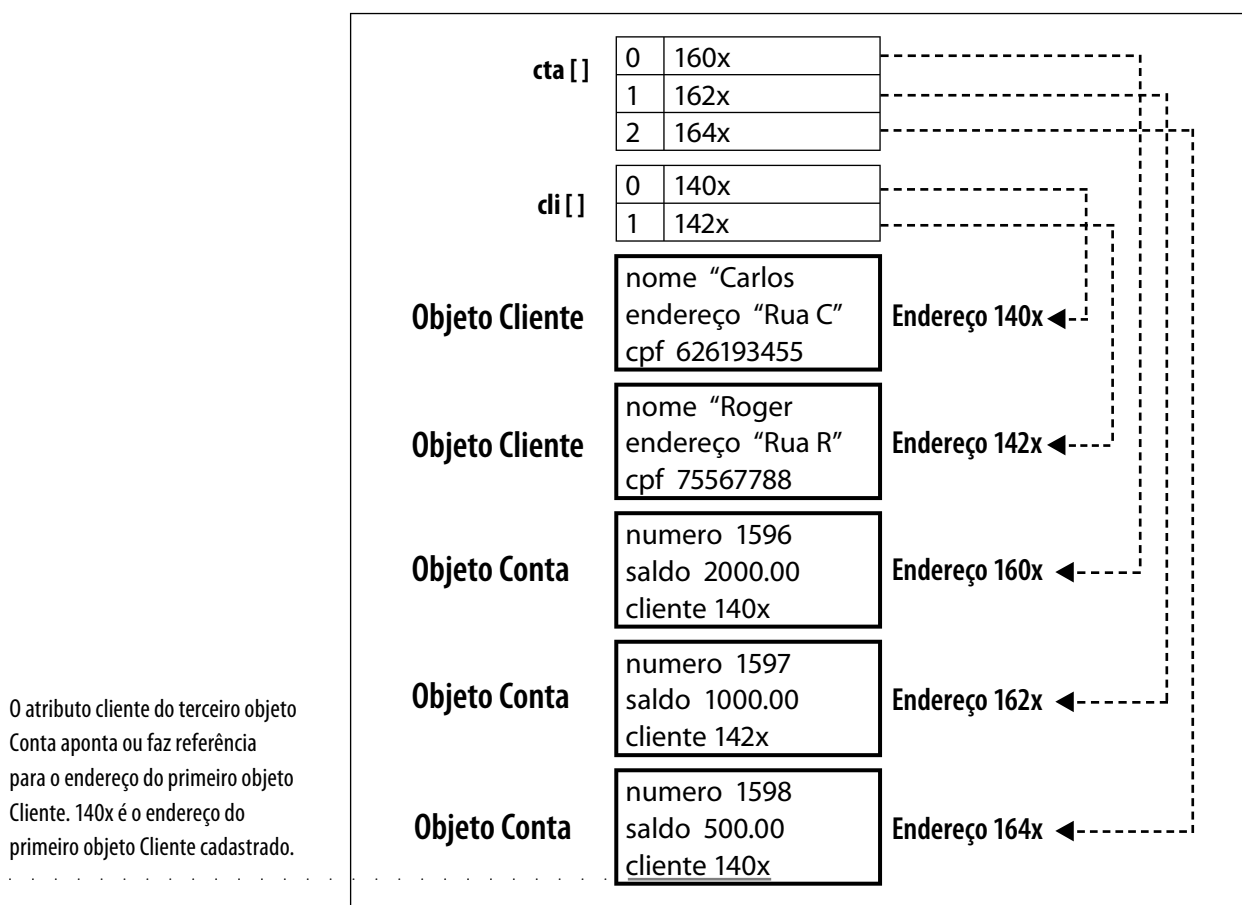
Linhas 19, 20, 21 e 22: Criação do segundo objeto Conta e atribuição da sua referência na posição 1 do vetor cta. Posteriormente, será armazenado número, saldo desse objeto e a referência de um objeto Cliente no atributo cliente desse objeto Conta.



O atributo cliente do segundo objeto Conta aponta ou faz referência para o endereço do segundo objeto Cliente. 142x é o endereço do segundo objeto Cliente cadastrado.

Linhas 24, 25, 26 e 27: agora será cadastrada outra conta, ou seja, será criado um objeto conta e ele será associado ao primeiro cliente cadastro. Isso é possível porque na nossa modelagem inicial do problema, um cliente pode ter várias contas associadas a ele. A referência desse terceiro objeto Conta será armazenada na terceira posição do vetor `cta[]`.

Posteriormente, será armazenado número, saldo desse objeto, conta e a referência do primeiro objeto Cliente no atributo cliente desse objeto Conta. Portanto, o primeiro cliente possui duas contas correntes.



Linhas 29 a 32: Impressão na tela do número da conta, saldo e nome dos clientes cadastrados. Para isso, é chamado o método `getNumero()` e `getSaldo()` de cada objeto Conta (dos três). Para imprimir o nome do cliente de uma conta, é chamado o método `getCliente()`. Esse método irá retorna a referência (endereço) do objeto Cliente associado àquela conta. A partir dessa referência é possível recuperar o nome do cliente através do seu método `getNome()`.

Vamos exemplificar com a linha 30.



O que existe no atributo **número** do objeto cuja referência está em `cta[0]`?

O que está em `cta[0]` é uma referência (endereço) de um objeto. Nesse endereço existe um objeto *Conta* cujo valor do atributo `numero` é 1596. Logo é 1596 que será impresso na tela.

```
Linha 30  cta[0].getNumero() + " " + cta[0].getSaldo() + "\n" + cta[0].getCliente().getNome() + "\n" +;
```



O que existe no atributo **cliente** do objeto cuja referência está em `cta[0]`?

Existe uma referência para um objeto do tipo *Conta*. O método `getCliente()` do objeto *Conta* irá retornar essa referência para um objeto *Cliente*. No caso dessa linha, será retornada a referência de número 140x para um objeto *Cliente*.

Lembre-se de que a instrução “\n” serve para imprimir o próximo valor na próxima linha.

```
Linha 30  cta[0].getNumero() + " " + cta[0].getSaldo() + "\n" + cta[0].getCliente().getNome() + "\n" +;
```

A partir dessa referência (ela aponta para um objeto *Cliente*), podemos chamar um método `getNome()` desse objeto *Cliente*. Nesse caso o nome retornado será “Carlos”. Carlos é o cliente desse objeto *conta* que é referenciado em `cta[0]`.

A saída em tela será como a listagem abaixo:

Número da conta	Saldo	Nome dos Clientes
1596	2000.00	Carlos
1597	1000.00	Roger
1598	500.00	Carlos

A primeira versão do sistema de contas correntes foi mais simples para facilitar o seu entendimento. Vamos elaborar uma nova versão desse mesmo sistema um pouco mais refinada. Novamente, o nosso sistema terá a limitação de possuir três contas e dois clientes. Vamos trabalhar assim para diminuir a complexidade do código.

```

Linha 0 import javax.swing.*;
Linha 1 public class SistemaContaCorrente{
Linha 2     public static void main(String args[]) {
Linha 3         Conta cta[ ]=new Conta[3];
Linha 4         Cliente cli[ ]= new Cliente[2];
Linha 5         int pcli=0, pta=0;
Linha 6         while (true) {
Linha 7             String op= JOptionPane.showInputDialog
("1 – Cadastra Cliente e Conta 2 – Cadastra Conta para Cliente 3 - Sair");
Linha 8             if (op.equalsIgnoreCase("1")){
Linha 9                 cli[pcli] = new Cliente( );
Linha 10                 cli[pcli]. setNome(JOptionPane.showInputDialog("Entre com o Nome"));
Linha 11                 cli[pcli].setEndereco(JOptionPane.showInputDialog("Entre com o Endereço"));
Linha 12                 cli[pcli].setCpf(Integer.parseInt(JOptionPane.showInputDialog("Entre com o CPF")));
Linha 13                 cta[pcta] = new Conta();
Linha 14                 cta[0].setNumero(Integer.parseInt(JOptionPane.showInputDialog
("Entre com o Numero")));
Linha 15                 cta[0].setSaldo(Double.parseDouble(JOptionPane.showInputDialog
("Entre com o saldo")));
Linha 16                 cta[0].setCliente(cli[0]);
Linha 17                 pcli ++ ; //igual a pcli = pcli + 1;
Linha 18                 pcta ++ ;
Linha 19             }
Linha 20             else
Linha 21                 if (op.equalsIgnoreCase("2")){
Linha 22                     String nomecli = JOptionPane.showInputDialog("Entre com o Nome do Cliente");
Linha 23                     for (int i=0;i<pcli;i++) {
Linha 24                         if (nomecli.equalsIgnoreCase(cli[i].getNome( ))) { //achou o cliente
Linha 25                             JOptionPane.showMessageDialog(null, "Cliente : "+ cli[i].getNome( ));
Linha 26                             cta[pta] = new Conta();
Linha 27                             cta[pta]. setNumero(Integer.parseInt(JOptionPane.showInputDialog
("Entre com o Numero")));
Linha 28                             cta[pta].setSaldo(Double.parseDouble(JOptionPane.showInputDialog
("Entre com o saldo")));
Linha 29                             cta[1].setCliente(cli[i]);
Linha 30                             pcta ++ ; //incrementa o contador de contas

```



```

Linha 31             break; //instrução de interrompe a repetição (for)
Linha 32             }
Linha 33             }
Linha 34             } //fim da op == 2
Linha 35             else
Linha 36                 if (op.equalsIgnoreCase("3"))
Linha 37                     break; //sai do while
Linha 38             } //fim do while
Linha 39             System.exit(0);
Linha 40             } //fim do método main( )
Linha 41             } //fim da classe
Linha 42
```

No código acima, é mostrada uma tela ao usuário para que ele escolha entre três opções:

- Se ele escolher a opção 1, é cadastrado um cliente e sua conta.
- Se ele escolher a opção 2, é cadastrada uma conta para um cliente já existente, por isso, o usuário deve entrar com o nome do cliente. Esse cliente deve ser pesquisado pelo nome digitado no vetor de cliente. Se encontrar, pode ser cadastrada uma nova conta e associada a referência desse cliente a essa conta.
- Se ele escolher a opção 3, significa que ele quer sair do sistema. A instrução `break` interrompe a repetição `while`, e o programa termina.



Síntese

Nesta unidade, você fortaleceu seu conhecimento sobre o relacionamento de associação entre classes.

Para isso, retomamos o exemplo de sistema que utilizamos nas primeiras unidades em que trabalhamos conceitos de OO. Foram implementadas as classes Cliente e Conta (conta corrente) e, posteriormente, foram desenvolvidos duas versões do sistema de cadastro de contas. A primeira versão foi mais simplificada e, a segunda, um pouco mais elaborada.

Espero que você tenha entendido o conceito de associação!



Atividades de auto-avaliação

- 1) Implemente o código dessa unidade.
- 2) Desenvolva um sistema para armazenar informações sobre professores e disciplinas de uma instituição educacional.

Requisitos do sistema:

- Sabe-se que cada professor pode ministrar várias disciplinas.
- Uma disciplina só é ministrada por um professor.
- É importante armazenar as seguintes informações sobre cada professor: nome, titulação máxima e carga horária.
- Sobre cada disciplina é necessário armazenar nome e carga horária.
- O sistema deve permitir através de um menu, que o usuário:

Faça o seguinte:

- a) cadastre as disciplinas e o professor que ministra a disciplina;
- b) entre com o nome de uma disciplina e o sistema mostre o nome do professor que ministra essa disciplina;
- c) entre com o nome de uma titulação e o sistema mostre o nome de todos os professores que possuem essa titulação.

Obs.: Antes da implementação, crie o diagrama de classes do sistema.

UNIDADE 13

13

Herança



Objetivos de aprendizagem

- Aprender o conceito de herança.
- Identificar a necessidade de sobrescrever um método.



Seção de estudo

Seção 1 Herança



Para início de conversa

Nas unidades anteriores, você aprendeu que duas classes que representam atributos e comportamentos de dois tipos de objetos podem se relacionar.

Nesta unidade, você irá aprender outro tipo de relacionamento entre classes, o relacionamento de herança.

Aprenderá o que é o conceito de herança e outros conceitos relacionados como o modificador `protected` e métodos sobrescritos.

SEÇÃO 1 - Herança

A Herança é um recurso da orientação a objetos que permite que **atributos e comportamentos (métodos) COMUNS** a diversos tipos de objetos, existentes em um problema, sejam agrupados e representados em uma única classe base, conhecida como **SUPERCLASSE**.

Os **atributos e comportamentos (métodos) ESPECÍFICOS** de cada tipo de objeto presente no problema são representados por classes específicas desses tipos de objetos. Essas classes específicas, conhecidas como **SUBCLASSES**, **herdam** os atributos e comportamentos comuns que foram agrupados na superclasse.

A partir de uma classe base (superclasse), outras classes podem ser **especificadas** ou **especializadas**.

- Uma subclasse é uma especialização de uma superclasse.
- Uma superclasse é uma generalização de uma subclasse.

Cada subclasse **herda** atributos e comportamentos da superclasse e acrescenta mais atributos e comportamentos específicos a essa subclasse.

O exemplo, a seguir, ilustra quando é necessário utilizar o recurso de herança ao modelar um problema dentro do paradigma orientado a objeto.



Uma empresa do ramo educacional necessita armazenar informações sobre seus funcionários. Nessa empresa, existem duas categorias de funcionários. Os que trabalham com funções técnico-administrativas, como secretárias, bibliotecárias, laboratoristas e os que trabalham com atividades de docência, como os professores. Os primeiros são chamados de 'Administrativo' e, os segundos, de 'Docente'. Para ambos os tipos de funcionários, necessita armazenar nome, endereço e salário.

Para os funcionários do tipo 'Administrativo', é necessário armazenar também cargo e setor. Para os funcionários do tipo 'Docente', é necessário armazenar número de horas aula semanal e a titulação do professor."

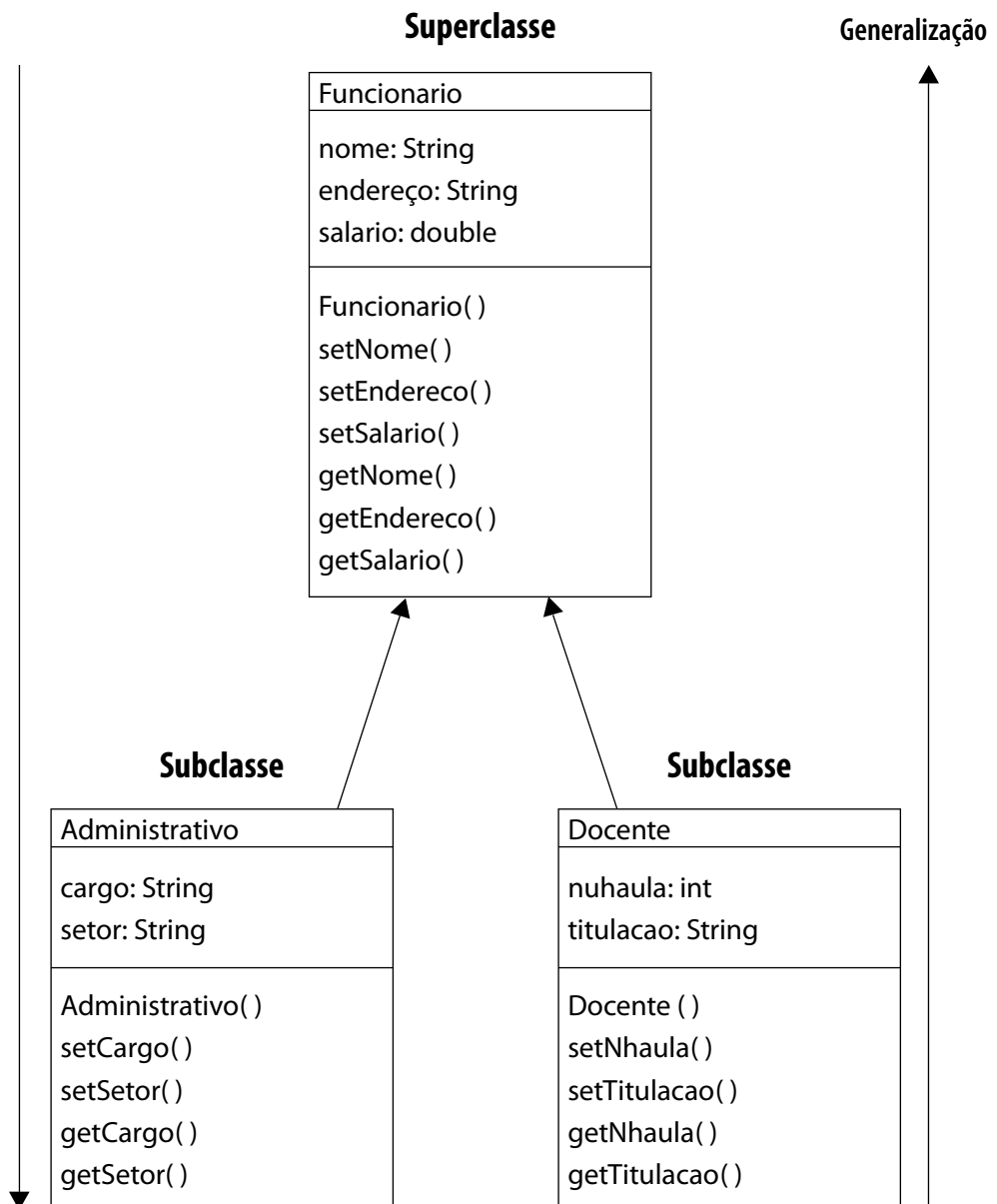
A princípio, poderíamos pensar que existem dois tipos de objetos nesse problema: Administrativo e Docente. Então, vamos identificar os atributos e comportamentos para esses dois tipos de objetos:

Administrativo	Docente
Nome	Nome
Endereço	Endereço
Salário	Salário
Cargo	Nu. Horas aula semanais
Setor	Titulação do docente (doutor, mestre, espec.)

Note que, tanto os objetos do tipo Administrativo', quanto os objetos do tipo Docente possuem três atributos (conseqüentemente, métodos get e set para esses atributos) em comum: nome, endereço e salário.

Em vez modelar e implementar esses atributos em comum nas duas classes (RETRABALHO), podemos modelar esses atributos em uma classe genérica chamada Funcionário, pois, tanto Administrativo quanto Docente, são tipos de Funcionário.

Sendo assim, a solução modelada em UML para esse problema é a seguinte:



Observe que os atributos e comportamentos em comum, foram modelados na classe Funcionário, e os atributos e comportamentos específicos, estão nas classes especializadas, Administrativo e Docente.

Como Administrativo e Docente são tipos de Funcionário, eles também possuem nome, endereço e salário, mas esses atributos não precisam ser modelados e implementados nessas

duas classes. Eles podem ser modelados e implementados na superclasse Funcionário e serem **HERDADOS** pelas subclasses Administrativo e Docente.



Logo, **Herança** é um tipo de relacionamento que pode existir entre classes em que uma subclasse herda atributo e comportamento de uma superclasse.

Pode-se empregar o relacionamento de Herança entre classes quando uma classe for “*um tipo de*” outra classe. Costuma-se interpretar o relacionamento de herança entre duas classes como: “*é um tipo de*” ou “*é um*”.



Docente “*é um tipo de*” Empregado.
Administrativo “*é um*” Empregado.

Em UML, o símbolo que expressa o relacionamento de herança é a seta no sentido da subclasse para a superclasse, como pôde ser visto na figura mais acima.

A maior característica desse tipo de relacionamento é, como o próprio nome diz, a *herança de métodos e atributos da superclasse* pela subclasse.

Quando uma subclasse herda atributos e comportamentos, é como se esses atributos e comportamentos fossem dela própria, ou seja, é como se eles tivessem sido implementados na própria subclasse. Mas, na verdade, eles foram implementados somente em uma classe, a superclasse e podem ser herdados por várias subclasses, evitando assim, a programação desses atributos e comportamentos em cada uma das classes especializadas.



Quando uma subclasse herda atributos e comportamentos de uma superclasse, pode-se dizer que essa subclasse **estende** a superclasse.

O sentido de estender é de aumentar, ou seja, uma subclasse terá todos os atributos e comportamentos da sua superclasse e mais seus atributos e comportamentos específicos.



Docente **estende** a superclasse Funcionario.

Administrativo **estende** a superclasse Funcionario.

Objetivo da herança

O objetivo principal da herança é a **reutilização de código**, já que novas classes (subclasses) podem ser criadas a partir de outra já existente (superclasse), herdando seus atributos e métodos.

A classe que herda ou estende outra classe, tem a capacidade de ter novos atributos e métodos de acordo com sua característica e, também pode modificar ou **sobrescrever** (*override*) os métodos herdados de acordo com sua necessidade.

A reutilização de código economiza tempo de desenvolvimento de programas.

Veremos o que significa sobrescrever um método herdado mais a frente.

Outras características

Podem existir vários níveis de relacionamento de herança entre classes, por exemplo, uma classe Docente herda de Funcionário que poderia herdar de Pessoa.



Chama-se **superclasse direta** de uma subclasse, aquela imediatamente superior a essa subclasse e, **superclasse indireta**, aquela de dois ou mais níveis acima da hierarquia.

Partindo do exemplo é possível verificar que, Funcionário é superclasse direta de Docente e Pessoa seria superclasse indireta de Docente.

Ao se instanciar um objeto de qualquer subclasse, pode-se acessar diretamente os membros com qualificador de acesso public da superclasse (normalmente métodos) como se fossem parte da subclasse.

Isso só é possível por causa do relacionamento de herança que existe entre a superclasse e a subclasse.

Veremos na prática como isso funciona.

A implementação da classe Funcionário continua a mesma.

Vamos implementar as subclasses Administrativo e Docente.

```

Linha 1      public class Administrativo extends Funcionario{
Linha 2          private String cargo, setor;
Linha 3
Linha 4          public Administrativo ( ){
Linha 5              super();
Linha 6              cargo = "";
Linha 7              setor = "";
Linha 8          }
Linha 9
Linha 10         public void setCargo(String scargo){
Linha 11             cargo=scargo;
Linha 12         }

                public String getCargo(){
                    return cargo;
                }

                public void setSetor(String ssetor){
                    setor=ssetor;
                }

                public String getSetor(){
                    return setor;
                }
            }

```

A palavra-chave **extends** indica que a subclasse Administrativo estende a superclasse Funcionário.

Essa instrução `super()` chama explicitamente o método construtor da superclasse. Sempre deve estar programada na primeira linha. Quando o método construtor de Administrativo for chamado (quando se cria um objeto do tipo Administrativo) a primeira instrução é chamar o método construtor da superclasse Funcionário.

```

Linha 1      public class Docente extends Funcionario{
Linha 2          private int nha;
Linha 3          private String titulacao;
Linha 4
Linha 5          public Docente(){
Linha 6              super(); //chamada explícita ao construtor
da superclasse
Linha 7              nha=0;
Linha 8              titulacao="";
Linha 9          }
Linha 10

```

```
Linha 11      public void setNha(int inha){  
Linha 12      nha=inha;  
Linha 13      }  
  
              public int getNha(){  
                return nha;  
              }  
  
              public void setTitulacao(String stitu){  
                titulacao=stitu;  
              }  
  
              public String getTitulacao(){  
                return titulacao;  
              }  
  
      }
```

Agora que modelamos e implementamos as classes que representam os dois tipos de objetos do nosso problema, podemos implementar o sistema que armazena informações para um funcionário Administrativo e Docente (problema inicial).

Inicialmente, vamos implementar um sistema que armazena informações para um funcionário docente e, para um funcionário administrativo.

```

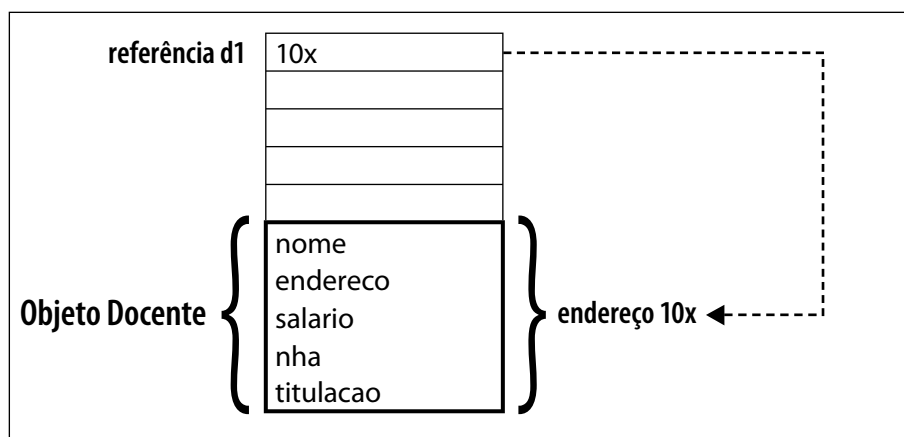
Linha 1  import javax.swing.JOptionPane;
Linha 2  public class CadastraAdmDocente{
Linha 3      public static void main(String args[]){
Linha 4          Docente d1= new Docente( );
Linha 5          d1.setNome(JOptionPane.showInputDialog("Entre com o Nome:"));
Linha 6          d1.setEndereco(JOptionPane.showInputDialog("Entre com o Endereco:"));
Linha 7          d1.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salario:")));
Linha 8          d1.setNha(Integer.parseInt(JOptionPane.showInputDialog("Entre com o Nu.de Horas Aula
Semanais:")));
Linha 9          d1.setTitulacao(JOptionPane.showInputDialog("Entre com a Titulacao:"));
Linha 10
Linha 11      Administrativo a1= new Administrativo( );
Linha 12      a1.setNome(JOptionPane.showInputDialog("Entre com o Nome:"));
Linha 13      a1.setEndereco(JOptionPane.showInputDialog("Entre com o Endereco:"));
Linha 14      a1.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre com o Salario:")));
Linha 15      a1.setCargo(JOptionPane.showInputDialog("Entre com o Cargo:"));
Linha 16      a1.setSetor(JOptionPane.showInputDialog("Entre com o Setor:"));
Linha 17      System.exit(0);
Linha 18  }
Linha 19  }
    
```

Vamos analisar o código linha a linha:

Linha 4: Para armazenar informações sobre um funcionário docente, precisamos criar um objeto do tipo Docente.

Quando se cria um objeto de uma subclasse, é alocado espaço em memória para todos os atributos desse objeto e para os atributos que ele herda da superclasse.

A figura abaixo ilustra a representação do objeto Docente em memória.



Observe que quando o objeto `Docente` é criado, é alocada memória para os seus atributos (`nha` e `titulação`) e para os atributos herdados de `Funcionário` (`nome`, `endereço` e `salário`). É como se esses atributos tivessem sido implementados dentro da classe `Docente`. Mas não foram. Estão sendo herdados de `Funcionário`.

Linha 5: chamado ao método `setNome()` e passagem do nome digitado pelo usuário como parâmetro para ser armazenado no atributo `nome` do objeto `d1`.

Note que esse método está sendo chamado precedido de um objeto do tipo `Docente` (`d1`), mas o método não está implementado na classe `Docente`. Ele está implementado na classe `Funcionário`, mas como todo objeto `Docente` herda atributos e métodos da classe `Funcionário`, é como se estivesse implementado em `Docente`.

Qualquer objeto `Docente` pode chamar, invocar métodos públicos da classe `Funcionário` por causa da herança.

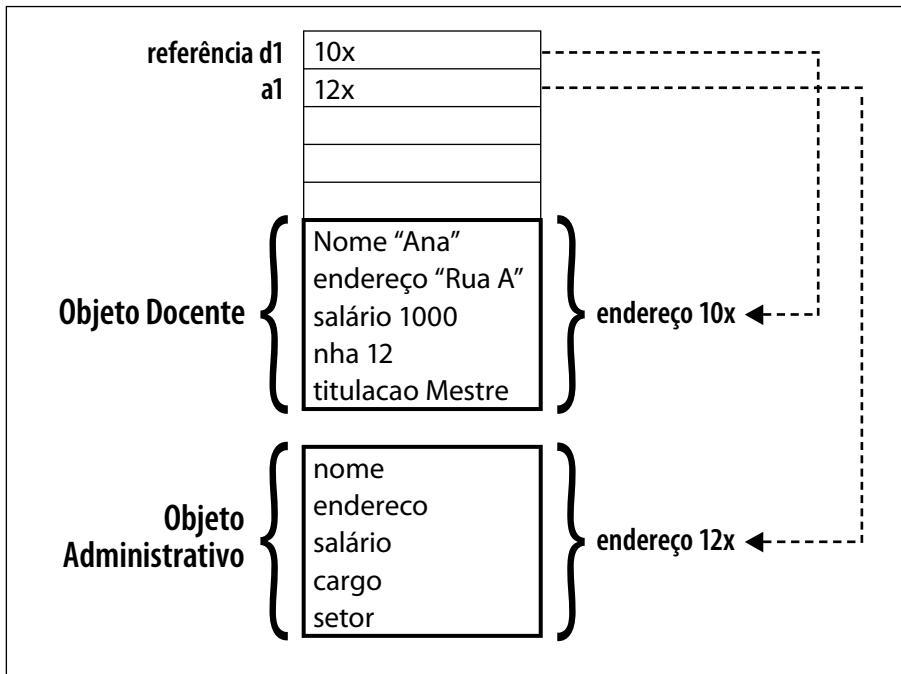
Linhas 6 e 7: o mesmo acontece nessas linhas.

Linha 8: chamada ao método `setNha()` e passagem do número de horas aula semanal do docente como parâmetro para ser armazenado no atributo `nha` do objeto `d1`.

Linha 9: chamada ao método `setTitulacao()` e passagem da `titulação` do docente como parâmetro para ser armazenado no atributo `titulacao` do objeto `d1`.

Linha 10: criação do objeto `Administrativo` para armazenar informações sobre um funcionário administrativo.

A figura abaixo ilustra a representação do objeto Docente (já preenchido) e o objeto Administrativo em memória.



Linha 12: chamado ao método `setNome()` e passagem do nome digitado pelo usuário como parâmetro para ser armazenado no atributo `nome` do objeto `a1`.

Note que esse método está sendo chamado precedido de um objeto do tipo Administrativo (`a1`), mas o método não está implementado na classe Administrativo. Ele está implementado na classe Funcionário, porém como todo objeto Administrativo, herda atributos e métodos da classe Funcionário, é como se estivesse implementado em Administrativo.

Qualquer objeto Administrativo pode chamar, invocar métodos públicos da classe Funcionário por causa da herança.

Método Construtor em Subclasse

Na implementação do método construtor de uma subclasse o construtor da superclasse deve ser explicitamente chamado na primeira linha através da instrução `super()`.

Veja isso na linha 5 da implementação da subclasse Administrativo e, na linha 6, da implementação da subclasse Docente.

A execução da instrução `super()` fará com que o método construtor sem parâmetros da superclasse seja invocado. Por que isso deve ser feito?

No nosso caso, para que se possa fazer a inicialização dos atributos de um objeto herdados da superclasse.

Vamos ver um exemplo prático:

Na linha 4 da classe `CadastraAdmDocente`, estamos criando um objeto do tipo `Docente`. Sempre que criamos ou alocamos um objeto em memória, o método construtor desse objeto é automaticamente chamado. Nesse caso, o método construtor `Docente()` está sendo chamado.



O que está programado dentro desse método construtor?

A inicialização dos atributos desse objeto que acabou de ser criado em memória. Nesse caso, dos atributos `nha` e `titulação` que são atributos de um objeto `Docente`.

Mas um objeto `Docente` não tem alocado em memória somente esses dois atributos. Ele alocou espaço para os atributos herdados de `Funcionário` que são: `nome`, `endereço` e `salário`.



Quem irá inicializar esses atributos?

O método construtor da superclasse `Funcionário`. É nele que esses atributos são inicializados.

Para isso, devemos chamar explicitamente esse método construtor da superclasse na primeira linha do método construtor da subclasse.

Modificador de acesso protected

Você deve lembrar que na unidade anterior, falamos sobre os modificadores de acesso public, private, protected e acesso de pacote.

Os membros (atributos e métodos) **public** de uma superclasse podem ser acessados diretamente (somente pelo nome) em todas as subclasses que estendem essa superclasse.

Isso significa que o trecho seguinte de código é válido na subclasse Docente.

O método `imprimeNomeTituSalario ()` do exemplo retorna o valor do atributo `nome` juntamente com a `titulação` e o `salário` de determinado objeto `Docente`.

```
public class Docente{
    .
    .
    public String imprimeNomeTituSalario() {
        return getNome( ) + " "+titulação+" "+getSalario( );
    }
    .
    .
}
```

O método público `getNome()` não existe na subclasse `Docente`. Ele está na superclasse `Funcionário`. Mas como a subclasse `Docente` herda de atributos e métodos de `Funcionário`, pode acessar esse método public como se estivesse implementado nela.

Os membros (atributos e métodos) **private** de uma superclasse somente são acessados pelos métodos dessa superclasse, ou seja, a subclasse não pode acessar diretamente os membros **private** da sua superclasse.

O trecho seguinte de código é **INVÁLIDO**, não pode acontecer, porque os atributos `nome` e `salario` são **private** na superclasse `Funcionário`.

O mesmo acontece com o método público `getSal()` que está sendo acesso acessado diretamente pelo nome da subclasse `Docente`.

```

public class Docente{
    .
    .
    public String imprimeNomeTituSalario( ) {
        return nome+ " "+titulação+" "+salario;
    }
    .
    .
}

```

A subclasse somente pode acessar qualquer atributo **private** da superclasse através dos métodos públicos get e set dessa superclasse.

Na unidade de modificadores, dissemos que o modificador de acesso **protected** iria ser explicado juntamente com o conceito de herança.

Agora que você já estudou esse conceito e sabe quando aplicá-lo, vamos aprender onde o modificador **protected** pode ser inserido.

O modificador **protected** pode ser utilizado na definição de atributos e métodos da superclasse.

O trecho de código da superclasse Funcionário ilustra essa explicação.

```

public classe Funcionário {
    private String nome, endereço;
    protected double salário;
    .
    .
}

```

O atributo salario foi definido como **protected**.

Quando atributos e métodos da superclasse são definidos com o modificador **protected**, eles são “liberados” para acesso direto (pelo nome) pelas subclasses que estendem essa superclasse.

A sobrescrita de método é necessária quando a subclasse não deseja herdar um determinado método da maneira como ele foi implementado.

Vamos ilustrar com um exemplo essa situação.

O sistema de cadastro de informações de funcionários Administrativo e Docente ganhará alguns requisitos de funcionamento que são:

- O salário de cada tipo de funcionário é calculado de maneira diferente. Cada tipo de funcionário ganha um valor base mais um adicional, dependendo do seu tipo: Administrativo ou Docente.
- Para os funcionários do tipo Docente, o adicional de salário é calculado da seguinte forma:

$$\text{nu. de horas-aula} * \text{valor da hora-aula}$$

Sendo que o valor da hora aula depende da titulação de cada professor. A tabela a seguir, mostra o valor da hora aula de um docente conforme sua titulação:

Titulação	Valor hora aula
Especialista	R\$ 15,00
Mestre	R\$ 20,00
Doutor	R\$ 30,00

Assim, o cálculo do salário de um docente é:

$$\text{Salário base} + (\text{nu. de horas-aula} * \text{valor da hora aula})$$

Para os funcionários do tipo Administrativo, o adicional de salário é calculado em função do seu cargo.

Para cada cargo, existe um valor fixo a ser adicionado ao salário base. A tabela abaixo mostra os cargos possíveis e o valor recebido em cada cargo:

Cargo	Valor
Técnico administrativo	R\$ 300,00
Bibliotecária	R\$ 400,00
Técnico de laboratório	R\$ 500,00

Assim, o cálculo do salário de um docente é: salário base + (Valor associado ao cargo)

Analisando esses novos requisitos, vemos que o método `setSalario(double dsal)` implementado na classe `Funcionario` e herdado por `Docente` e `Administrativo` não atende a eles, pois esse método só recebe um valor do salário e armazena no atributo salário do objeto.

Logo, esse método está sendo herdado por `Docente` e `Administrativo`, mas não atende às exigências de nenhuma dessas classes já que, a maneira de calcular o salário, não é a maneira exigida por `Docente` e nem a exigida por `Administrativo`.

O que fazer? Sobrescrever o método `setSalario()` que está sendo herdado na classe `Docente` e `Administrativo`.

A seguir segue o trecho código da classe `Docente` que mostra a sobrescrita do método `setSalario()`.

```

Linha 1 public class Docente extends Funcionario{
Linha 2     private int nha;
Linha 3     private String titulacao;
Linha 4
Linha 5     //métodos construtor, set's e get's
Linha 6
Linha 7     public void setSalario(double dsalario){
Linha 8         if (titulacao.equalsIgnoreCase("Especialista"))
Linha 9             super.setSalario(dsalario+(nha*15));
Linha 10        else
Linha 11            if (titulação.equalsIgnoreCase("Mestre"))
Linha 12                super.setSalario(dsalario+(nha*20));
Linha 13            else
Linha 14                super.setSalario(dsalario+(nha*20));
Linha 15
Linha 16        }
Linha 17    }

```

Note que a assinatura (cabeçalho) do método `setSalario(double dsal)` está idêntica a implementada na superclasse `Funcionário`, mas a sua implementação (lógica) está totalmente diferente para atender ao novo requisito de cálculo de salário de `Docente`.

Vamos analisar essa lógica.

Na linha 8, começa o teste para verificar se a titulação do objeto `Docente` é igual a `Especialista`. Se for igual, o método `setSalario` da superclasse está sendo chamado novamente, e está sendo passado como parâmetro o novo salário calculado em função das horas-aulas * titulação do professor.

Observe que essa instrução de chamada ao método `setSalario` da superclasse está sendo feita dentro da própria redefinição do método `setSalario` na subclasse.

Chamar o método do mesmo nome dentro de um método do mesmo nome que está sendo redefinido é possível, mas deve-se preceder essa chamada com a palavra `super` seguida do nome do método, como mostra a linha 9.

As demais linhas seguem testando as outras opções de titulação do `Docente`.

Depois de sobrescrever o método na subclasse `Docente` vamos sobrescrever o método `setSalario` na subclasse `Administrativo`.

A seguir segue o trecho de código da classe `Administrativo` que mostra a sobrescrita do método `setSalario()`.

```

Linha 1  public class Administrativo extends Funcionario{
Linha 2      private String cargo, setor;
Linha 3
Linha 4      //métodos construtor, set's e get's
Linha 5
Linha 6
Linha 7      public void setSalario(double dsalario){
Linha 8          if (cargo.equalsIgnoreCase("Tecnico Administrativo"))
Linha 9              super.setSalario(dsalario+(300));
Linha 10         else
Linha 11             if (cargo.equalsIgnoreCase("Bibliotecaria"))
Linha 12                 super.setSalario(dsalario+(400));
Linha 13             else
Linha 14                 super.setSalario(dsalario+(500));
Linha 15
Linha 16     }
Linha 17 }

```

O mesmo acontece com a redefinição do método `setSalario()` na classe `Administrativo`. São testadas todas as opções de valores do atributo `cargo`, pois, dependendo de cada uma das opções de valores (Técnico administrativo, Bibliotecária ou Técnico de laboratório) um determinado valor adicional é acrescido ao salário.

Vamos fazer uma pequena modificação no código da classe `CadastaAdmDocente`, já mostrado anteriormente nessa unidade.

```

Linha 1  import javax.swing.JOptionPane;
Linha 2  public class CadastaAdmDocente{
Linha 3      public static void main(String args[]){
Linha 4          Docente d1= new Docente();
Linha 5          d1.setNome(JOptionPane.showInputDialog("Entre com o Nome:"));
Linha 6          d1.setEndereco(JOptionPane.showInputDialog("Entre com o
Linha 7              Endereco:"));
Linha 8          d1.setNha(Integer.parseInt(JOptionPane.showInputDialog("Entre
Linha 9              com o Nu.de Horas Aula Semanais:")));
Linha 10         d1.setTitulacao(JOptionPane.showInputDialog("Entre com a
Linha 11             Titulacao:"));
Linha 12         d1.setSalario(Double.parseDouble(JOptionPane.InputDialog("Entre
Linha 13             com o Salario:")));
Linha 14         Administrativo a1= new Administrativo();

```

A chamada ao método `setSalario()` deve ser feita depois da chamada aos métodos `setNha()` e `setTitulação` pois o método `setSalario()` na sua nova implementação na classe `Docente` precisa dos valores dos atributos `nha` e `titulacao`.

A chamada ao método `setSalario()` deve ser feita depois da chamada ao método `setCargo()` pois o método `setSalario()` na sua nova implementação na classe `Administrativo` precisa do valor do atributo `cargo`.

```

Linha 11  a1.setNome(JOptionPane.showInputDialog("Entre com o Nome:"));
Linha 12  a1.setEndereco(JOptionPane.showInputDialog("Entre com o Endereco:"));
Linha 13  a1.setCargo(JOptionPane.showInputDialog("Entre com o Cargo:"));
Linha 14  a1.setSetor(JOptionPane.showInputDialog("Entre com o Setor:"));
Linha 15  a1.setSalario(Double.parseDouble(JOptionPane.showInputDialog("Entre
com o Salario:"))));
Linha 16  System.exit(0);
Linha 17  }
Linha 19  }
```



Síntese

Nessa unidade você aprendeu outro conceito importante da OO, o conceito de herança. É através da herança que a característica de reutilização de código do paradigma orientado a objeto se torna mais evidente. Isso acontece porque podemos modelar e implementar numa classe genérica, chamada de superclasse, todos os atributos e comportamentos comuns a diversos tipos de objetos e, esses atributos e comportamento, podem ser herdados por várias outras classes (que representam esses tipos de objetos diferentes com atributos e comportamentos em comum) evitando assim a duplicação de código.



Atividades de auto-avaliação

- 1) Desenvolva um sistema orientado a objetos para automatizar as informações de uma Administradora de Imóveis.
 - Essa Administradora necessita de um sistema para automatizar o conjunto de informações relativas aos imóveis que administra.
 - Sobre qualquer imóvel se necessita saber o nome do proprietário, o endereço, se é para venda ou aluguel, valor (venda ou aluguel).
 - Os imóveis são divididos em duas categorias: Apto e Casa. Sobre os imóveis do tipo Apto é necessário saber o andar e número do apto. Sobre os imóveis do tipo casa é necessário saber se tem piscina ou não.
 - Modele o sistema descrito acima, identificando os objetos do problema, se existem atributos e comportamentos em comum, etc.
 - Represente através do Diagrama de classe da UML.

Herança na prática



Objetivo de aprendizagem

- Fortalecer o conceito de herança mediante exemplos práticos.



Seção de estudo

Seção 1 Herança mediante exemplo prático



Para início de conversa

Na unidade anterior, você aprendeu que ao modelar um problema que possua no mínimo dois tipos de objetos, e, que esses objetos possuam alguns atributos e comportamentos em comum e igualmente atributos e comportamentos específicos para cada tipo, é interessante representar esses atributos e comportamentos em comum, mediante uma classe genérica chamada de superclasse, e os atributos e comportamentos específicos em classes conhecidas como subclasses, que representam cada tipo de objeto. A herança acontece porque os objetos dessas subclasses herdam todos atributos e comportamentos definidos na superclasse.

Nesta unidade, você irá fortalecer o conceito de herança e os vários conceitos relacionados através de alguns exemplos práticos.

Vamos colocar a “mão na massa” e aprender um pouco mais?

SEÇÃO 1 -Herança através de exemplo prático

Vamos modelar no paradigma orientado a objetos o seguinte sistema:

Requisitos:

Uma empresa de locação de veículos necessita de um sistema para armazenar as informações de sua frota de veículos. Para todos os tipos de veículos, é necessário armazenar informações como: placa, marca, modelo, ano, valor do km rodado, km inicial (antes da locação), km final (depois da locação). Essa locadora de veículos possui duas categorias de veículos para locação: Passeio e Carga. O total da frota é 15 veículos, sendo 10 do tipo Passeio e 5 do tipo Carga.

Para os veículos do tipo Passeio, é necessário armazenar se o veículo possui ar-condicionado e o número de portas (2 ou 4).

Para os veículos do tipo Carga, é necessário armazenar a capacidade de carga em toneladas do veículo.

Todos os veículos cadastrados possuem um valor de locação.

Esse valor de locação é calculado da seguinte forma:

$$\text{nº de km rodados} * \text{valor do km rodado}$$

O número de km rodados é obtido através do seguinte cálculo:

$$\text{km final} - \text{km inicial}$$

Nesse sistema os veículos do tipo Carga sofrem no cálculo de locação:

$$(\text{nº de km rodados} * \text{valor do km rodado}) + 10\%.$$

Além de armazenar as informações acima, o sistema deve permitir as seguintes consultas (listagens):

- Listagem de todos os veículos cadastrados;
- consulta de valor de locação mediante a digitação da placa do veículo;
- número de veículos com ar-condicionado;

Depois da apresentação dos requisitos do sistema, vamos começar a modelar o sistema dentro do paradigma orientado a objetos.

Modelagem:

- Identificar quais os objetos existentes nesse problema:
É necessário armazenar informações sobre objetos do tipo Veículo Carga e Veículo Passeio.
- Identificar atributos e comportamentos dos objetos identificados.

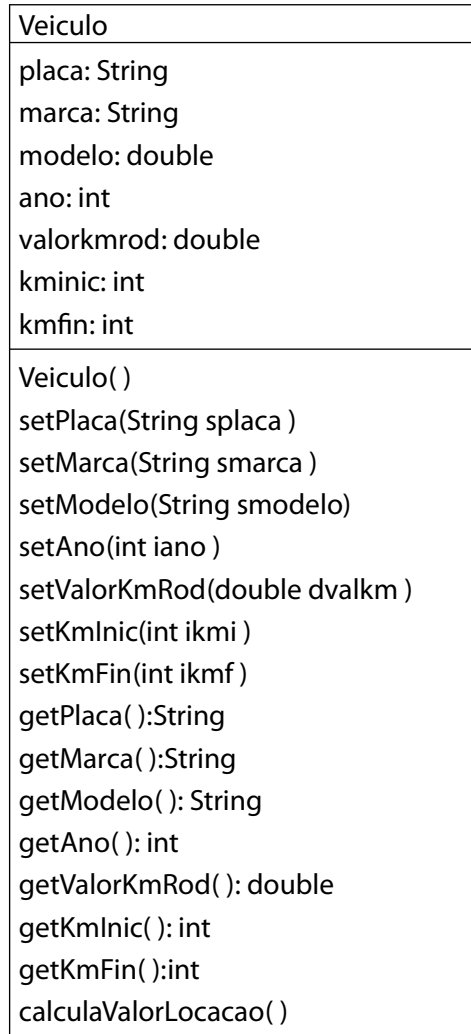
Veículo Passeio	Veículo Carga
Atributos	
Placa	Placa
Marca	Marca
Modelo	Modelo
Ano	Ano
valor do km rodado	valor do km rodado
km inicial	km inicial
km final	km final
Ar-condicionado	capacidade
Número de portas	
Comportamentos ou métodos	
Passeio ()	Carga ()
Métodos set e set para cada atributo	Métodos set e set para cada atributo
calculaValorLocacao()	calculaValorLocacao()

Pela tabela acima podemos identificar que existe um conjunto de atributos e métodos em comum entre os objetos Passeio e Carga.

Diante disso, podemos optar por representar esses atributos e métodos em uma classe genérica (superclasse) chamada Veículo e representar os atributos e métodos específicos nas classes Passeio e Carga.

A seguir está a representação em UML das classes desse sistema.

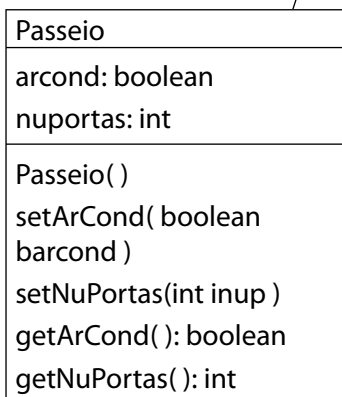
Superclasse



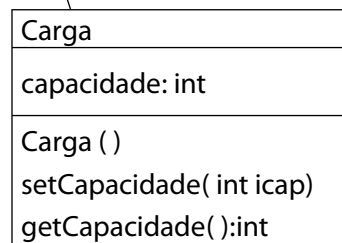
A seta apontada para a superclasse indica o relacionamento de herança.

Carga é **um tipo** de Veiculo.
Passeio é um tipo de Veiculo.

Subclasse



Subclasse



Depois de modelado, vamos partir para a implementação das classes.

Implementação:

```
//Implementação da superclasse veiculo
public class Veiculo {
    private String placa, marca, modelo;
    private int ano, kminic, kmfin;
    private double valorkmrod;
    public Veiculo(){
        placa="";
        marca="";
        modelo="";
        ano=0;
        valorkmrod=0;
        kminic=0;
        kmfin=0
    }

    public void setPlaca(String splaca){
        placa=splaca;
    }

    public void setMarca(String smarca){
        marca=smarca;
    }

    public void setModelo(String smodelo){
        modelo=smodelo;
    }

    public void setAno(int iano){
        ano=iano;
    }

    public void setValKmRod(double dvalkmrod){
        valkmrod=dvalkmrod;
    }

    public void setKminic(int ikmi){
        kminic= ikmi;
    }

    public void setKmFim(int ikmf){
        kmfim= ikmf;
    }

    public String getPlaca(){
        return placa;
    }

    public String getMarca(){
        return marca;
    }
}
```



```

        public String getModelo(){
            return modelo;
        }

        public int getAno(){
            return ano;
        }

        public double getValKmRod(){
            return valkmrod;
        }

        public int getKmInic(){
            return kminic;
        }

        public int getKmFim(){
            return kmfim;
        }

        public double calculaValorLocacao(){
            return (kmfim – kminic) * valkmrod;
        }
    }

```

O método **calculaValorLocação ()** deve existir para Passeio e Carga. Logo ele está implementado aqui na superclasse Veículo e será **HERDADO** por Passeio e Carga. O método contém a fórmula que determina o valor de locação do veículo.

Note que Passeio **extends** (estende) Veiculo. É essa palavra que indica o relacionamento de herança entre essas duas classes.

//Implementação da subclasse Passeio

```

public class Passeio extends Veiculo{
    private int nuportas;
    private boolean arcond;
    public Passeio (){
        super ();
        portas=0;
        arcond=false;
    }
    public void setNuPortas (int inup){
        nuportas=inup;
    }
    public void setArCond (boolean barcond){
        arcond=barcond;
    }
    public int getNuPortas (){
        return nuportas;
    }
    public boolean getArCond (){
        return arcond;
    }
}

```

O método **calculaValorLocação** () está sendo sobrescrito na classe Carga. Isso acontece porque na classe Carga, o método de calcular o valor da locação é diferente. Lembrando: $((\text{kmfin} - \text{kminic}) * \text{valorkmrod}) + 10\%$. Logo, a classe Carga não deseja a herança do método **calculaValorLocação** () da maneira como ele foi implementado na classe Veículo. Com isso, esse método pode ser redefinido na classe Carga com a fórmula correta.

//Implementação da subclasse Carga

```
public class Carga extends Veiculo{
    private int capacidade;
    public Carga (){
        super ();
        capacidade=0;
    }
    public void setCapacidade (int icap){
        capacidade=icap;
    }
    public int getCapacidade (){
        return capacidade;
    }
    public double calculaValorLocacao(){
        return ((getKmFim() – getKmlnic()) *
        getValKmRod())*1.1;
    }
}
```

Depois de implementar as classes básicas do sistema, vamos partir para a implementação do sistema propriamente dito. O sistema deve ser capaz de armazenar informações para 10 veículos do tipo Carga e, 5 veículos do tipo Passeio (total da frota de veículos) e atender aos outros requisitos propostos na descrição do sistema no início dessa unidade.

Vamos revisar os requisitos de saída do sistema?

- Listagem de todos os veículos cadastrados;
- Consulta de valor de locação através da digitação da placa do veículo;
- Número de veículos com ar-condicionado.

```

Linha 0  import javax.swing.*;
Linha 1  public class CadastraVeiculosLocacao{
Linha 2      public static void main(String args[]){
Linha 3          Passeio p[]=new Passeio[10];
Linha 4          Carga c[]=new Carga[5];
Linha 5          int pp=0,pc=0;
Linha 6          int op = 0;
Linha 7          while (op!=6){
Linha 8              op=Integer.parseInt(JOptionPane.showInputDialog("Digite a opção \n"+
Linha 9                  "1= CADASTRAR VEICULO TIPO PASSEIO \n"+
Linha 10                 "2= CADASTRAR VEICULO TIPO CARGA \n"+
Linha 11                 "3= LISTAGEM DOS VEICULOS CADASTRADOS \n"+
Linha 12                 "4= CONSULTA VALOR DE LOCAÇÃO\n" +
Linha 13                 "5= QUANTIDADE DE CARROS COM AR-CONDICIONADO\n"+
Linha 14                 "6= SAIR"));
Linha 15
Linha 16              if (op==1) {
Linha 17                  p[pp]=new Passeio( );
Linha 18                  p[pp].setPlaca(JOptionPane.showInputDialog("Digite a placa"));
Linha 19                  p[pp].setMarca(JOptionPane.showInputDialog("Digite a marca"));
Linha 20                  p[pp].setModelo(JOptionPane.showInputDialog("Digite o modelo"));
Linha 21                  p[pp].setAno(Integer.parseInt(JOptionPane.showInputDialog("Digite o ano da Fabricação")))
Linha 22                  p[pp].setValKmRod(Double.parseDouble(JOptionPane.showInputDialog("Digite o valor do Kilometro Rodado")));
Linha 23                  p[pp].setKmInic(Integer.parseInt(JOptionPane.showInputDialog("Digite a Kilometragem Inicial")));
Linha 24                  p[pp].setKmFim(Integer.parseInt(JOptionPane.showInputDialog("Digite a Kilometragem Final")));
Linha 25                  p[pp].setNuPortas(Integer.parseInt(JOptionPane.showInputDialog("Digite o nº de portas")));
Linha 26                  String resp=JOptionPane.showInputDialog("Possui ar-condicionado? Sim ou Não?");
Linha 27                  if(resp.equalsIgnoreCase("sim"))
Linha 28                      p[i].setArCond(true);
Linha 29                  else
Linha 30                      p[i].setArCond(false);
Linha 31                  pp=pp+1;
Linha 32              } //FIM opção 1
Linha 33              else
Linha 34                  if (op==2) {
Linha 35                      c[pc]=new Carga( );
Linha 36                      c[pc].setPlaca(JOptionPane.showInputDialog("Digite a placa"));
Linha 37                      c[pc].setMarca(JOptionPane.showInputDialog("Digite a marca"));
Linha 38                      c[pc].setModelo(JOptionPane.showInputDialog("Digite o modelo"));
Linha 39                      c[pc].setAno(Integer.parseInt(JOptionPane.showInputDialog("Digite o ano da fabricação")));
Linha 40                      c[pc].setValKmRod(Double.parseDouble(JOptionPane.showInputDialog("Digite o Valor do
Kilometro Rodado")));
Linha 41                      c[pc].setKmInic(Integer.parseInt(JOptionPane.showInputDialog("Digite a Kilometragem
Inicial")));
Linha 42                      c[pc].setKmFim(Integer.parseInt(JOptionPane.showInputDialog("Digite a Kilometragem
Final")));
Linha 43                      c[pc].setCapacidade(Integer.parseInt(JOptionPane.showInputDialog("Digite a capacidade
máxima permitida")));

```

```

Linha 44         pc = pc + 1;
Linha 44     } //FIM opção == 2
Linha 45     else
Linha 46         if (op == 3) {
Linha 47             for (int i=0; i<pp; i++){
Linha 48                 JOptionPane.showMessageDialog(null,"LISTAGEM DE VEICULOS TIPO PASSEIO: " + "\n" +
Linha 50                     "PLACA: " + p[i].getPlaca () + "\n" +
Linha 51                     "MARCA: " + p[i].getMarca() + "\n" +
Linha 52                     "MODELO: " + p[i].getModelo() + "\n" +
Linha 53                     "ANO DE FABRICAÇÃO: " + p[i].getAno () + "\n" +
Linha 54                     "VALOR KM RODADO: " + p[i].getValKmRod() + "\n" +
Linha 55                     "KM INICIAL: " + p[i].getKmInic() + "\n" +
Linha 56                     "KM FINAL: " + p[i].getKmFin() + "\n" +
Linha 57                     "NUMEROS DE PORTAS: " + p[i].getNuPortas() + "\n");
Linha 58             }
Linha 59             for (int i=0; i<pc; i++){
Linha 60                 JOptionPane.showMessageDialog(null,"LISTAGEM VEICULOS TIPO CARGA: " + "\n" +
Linha 61                     "PLACA: " + c[i].getPlaca () + "\n" +
Linha 62                     "MARCA: " + c[i].getMarca() + "\n" +
Linha 63                     "MODELO: " + c[i].getModelo() + "\n" +
Linha 64                     "ANO DE FABRICAÇÃO: " + c[i].getAno () + "\n" +
Linha 65                     "VALOR KM RODADO: " + c[i].getValKmRod() + "\n" +
Linha 66                     "KM INICIAL: " + c[i].getKmInic() + "\n" +
Linha 67                     "KM FINAL: " + c[i].getKmFin() + "\n" +
Linha 68                     "CAPACIDADE MAXIMA: " + c[i].getCapacidadea() + "\n");
Linha 69             }
Linha 70         } //FIM opcao == 3
Linha 71     else
Linha 72         if (op == 4) {
Linha 74             String placa=JOptionPane.showInputDialog("Digite a PLACA do veiculo a ser pesquisado");
Linha 75             for (int i=0; i<pp; i++){
Linha 76                 if (p[i].getPlaca().equalsIgnoreCase(placa)){
Linha 77                     JOptionPane.showMessageDialog(null,"O valor de locação desse veículo é:
+ p[i].calculaValorLocacao( );
Linha 78                 }
Linha 79             }
Linha 80             for (int i=0; i<pc; i++){
Linha 81                 if (c[i].getPlaca().equalsIgnoreCase(placa)){
Linha 82                     JOptionPane.showMessageDialog(null,"O valor de locação desse veículo é:
+ c[i].calculaValorLocacao( );
Linha 83                 }
Linha 84             } //fim do for
Linha 85         } //fim da opção 4
Linha 86     else
Linha 87         if (op == 5){
Linha 88             int cot=0;
Linha 89             for (int i=0; i<10; i++){

```

```

Linha 90      if (c[i].getArCond() == true)
Linha 91      cont = cont + 1;
Linha 92      }
Linha 93      JOptionPane.showMessageDialog(null,"Número de Veículos do Tipo Passeio com
               Ar-Condicionado: " + cont );
Linha 94      }
Linha 95      else
Linha 96      if (op == 6)
Linha 97      break;
Linha 98  }// fim while
Linha 99  }// fim do método main ( )
Linha 100} // fim da classe

```

Vamos analisar o código, linha a linha:

Linhas 3 e 4: criação do vetor para armazenar os objetos Carga e Passeio. Lembre que o vetor não contém os objetos ainda. Ele está inicializado com null.

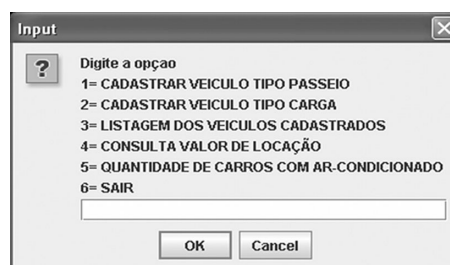
Linha 5: criação e inicialização das variáveis pp e pc. Essas variáveis controlarão as posições dentro do vetor p (Passeio) e c (Carga) onde os objetos Passeio e Carga serão criados.

Linha 6: criação da variável op e inicialização dela com o valor 0.

Linha 7: início da estrutura de repetição, que fará com que o programa volte à tela de escolha das opções (menu) depois de ser executada a opção digitada pelo usuário. Enquanto a opção digitada for diferente de 6 (opção de Sair), o programa segue sendo executado.

Linha 8: entrada de dados da opção desejada. O usuário deverá digitar o número equivalente à opção desejada. A instrução \n faz com que o texto seguinte seja impresso na próxima linha da caixa de diálogo de entrada.

A tela de entrada de dados será igual à mostrada a seguir:



Linha 16: verifica se o usuário digitou a opção 1 (significa que ele quer cadastrar um veículo do tipo Passeio). Se for verdade, somente as instruções das linhas 17 a 31 serão executadas;

Linha 17: lembre-se que o vetor p do tipo Passeio foi criado e está vazio. Essa instrução cria um objeto do tipo Passeio e associa a próxima posição livre do vetor p. A próxima posição livre do vetor p é controlada pela variável pp. Na primeira vez que entrar nessa opção, o valor da variável pp será 0, logo, o objeto Passeio, será criado na posição 0 do vetor. Note que a variável pp é incrementada na linha 30. Isso significa que, da próxima vez que o usuário quiser cadastrar um veículo do tipo Passeio, ou seja, que a opção 1 for digitada, o objeto Passeio será criado na posição 2 do vetor p.

Linhas 18 a 25: depois que o objeto Passeio é associado à determinada posição do vetor p, podem ser chamados os métodos públicos desse objeto Passeio, para armazenar os valores digitados pelo usuário nos atributos desse objeto. Note que todo objeto Passeio herda os atributos e métodos de Veículo, ou seja, quando um objeto Passeio é criado, ele possui em memória todos os atributos de Veículos + os atributos de Passeio.

Linha 26: para armazenar um valor lógico (true ou false) no atributo arCond de um objeto passeio, é necessário “tratar a entrada de dados desse valor”. O usuário não vai poder digitar valor true ou o valor false na caixa de diálogo de entrada de dados, portanto, pedimos que ele digite sim ou não (String), e nas linhas seguintes é testado. Se o valor digitado for “sim”, é chamado o método setArCond(true) e passado o valor lógico true, caso contrário, é passado o valor lógico false.

Linha 32: fim da opção 1.

Linha 34: verifica se a opção digitada é igual a 2 (significa que o usuário quis cadastrar veículos do tipo Carga).

Linha 35: lembre-se de que o vetor c do tipo Carga foi criado e está vazio. Essa instrução cria um objeto do tipo Carga e associa à próxima posição livre do vetor c. A próxima posição livre do vetor c é controlada pela variável pc. Na primeira vez que entrar nessa opção, o valor da variável pc será 0 logo, o objeto Carga será criado na posição 0 do vetor. Note que a variável pc é

incrementada na linha 44. Isso significa que da próxima vez que o usuário quiser cadastrar um veículo do tipo Carga, ou seja, que a opção 2 for digitada, o objeto Carga será criado na posição 2 do vetor c.

Linhas 36 a 43: depois que o objeto Carga é associado à determinada posição do vetor c, podem ser chamados os métodos públicos desse objeto Carga para armazenar os valores digitados pelo usuário nos atributos desse objeto. Note que todo objeto Carga herda os atributos e métodos de Veículo, ou seja, quando um objeto Carga é criado, ele possui em memória todos os atributos de Veículos + os atributos de Carga.

Linha 46: teste da opção 3 (significa que o usuário deseja consultar, listar todos os veículos cadastradas até o momento). Para isso, é necessário percorrer o vetor p de Passeio até a posição do último veículo Passeio cadastrado (quem guarda esse valor é a variável pp) e também o vetor Carga, acessando cada um dos objetos. Só assim é possível recuperar e mandar imprimir os valores dos atributos desses objetos.

Linha 47: estrutura de repetição for para determinar quais as posições do vetor p que serão acessadas a cada iteração. Note que a variável pp controla a última posição acessada do vetor p. Não se deve percorrer o vetor p até 10, porque não sabemos se existirão 10 objetos do tipo Passeio criados.

Linhas 48 a 57: imprimem na caixa de diálogo todos os valores dos atributos de um determinado objeto Passeio. Quem determina o objeto que está sendo acessado no momento, é a variável de controle da estrutura de repetição for, no caso, i. Por exemplo, quando i contiver o valor 1, é o objeto associado à posição 1 do vetor p que está sendo acessado.

Linhas 59 a 69: o mesmo acontece para o vetor c, já que a opção se destina a listar todos os veículos e os objetos do vetor c também são veículos.

Linha 72: verifica se opção digita é quatro (significa que o usuário deseja consultar o valor de locação de um determinado veículo). Para achar o valor de locação de um determinado veículo, o usuário deverá entrar com alguma informação, no caso, a placa do veículo. O sistema irá procurar no vetor p (Passeio)

e c (Carga) se encontra algum objeto com o valor do atributo placa igual à placa que o usuário digitou. Se encontrar, o método calculaValorLocacao() desse objeto será chamado.

Linha 74: entrada da placa do veículo sobre o qual se deseja calcular o valor de locação. A placa digitada será armazenada na variável de memória placa.

Linhas 75 a 79: o vetor p é percorrido até a posição do último objeto Passeio criado. Quem controla essa posição é a variável pp. Para cada objeto do vetor p, é testado que o valor do atributo placa é igual ao conteúdo da variável placa. Se for igual, é chamado o método calculaValorLocacao() desse objeto e o valor retornado é impresso na caixa de diálogo.

Linhas 80 a 83: o mesmo é feito com o vetor c, pois a placa digitada pode ser de um veículo carga.

Linhas 87 a 94: verifica se a opção digitada é 5 (significa que o usuário deseja saber o número de veículos do tipo Passeio que possuem ar-condicionado). Nessa opção é percorrido somente o vetor passeio, pois somente veículos Passeio possuem ar-condicionado, verificando se o atributo arcond possui o valor true. A cada vez que encontrar, é incrementada a variável cont. No final, é impresso o valor da variável cont.

Linha 96: verifica se a opção digitada é 6 (significa que o usuário deseja sair do sistema). Como toda a lógica de execução está dentro de uma estrutura de repetição while, é necessário forçar a saída dela. A instrução que faz isso é chamada de **break**. Quando essa instrução é executada, o fluxo de repetição é interrompido e, a próxima instrução a ser executada, {será a próxima depois do símbolo de } (fim) do while.



Síntese

Nessa unidade você reforçou o conceito de herança através de um outro exemplo prático.

Você estudou sobre modelagem e implementou um sistema para uma locadora de veículos. Nesse sistema identificou que existia dois tipos de objetos: Veículos Passeio e Veículos Carga e que existia uma série de atributos e comportamentos comuns entre esses dois tipos de objetos.

Diante dessa situação, modelamos esses atributos e comportamentos em comum dentro de uma classe genérica chamada Veículo e os atributos e comportamentos específicos de cada tipo de objeto foram modelagem em classes específicas.

A herança é um recurso muito útil na orientação a objetos porque propicia a reutilização de código, evitando assim o retrabalho de codificação.



Atividades de auto-avaliação

- 1) Implemente as classes modeladas no sistema da Administradora de Imóveis descrito na atividade da unidade anterior. Desenvolva um sistema que permita:
 - Cadastrar os imóveis do tipo Casa e do tipo Apto. (máximo 10 para cada)
 - Imprimir os seguintes dados de todos os imóveis: nome do proprietário, endereço e o tipo, Casa ou Apto.
 - Impressão o número de casas com piscina.
- 2) Implemente as classes e o sistema da Locadora de Veículos descrito nessa unidade.



Para concluir o estudo

Chegamos ao fim de mais uma disciplina: Programação Orientada a Objeto.

Foi um trabalho longo, em que você começou aprendendo a utilizar uma linguagem de programação (Java) colocando em pratica os conhecimentos de lógica obtidos em Lógica I e Lógica. Um pouco menos da metade das unidades dessa disciplina foram reservados a essa atividade: apresentá-lo a uma linguagem de programação.

Escolhemos a linguagem Java por ser bastante utilizada no ambiente de desenvolvimentos de softwares, por ser free e por ser orientada a objetos. Foi importante você ter esse contato com a parte “prática” do desenvolvimento de programas para conseguir assimilar melhor os conceitos relacionados ao paradigma orientado a objetos.

Mais da metade das unidades foi dedicada à explanação dos conceitos do paradigma orientado a objetos.

Esse paradigma propõe uma forma de pensar e implementar sistemas, em que devemos pensar e, conseqüentemente, implementar o problema tal como ele acontece no mundo real, identificando os objetos que atuam nesse problema, identificando atributos e comportamentos desses objetos e os relacionamentos entre os mesmos. Um sistema orientado a objetos é composto de uma coleção de objetos que se relacionam e trocam mensagens entre si.

Todos os conceitos de OO apresentados a partir da unidade 7 são indispensáveis para você seguir em frente com os estudos. Proporcionarão, até o final do curso, o desenvolvimento de um sistema completo.

Espero que você tenha gostado!!



Referências

DEITEL, H.M.; DEITEL, P.J. **Java - Como programar**. Sexta Edição. Ed. Pearson, 2005.

LEMAY, Laura; CADENHEAD, Rogers. **Aprenda em 21 dias Java 1.2**. Rio de Janeiro: Campus, 1999.

HORSTMANN, Cay S.; CORNELL, Gary. **Core Java 2**. São Paulo: Makron Books, 2004.

HORSTMANN, Cay S. **Big Java**. Porto Alegre: Bookman, 2004.

The Java Tutorial - A practical guide for programmers.
Disponível em: <http://java.sun.com/tutorial>

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML – Guia do Usuário**. Rio de Janeiro: Campus, 2000.

FOWLER, Martin; SCOTT, Kendall. **UML essencial**: um breve guia para a linguagem-padrão de modelagem de objetos. Segunda Edição. Porto Alegre: Bookman, 2000.

Sobre a professora conteudista

Andréa Sabedra Bordin

Graduada em Análise de Sistemas. Especialista em Sistemas de Informação. Mestre em Ciência da Computação. Doutoranda em Engenharia e Gestão do Conhecimentos pela UFSC. Professora dos Cursos de Sistemas de Informação e Ciência da Computação da Unisul. Analista de Sistemas do Instituto Stela.





Respostas e comentários das atividades de auto-avaliação

Unidade 1

1) Um programa feito em uma linguagem de programação de alto nível não pode ser entendido diretamente nessa linguagem pelo computador. O que precisa ser feito com esse programa em alto nível para que suas instruções possam ser entendidas pelo computador?

Resposta: O programa precisa ser interpretado ou compilado previamente antes das suas instruções serem executadas. O processo de interpretação consiste em transformar cada instrução programada para a linguagem binária no momento em que o programa está sendo executado. É um processo que deixa a execução do programa mais lento. A cada vez que o programa for executado esse processo irá se repetir.

O processo de compilação consiste em transformar cada instrução programada para a linguagem binária numa fase anterior a execução do programa. Depois que todo o conjunto de instruções for passado para linguagem binária o programa está pronto para ser executado. Uma vez compilado sem problemas, o programa pode ser sempre executado sem necessidade de passar pelo processo de compilação novamente.

2) Explique a relação existente entre a característica Multiplataforma da linguagem Java e a expressão “write once run anywhere”.

Resposta: A característica multiplataforma da linguagem Java e a expressão “write once run anywhere” estão ligadas porque quando se desenvolve um programa na linguagem Java é necessário compilar esse programa uma vez. Depois de compilado ele será executado em outras plataformas de hardware/sistema operacional sem precisar voltar ao programa e alterar o mesmo, ou seja, escreva o código uma vez e rode em qualquer lugar.

3) Explique o que é a máquina virtual (JVM - *Java Virtual Machine*) da linguagem Java.

Resposta: A máquina virtual da linguagem Java é um software que permite que os programas desenvolvidos e compilados (bytecodes) nela sejam executados. Um programa em Java não é executado diretamente pelo sistema operacional, ele passa por essa “camada” que é a JVM.

Unidade 2

1) Desenvolva um pequeno programa na linguagem Java que imprima o seu nome da tela. Mude o nome do programa para MeuNome.java, ou seja, no lugar de AloMundo você deve digitar MeuNome.

Resposta:

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

Unidade 3

1) Faça as seguintes declarações de variáveis na linguagem Java:

a. Declare uma variável de memória para armazenar um valor numérico inteiro, um valor numérico real, um valor do tipo lógico, um valor do tipo Literal.

Resposta:

```
int num;  
double numreal;  
boolean status;  
String endereço;
```

b. Declare um vetor de 5 posições para armazenar valores inteiros.

Resposta:

```
int val[]=new int[5]
```

- c. Declare uma matriz de 3 x 2 para armazenar valores do tipo literal.

Resposta:

```
String nome[][]=new String [3][2]
```

- 2) Produza o seguinte trecho de código na linguagem Java:

Resposta: Criar uma variável do tipo literal e armazenar o valor "Unisul"; criar outra variável do tipo literal e armazenar o valor "Virtual"; concatenar (unir) essas duas variáveis e armazenar em uma terceira variável.

```
String X="Unisul";
String Y="Virtual";
String XY= X + Y;
```

- 3) Traduza os seguintes trechos de pseudocódigo para a linguagem Java:

Resposta:

A,B: numérico
 A ← 5
 B ← 6
 C ← A+B
Escreva C;

```
int A,B;
A = 5;
B = 6;
C = A +B;
JOptionPane.showMessageDialog(null, "Valor de C é " + C);
```

- 4) Escreva um trecho de código em Java que leia o nome de uma pessoa e a sua idade. Logo após, escreva o nome da pessoa e a sua idade.

Resposta:

```
String nome;
int idade;
nome = JOptionPane.showInputDialog("Entre com o nome ");
idade = Integer.parseInt(JOptionPane.showInputDialog("Entre com a idade "));
JOptionPane.showMessageDialog(null, "O nome da pessoa é " + nome + " e sua idade é " +
idade);
```

Unidade 4

1) Desenvolver em Java TODOS os exercícios de auto-avaliação da seção 1 da Unidade 5 da disciplina de Lógica de Programação I.

Resposta:

Algoritmo 1

```
import javax.swing.*;

public class Compra{
    public static void main(String args[])
    {
        String CLIENTE;
        int QTDHOT, QTDXEGG, QTDREFRI, QTDBATATA;
        double PREHOT, PREXEGG, PREREFRI, PREBATATA, TOTAL;
        // QTDHOT = Quantidade de Cachorro-Quente
        // QTDXEGG = Quantidade de XEGG;
        // QTDREFRI = Quantidade de Refrigerante;
        // QTDBATATA = Quantidade de Batatas Fritas;
        // TOTAL = Valor total comprado

        CLIENTE=JOptionPane.showInputDialog("Digite o nome do cliente:");
        QTDHOT= Integer.parseInt(JOptionPane.showInputDialog("Digite a quantidade de Cachorro-Quente:"));
        PREHOT= Double.parseDouble(JOptionPane.showInputDialog("Digite o preço do Cachorro-Quente:"));
        QTDXEGG= Integer.parseInt(JOptionPane.showInputDialog("Digite a quantidade de X-EGG:"));
        PREXEGG= Double.parseDouble(JOptionPane.showInputDialog("Digite o preço do X-EGG:"));
        QTDREFRI= Integer.parseInt(JOptionPane.showInputDialog("Digite a quantidade de refrigerantes:"));
        PREREFRI= Double.parseDouble(JOptionPane.showInputDialog("Digite o preço do refrigerante:"));
        QTDBATATA= Integer.parseInt(JOptionPane.showInputDialog("Digite a quantidade de batatas fritas:"));

        PREBATATA= Double.parseDouble(JOptionPane.showInputDialog("Digite o preço da batata frita:"));
        TOTAL = (QTDHOT * PREHOT) + (QTDXEGG * PREXEGG) + (QTDREFRI * PREREFRI) + (QTDBATATA * PREBATATA);
        JOptionPane.showMessageDialog(null,"Cliente "+CLIENTE);
        JOptionPane.showMessageDialog(null,"Total "+TOTAL);
    }
}
```

Algoritmo 2

```
import javax.swing.*;

public class Aluno{
    public static void main(String args[])
    {
        String nome, endereco, sexo, cidade, estado, nomedopai, nomedamae, telefonecontato;
        String datadenascimento, grauescolar;
        nome=JOptionPane.showInputDialog("Digite o nome do aluno:");
```

```

endereco= JOptionPane.showInputDialog("Digite o endereco:");
sexo= JOptionPane.showInputDialog("Digite o sexo:");
cidade= JOptionPane.showInputDialog("Digite a cidade:");
estado= JOptionPane.showInputDialog("Digite o estado:");
nomedopai= JOptionPane.showInputDialog("Digite o nome do pai:");
nomedamae= JOptionPane.showInputDialog("Digite o nome da mãe:");
telefonecontato= JOptionPane.showInputDialog("Digite o telefone:");
datadenascimento= JOptionPane.showInputDialog("Digite a data de nascimento:");
grauescolar= JOptionPane.showInputDialog("Digite o grau escolar:");

JOptionPane.showMessageDialog(null,"Nome "+nome);
JOptionPane.showMessageDialog(null,"Endereco "+endereco);
JOptionPane.showMessageDialog(null,"Sexo "+sexo);
JOptionPane.showMessageDialog(null,"Cidade "+cidade);
JOptionPane.showMessageDialog(null,"Estado "+estado);
JOptionPane.showMessageDialog(null,"Nome do pai "+nomedopai);
JOptionPane.showMessageDialog(null,"Nome da mae "+nomedamae);
JOptionPane.showMessageDialog(null,"Telefone de contato "+telefonecontato);
JOptionPane.showMessageDialog(null,"Data de nascimento "+datadenascimento);
JOptionPane.showMessageDialog(null,"Grau escolar "+grauescolar);
}
}

```

Algoritmo 3

```

import javax.swing.*;
public class Professor{
    public static void main(String args[])
    {
        String nome, endereco, cidade, estado, cep ;
        String datadenascimento,grauescolar, curso;
        int rg;

        nome=JOptionPane.showInputDialog("Digite o nome do aluno:");
        endereco= JOptionPane.showInputDialog("Digite o endereco:");
        cidade= JOptionPane.showInputDialog("Digite a cidade:");
        estado= JOptionPane.showInputDialog("Digite o estado:");
        cep= JOptionPane.showInputDialog("Digite o cep:");
        datadenascimento= JOptionPane.showInputDialog("Digite a data de nascimento:");
        grauescolar= JOptionPane.showInputDialog("Digite o grau de escolaridade:");
        curso= JOptionPane.showInputDialog("Digite o curso que leciona:");
        rg= Integer.parseInt(JOptionPane.showInputDialog("Digite o RG:"));

        JOptionPane.showMessageDialog(null,"Nome "+nome);
        JOptionPane.showMessageDialog(null,"Endereco "+endereco);
        JOptionPane.showMessageDialog(null,"Cidade "+cidade);
        JOptionPane.showMessageDialog(null,"Estado "+estado);
    }
}

```

```

JOptionPane.showMessageDialog(null,"CEP "+cep);
JOptionPane.showMessageDialog(null,"Data de nascimento "+datadenascimento);
JOptionPane.showMessageDialog(null,"Grau de escolaridade "+graueducacional);
JOptionPane.showMessageDialog(null,"Curso que leciona "+curso);
JOptionPane.showMessageDialog(null,"RG "+rg);
}
}

```

Algoritmo 4

```

import javax.swing.*;
public class Media{
    public static void main(String args[])
    {
        double nota1, nota2, nota3, media;
        String nomedisci;
        int numeroturma;

        nota1=Double.parseDouble(JOptionPane.showInputDialog("Digite a primeira nota:"));
        nota2=Double.parseDouble(JOptionPane.showInputDialog("Digite a segunda nota:"));
        nota3=Double.parseDouble(JOptionPane.showInputDialog("Digite a terceira nota:"));
        nomedisci= JOptionPane.showInputDialog("Digite o nome da disciplina:");
        numeroturma= Integer.parseInt(JOptionPane.showInputDialog("Digite o numero da turma:"));

        JOptionPane.showMessageDialog(null,"Nota 1 "+nota1);
        JOptionPane.showMessageDialog(null,"Nota 2 "+nota2);
        JOptionPane.showMessageDialog(null,"Nota 3 "+nota3);
        JOptionPane.showMessageDialog(null,"Disciplina "+nomedisci);
        JOptionPane.showMessageDialog(null,"Numero da turma "+numeroturma);
        JOptionPane.showMessageDialog(null,"Media Final "+(nota1+nota2+nota3)/3);

    }
}

```

Algoritmo 5

```

import javax.swing.*;
public class Conta{
    public static void main(String args[])
    {
        double limite, saldoatual, valor;
        String nomecor, nomebanco;
        int numeroconta;

        nomecor= JOptionPane.showInputDialog("Digite o nome do correntista:");
        nomebanco= JOptionPane.showInputDialog("Digite o nome do banco:");
        numeroconta= Integer.parseInt(JOptionPane.showInputDialog("Digite o numero da conta:"));
    }
}

```

```

    limite=Double.parseDouble(JOptionPane.showInputDialog("Digite o limite:"));
    saldoatual=Double.parseDouble(JOptionPane.showInputDialog("Digite o saldo atual:"));
    //entrar com um valor de deposito (credito)
    valor=Double.parseDouble(JOptionPane.showInputDialog("Digite o valor de deposito:"));
    saldoatual = saldoatual + valor;
    JOptionPane.showMessageDialog(null,"O saldo atual é "+saldoatual);

    //entrar com um valor de saque (debito)
    valor=Double.parseDouble(JOptionPane.showInputDialog("Digite o valor de saque:"));
    saldoatual = saldoatual - valor;
    JOptionPane.showMessageDialog(null,"O saldo atual é "+saldoatual);

}
}

```

Unidade 5

1) Desenvolver em Java todos os exercícios de auto-avaliação da seção 2 da Unidade 5 da disciplina de Lógica de Programação I (Exercícios com estrutura condicional)

Resposta:

Resolução Algoritmo 6.1

```

import javax.swing.*;
class Ordem{
public static void main(String args[]) {
    int A, B, C, MENOR, INTERMEDIARIO, MAIOR;
    A=Integer.parseInt(JOptionPane.showInputDialog("Digite o primeiro número: "));
    B=Integer.parseInt(JOptionPane.showInputDialog("Digite o segundo número: "));
    C=Integer.parseInt(JOptionPane.showInputDialog("Digite o terceiro número: "));
    if (A > B)
        if (C > A){
            MAIOR = C;
            INTERMEDIARIO = A;
            MENOR = B;
        }else{
            if (C > B){
                MAIOR = A;
                INTERMEDIARIO = C;
                MENOR = B;
            }else {

```

```

        MAIOR = A;
        INTERMEDIARIO = B;
        MENOR = C;
    }
}
else {
    if (C > B){
        MAIOR = C;
        INTERMEDIARIO = B;
        MENOR = A;
    }else {
        if (C > A){
            MAIOR = B;
            INTERMEDIARIO = C;
            MENOR = A;
        }else {
            MAIOR = B;
            INTERMEDIARIO = A;
            MENOR = C;
        }
    }
}
OptionPane.showMessageDialog(null,"O maior número é: "+MAIOR);
OptionPane.showMessageDialog(null,"O número intermediário é: "+INTERMEDIARIO);
OptionPane.showMessageDialog(null,"O menor número é: "+MENOR);
}
}

```

Resolução Algoritmo 6.2

```

import javax.swing.*;
public class MaiorMenor{
    public static void main(String args[])
    {
        int NUMERO;
        NUMERO=Integer.parseInt(JOptionPane.showInputDialog("Entre com um número "));
        JOptionPane.showMessageDialog(null,"Entre com um número é: "+NUMERO);
        if (NUMERO < 20)
            JOptionPane.showMessageDialog(null,"Número digitado é menor que 20 "+NUMERO);
        else
            if (NUMERO == 20)
                JOptionPane.showMessageDialog(null,"Número digitado é igual a 20 "+NUMERO);
            else
                JOptionPane.showMessageDialog(null,"Número digitado é maior que 20 "+NUMERO);
        }
    }
}

```


Resolução Algoritmo 6.3

```

import javax.swing.*;
class Intervalo{
public static void main(String args[])
{
int NUMERO;
NUMERO = 0;
NUMERO=Integer.parseInt(JOptionPane.showInputDialog("Digite o número "+NUMERO));
if ((NUMERO >= 30) && (NUMERO <= 90))
JOptionPane.showInputDialog("Número digitado esta dentro do intervalo considerado");
else
JOptionPane.showInputDialog("Número digitado esta fora do intervalo considerado");
}
}

```

Resolução Algoritmo 6.4

```

import javax.swing.*;
class Inss{
public static void main(String args[]) {
Double SAL_BRUTO, SAL_LIQUIDO;
SAL_BRUTO = 0.0;
SAL_BRUTO=Double.parseDouble(JOptionPane.showInputDialog("Digite o salário bruto do
trabalhador: "+SAL_BRUTO));
if (SAL_BRUTO <= 600)
SAL_LIQUIDO=SAL_BRUTO; // não há desconto
else
if ((SAL_BRUTO > 600) && (SAL_BRUTO <= 1200))
SAL_LIQUIDO=(SAL_BRUTO-SAL_BRUTO*0.2); //desconto de 20%
else
if ((SAL_BRUTO > 1200) && (SAL_BRUTO <=2000))
SAL_LIQUIDO=(SAL_BRUTO-SAL_BRUTO*0.25); //desconto de 25%
else
SAL_LIQUIDO=(SAL_BRUTO-SAL_BRUTO*0.3); //desconto de 30%

//saida do algoritmo
JOptionPane.showMessageDialog(null,"O salário líquido do trabalhador após os descontps de INSS é
de: "+SAL_LIQUIDO);
}
}

```

Resolução Algoritmo 6.5

```

import javax.swing.*;
public class VendaProduto{
public static void main(String args[])
{
double VCOMPRA, VVENDA;//valor de compra e de venda do produto

```

```

VCOMPRA=Double.parseDouble(JOptionPane.showInputDialog("Entre com valor de compra do produto: "));
if (VCOMPRA < 20)
VVENDA = (VCOMPRA + (VCOMPRA*0.5)); // 50% de lucro
else
VVENDA = (VCOMPRA + (VCOMPRA*0.35)); // 35% de lucro
JOptionPane.showMessageDialog(null,"O valor de venda do produto: "+VVENDA);
}
}

```

2) Desenvolver em Java todos os exercícios de auto-avaliação da Unidade 6 da disciplina de Lógica de Programação I (Exercícios com estrutura de repetição):

Resposta:

Algoritmo 1: (Ler 200 números e imprimir quantos são pares e quantos são ímpares)

```

import javax.swing.*;
public class RepeticaoParImpar{
    public static void main(String args[])
    {

        int num;
        for (int i=1;i<=200;i++){
            num = Integer.parseInt(JOptionPane.showInputDialog("Digite o numero:"));
            if ( num % 2 == 0 )
                JOptionPane.showMessageDialog(null,"Numero é par ");
            else
                JOptionPane.showMessageDialog(null,"Numero é impar ");
        }

    }
}

```

Algoritmo 2: (Entrar com 20 números e imprimir a soma dos positivos, e o total de números negativos)

```

import javax.swing.*;
public class PositivoNegativo{
    public static void main(String args[])
    {

        int num;
        int soma=0;
        int cont=0;
        for (int i=1;i<=20;i++){
            num = Integer.parseInt(JOptionPane.showInputDialog("Digite o numero:"));
            if ( num > 0 )

```

```

        soma = soma + num;
    else
        if (num < 0)
            cont=cont + 1;
    }
    JOptionPane.showMessageDialog(null,"Somatorio dos numeros positivos "+soma);
    JOptionPane.showMessageDialog(null,"Total de numeros negativos "+cont);
}

}

```

Algoritmo 3: Entrar com 10 números (positivos ou negativos) e imprimir o maior e o menor

```

import javax.swing.*;
public class MaiorMenor{
    public static void main(String args[])
    {

        int num;
        int maiornum=0;
        int menornum=999999;
        for (int i=1;i<=10;i++){
            num = Integer.parseInt(JOptionPane.showInputDialog("Digite o numero:"));
            if ( num > maiornum )
                maiornum = num;
            if ( num < menornum )
                menornum = num;
        }
        JOptionPane.showMessageDialog(null,"O maior numero é "+maiornum);
        JOptionPane.showMessageDialog(null,"O menor numero é "+menornum);
    }

}

```

Algoritmo 4: (entrar com o nome, idade e sexo de 20 pessoas. Imprimir o nome, se a pessoa for do sexo masculino e tiver mais de 21anos)

```

import javax.swing.*;
public class Maior21{
    public static void main(String args[])
    {
        String nome,sexo;
        int idade;
        for (int i=1;i<=20;i++){
            nome=JOptionPane.showInputDialog("Digite o nome do aluno:");
            idade= Integer.parseInt(JOptionPane.showInputDialog("Digite o endereco:"));

```

```
        sexo= JOptionPane.showInputDialog("Digite o sexo:");
        if ((idade > 21) && sexo.equalsIgnoreCase("masculino"))
            JOptionPane.showMessageDialog(null,"Nome "+nome);
    }

}

}
```

Unidade 6

1) Faça um programa em Java que leia 3 números e calcule a sua média. O programa deve ser modularizado. Utilize uma função para calcular e retornar a média dos números. Essa função deve receber os três números como parâmetro.

Resposta:

```
import javax.swing.*;

public class FuncaoMedia{
    public static void main(String args[])
    {
        int num1, num2, num3;
        double media;

        num1 = Integer.parseInt(JOptionPane.showInputDialog("Digite o primeiro numero:"));
        num2 = Integer.parseInt(JOptionPane.showInputDialog("Digite o segundo numero:"));
        num3 = Integer.parseInt(JOptionPane.showInputDialog("Digite o terceiro numero:"));
        media = calcMedia(num1,num2,num3);

        JOptionPane.showMessageDialog(null,"A media é " + media);

    }

    //aqui começa a funcao calcMedia
    public static double calcMedia(int n1,int n2, int n3){
        return (n1+n2+n3)/3;
    }
}
```

2) Crie um programa em Java que leia o raio de uma esfera (do tipo real) e passe esse valor para a função volumeEsfera. Essa função deve calcular o volume da esfera na tela e retornar o seu valor. Para o cálculo do volume deve ser usada a seguinte fórmula: $\text{Volume} = (4.0 / 3.0) * \text{PI} * \text{raio}^3$

Resposta:

```
import javax.swing.*;
public class FuncaoVolumeEsfera{
    public static void main(String args[])
    {

        double raio, volume;

        raio = Double.parseDouble(JOptionPane.showInputDialog("Digite o raio:"));
        volume = volumeEsfera(raio);

        JOptionPane.showMessageDialog(null,"O volume é " + volume);

    }

    //aqui começa a funcao
    public static double volumeEsfera(double draio){
        return (4.0 / 3.0) * 3.14 * (draio*draio*draio);
    }
}
```

3) Faça um programa em Java em que o usuário entre com um valor de base e um valor de expoente. O programa deve calcular a potência. O programa deve ser modularizado. Você deve decidir se a subrotina será um procedimento ou uma função. A formula é: $\text{base}^{\text{expoente}}$

Resposta:

```
import javax.swing.*;
public class FuncaoVolumeEsfera{
    public static void main(String args[])
    {

        int base, expoente, pot;

        base = Integer.parseInt(JOptionPane.showInputDialog("Digite a base:"));
        expoente = Integer.parseInt(JOptionPane.showInputDialog("Digite o expoente:"));

        pot = potencia(base, expoente);

        JOptionPane.showMessageDialog(null,"A potencia é " + pot);
    }

    //aqui começa a funcao
    public static int potencia(int base, int expoente){
        int pote=1;
        for (int i=1;i<=expoente;i++)
            pote=pote*base;

        return pote;
    }
}
```

Unidade 7

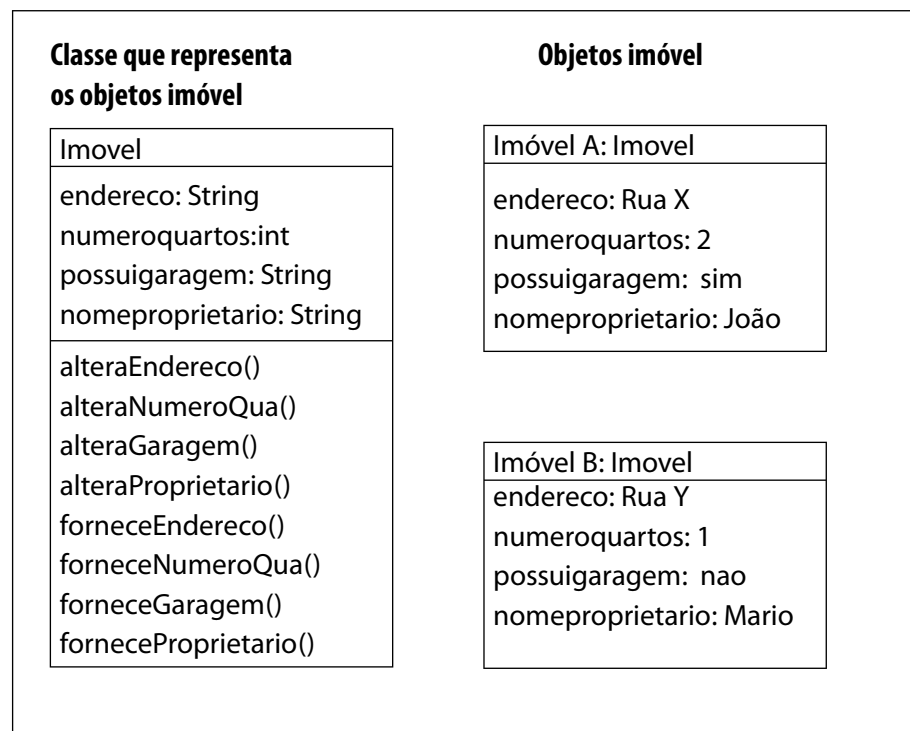
1) Edite a classe Cliente no editor de texto (Bloco de Notas ou qualquer ambiente para desenvolvimento de programas Java) e salve na pasta CURSOWEBOO. Essa pasta deve ser criada anteriormente. Após a edição (digitação) compile a classe. O arquivo Cliente.class deve ter sido gerado.

Resposta: O código já está na unidade.

2) Edite a classe UsaCliente no editor de texto. e salve na pasta CURSOWEBOO. Após a edição, compile a classe. O arquivo UsaCliente.class deve ter sido gerado. O ultimo passo é executar a classe. Dessa forma, você verá funcionando seu primeiro programa orientado a objeto desenvolvido na linguagem Java.

Resposta: O código já está na unidade.

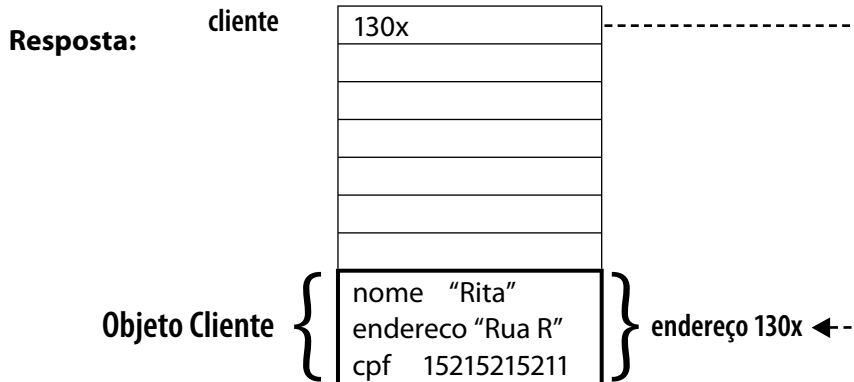
3) Cite exemplos de objetos do mundo real. Identifique atributos e comportamentos para esses objetos. Agrupe esses objetos nas suas respectivas classes. Faça isso usando a notação UML para classes e objetos.



Unidade 8

1) Represente graficamente em memória, tal como foi feito nessa unidade, a seguinte instrução. Identifique na representação gráfica o que é referência e o que é objeto.

Cliente cliente = new Cliente ("Rita", "Rua R", 15215215211);



2) Quando um método construtor é chamado?

Resposta: Quando um objeto de determinado tipo é criado em memória o método construtor implementado na classe desse objeto é chamado.

3) Qual a função de um método modificador numa classe? E de um método recuperador?

Resposta: Os métodos modificadores devem começar pelo nome set seguido do nome do atributo que esse método irá modificar o valor. Os métodos recuperadores devem começar pelo nome get seguido do nome do atributo que esse método irá recuperar o valor.

Unidade 9

A resposta é a implementação (programação) do código da unidade.

Unidade 10

1) Analise o seguinte trecho de código e identifique se existe algum erro. Caso exista, explique qual o erro e como solucioná-lo.

Resposta: Na classe `CalculoRetangulo`, os atributos `base` e `altura` do objeto `r` estão sendo acessados diretamente. Isso não é possível porque esses atributos foram definidos com o modificador `private` na classe `Retângulo`. Quando os atributos de um objeto são `private` (privados) eles só podem ser acessados via algum método `public` `set` ou `get`.

2) A seguinte classe (*ver o algoritmo ao final da unidade*) tem dois métodos `static`. Implemente outra classe chamada `UsaCalculos.java` e demonstre como esses métodos `static` podem ser chamados dessa classe.

Resposta:

```
import javax.swing.*;

public class UsaCalculos{
    public static void main(String args[])
    {
        /*estou chamando o metodo static potencia precedido do nome da classe onde ele está e estou
        passando dois parametros 2 - base e 3 - expoente. O valor de retorno desse metodo será impresso na
        tela*/
        System.out.println(Calculos.potencia(2,3));

        //idem para o metodo static fatorial
        System.out.println(Calculos.fatorial(5));

    }
}
```

Unidade 11

1) **Resposta:**

```
//Implementação da classe Cep

public class Cep
{
    private int codigo;
    private String numerua, bairro;

    public Cep()
```



```

    {
        codigo=0;
        nomerua=" ";
        bairro=" ";
    }
    public void setCodigo(int icodigo)
    {
        codigo=icodigo;
    }
    public void setNomeRua(String snome)
    {
        nomerua=snome;
    }
    public void setBairro(String sbairro)
    {
        bairro=sbairro;
    }
    public int getCodigo()
    {
        return codigo;
    }
    public String getNomeRua()
    {
        return nomerua;
    }
    public String getBairro()
    {
        return bairro;
    }
}
// Implementação da classe Funcionário (sem o atributo setor)

```

```

public class Funcionario
{
    private String nome, end;
    private double salario;
    private Cep cep;
    public Funcionario()
    {
        nome="";
        end="";
        salario=0;
        cep=null;
    }
    public void setNome(String snome)
    {

```

```
        nome=snome;
    }
    public void setEnd(String send)
    {
        end=send;
    }
    public void setSalario(double dsalario)
    {
        salario=dsalario;
    }
    public void setCep(Cep ccep)
    {
        cep=ccep;
    }
    public String getNome()
    {
        return nome;
    }
    public String getEnd()
    {
        return end;
    }

    public double getSalario()
    {
        return salario;
    }
    public Cep getCep()
    {
        return cep;
    }
}
```

//Implementação da classe que cadastra Funcionários

```
import javax.swing.*;
public class CadastroFuncCep
{
    public static void main(String args[])
    {
        Cep c[]=new Cep[3];

        for (int i=0; i<3; i++)
        {
            c[i]=new Cep();
        }
    }
}
```

```

        c[i].setCodigo(Integer.parseInt(JOptionPane.showInputDialog("Digite o código
de endereçamento postal")));
        c[i].setNomeRua(JOptionPane.showInputDialog("Digite o nome da rua"));
        c[i].setBairro(JOptionPane.showInputDialog("Digite o bairro"));
    }
    Funcionario f[]=new Funcionario[5];

    for (int i=0; i<5; i++)
    {
        f[i]= new Funcionario();
        f[i].setNome(JOptionPane.showInputDialog("Digite o nome do funcionário"));
        f[i].setEnd(JOptionPane.showInputDialog("Digite o endereço do funcionário"));
        f[i].setSalario(Double.parseDouble(JOptionPane.showInputDialog("Digite o
salario do funcionário")));
        String digCep=JOptionPane.showInputDialog("Digite o cep do funcionário");
        for (int j=0; j<3; j++)
        {
            if (digCep.equalsIgnoreCase(c[j].getCodigo() ))
            {
                f[i].setCep(c[j]);
                break;
            }
        }
    }
    for (int i=0; i<5; i++)
    {
        JOptionPane.showMessageDialog(null, "Nome:"+" "+f[i].getNome()+"\n"
n"+"Rua:"+" "+f[i].getCep().getNomeRua()+"\n"+"Bairro:"+" "+f[i].getCep().getBairro());
    }
    System.exit(0);
}
}

```

Unidade 12

1) A resposta é a implementação (programação) do código da unidade.

2) Desenvolva um sistema para armazenar informações sobre professores e disciplinas de instituição educacional.

Resposta:

//Implementação da classe Professor

```
public class Professor{
    private String nome;
    private String titumax;
    private int ch;

    public Professor ( ){
        nome= "";
        titumax= "";
        ch= 0;
    }
    public void setnome (String snome){
        nome=snome;
    }
    public void settitumax (String stitumax){
        titumax=stitumax;
    }
    public void setch (int ich){
        ch=ich;
    }
    public String getnome (){
        return nome;
    }
    public String gettitumax (){
        return titumax;
    }
    public int getch (){
        return ch;
    }
}
```

//Implementação da Classe Disciplina

```
public class Disciplina{
    private String nome;
    private int ch;
    private Professor prof;

    public Disciplina ( ){
        nome= "";
        ch= 0;
        prof=null;
    }
}
```

```

    public void setnomedisc (String snome){
        nome=snome;
    }
    public void setchdisc (int ich){
        ch=ich;
    }
    public void setProf (Professor p){
        prof = p;
    }
    public String getnomedisc (){
        return nome;
    }
    public int getchdisc (){
        return ch;
    }
    public Professor getProf (){
        return prof;
    }
}

```

//Implementação de um programa simples para cadastrar Disciplina e Professor

```

import javax.swing.*;
public class CadastroDisciplinaProf
{
    public static void main(String args[])
    {
        //opcao 1 - cadastro de disciplinas e professores

        //cria todos os objetos Professor
        Professor p[]=new Professor[5];

        for (int i=0; i<5; i++)
        {
            p[i]= new Professor();
            p[i].setnome(JOptionPane.showInputDialog("Digite o nome do professor"));
            p[i].settitumax(JOptionPane.showInputDialog("Digite a titulação máxima do prof."));
            p[i].setch(Integer.parseInt(JOptionPane.showInputDialog("Digite a carga horaria")));
        }

        /*cadastra três disciplina e associa cada uma a um professor, porque uma disciplina só pode ser
        ministrada por um professor*/

        Disciplina d[]=new Disciplina[3];

        for (int i=0; i<3; i++)
        {
            d[i]=new Disciplina();
            d[i].setnomedisc(JOptionPane.showInputDialog("Digite o código de endereçamento postal"));

```

```

        d[i].setchdisc(Integer.parseInt(JOptionPane.showInputDialog("Digite o nome da rua")));

        /* a referência para o objeto professor que ministra a disciplina que está sendo cadastrada deve ser
        procurada no vetor p[] através do nome do professor digitado pelo usuário*/

        String nomeprof=JOptionPane.showInputDialog("Digite o nome do professor");

        for (int j=0; j<5; j++){
            if (nomeprof.equalsIgnoreCase(p[j].getnome())){
                d[i].setProf(p[j]);
                break;
            }
        }
    }

    //opcao 2 - entra com o nome da disciplina e imprime o nome do professor que ministra disciplina

    String nomedisc=JOptionPane.showInputDialog("Digite o nome da disciplina");
    for (int i=0; i<3; i++)
    {
        if (nomedisc.equalsIgnoreCase(d[i].getnomedisc())) {
            JOptionPane.showMessageDialog(null, "Nome do professor: " + d[i].getProf().getnome());
            break;
        }
    }

    //opcao 3 - entra com o nome da titulacao do professor e imprime o nome de todos os profes com essa titulação

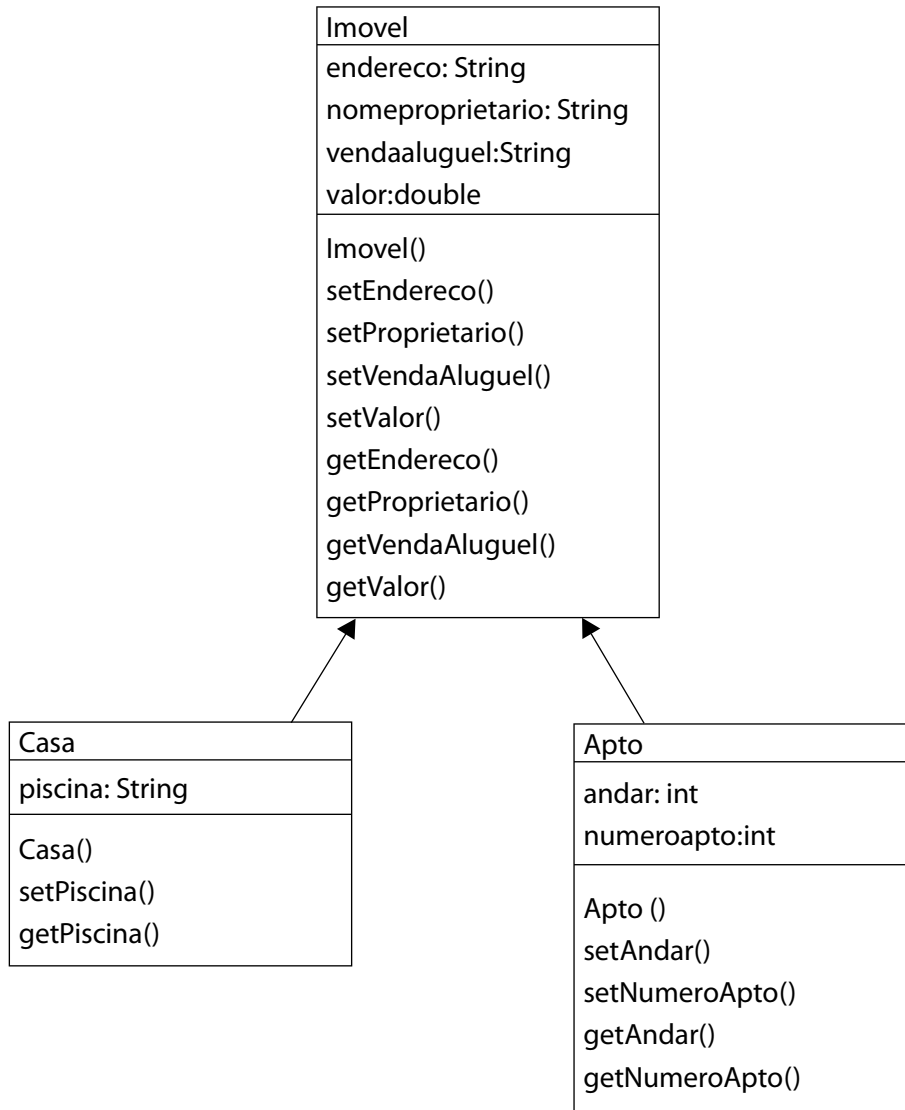
    String titulacao=JOptionPane.showInputDialog("Digite a titulação do professor");
    for (int i=0; i<5; i++)
    {
        if (titulacao.equalsIgnoreCase(p[i].gettitumax())) {
            JOptionPane.showMessageDialog(null, "Nome do professor: " + p[i].getnome());
        }
    }
    System.exit(0);
}
}

```

Unidade 13

1) Desenvolva um sistema orientado a objetos para automatizar as informações de uma Administradora de Imóveis.

Resposta:



Unidade 14

1) Implemente as classes modeladas no sistema da Administradora de Imóveis descrito na atividade da unidade anterior.

Resposta:

//Implementação da classe Imóvel

```
public class Imovel {

    private String nomeproprietario;
    private String endereco;
    private String vendaaluguel;
    private double valor;

    public Imovel (){
        nomeproprietario="";
        endereco="";
        vendaaluguel="";
        valor = 0;
    }

    public void setNomeProprietario ( String snome){
        nomeproprietario=snome;
    }

    public void setEndereco ( String sendereco){
        endereco=sendereco;
    }

    public void setVendaAluguel (String stipo){
        vendaaluguel=stipo;
    }

    public void setValor ( double dvalor){
        valor=dvalor;
    }

    public String getNomeProprietario(){
        return nomeproprietario ;
    }

    public String getEndereço(){
        return endereco;
    }

    public String getVendaAluguel(){
        return vendaaluguel;
    }

    public double getValor(){
        return valor;
    }

}
```

// Implementação da classe Apto


```
public class Apto extends Imovel{
    private int andar;
    private int numero;

    public Apto (){
        super ();
        andar=0;
        numero=0;
    }
    public void setAndar (int iandar){
        andar=iandar;
    }
    public void setNumero ( int inumero){
        numero=inumero;
    }
    public int getAndar (){
        return andar;
    }
    public int getNumero (){
        return numero;
    }
}
```

//Implementação Casa

```
public class Casa extends Imovel{
    private String piscina;

    public Casa (){
        super ();
        piscina="nao";
    }
    public void setPiscina (String spiscina){
        piscina=spiscina;
    }
    public String getPiscina (){
        return piscina;
    }
}
```

2) Implementa as classes e o sistema da Locadora de Veículos descrito nessa unidade.

Resposta: código já está na unidade.

