UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
Departamento de Ciência da Computação

Introdução à Ciência da Computação com Java e Orientação a Objetos

1ª edição

Alfredo Goldman
Fabio Kon
Paulo J. S. Silva

Editado e Revisado por:
Raphael Y. de Camargo

A nossas esposas e filhos,
fonte de força e alegria em nossas vidas.

Agradecimentos

Este livro não seria possível sem a colaboração de inúmeros alunos e professores do IME/USP.

Leo Kazuhiro Ueda e Nelson Posse Lago atuaram como assistentes de ensino na primeira vez em que esta disciplina foi ministrada e foram responsáveis por inúmeras contribuições. O apêndice sobre o Dr. Java foi preparado pelo Leo. Fabiano Mitsuo Sato, George Henrique Silva e Igor Ribeiro Sucupira foram monitores da disciplina também em 2003 e colaboraram com alguns exercícios.

Raphael Y. de Camargo realizou um excelente trabalho na edição e revisão do livro além de colaborar com alguns exercícios. O Prof. João Eduardo Ferreira, nosso colega no ensino da disciplina de introdução, o Prof. Valdemar Setzer, nosso experiente e sábio colega de departamento, e o Prof. Marcos Chaim, da USPLeste, nos deram inúmeras sugestões úteis que, sempre que possível, foram incorporadas ao texto final.

Agradecemos ao Prof. Walter Savitch da Universidade da California em San Diego por ter autorizado o uso de sua classe para entrada de dados.

Finalmente, agradecemos aos alunos e professores que não foram citados mas que deram sugestões e nos incentivaram a escrever este livro.

Sumário

Lista de Figuras

Prefácio

Caros leitores, sejam bem-vindos ao maravilhoso mundo da Ciência da Computação. Com este livro, vocês terão a oportunidade de aprender os elementos básicos da programação de computadores e terão contato com alguns dos conceitos fundamentais da Ciência da Computação.

Este livro é resultado da experiência dos autores na disciplina de Introdução à Ciência da Computação ministrada no IME/USP utilizando orientação a objetos e a linguagem Java de 2003 a 2006. Desde 2005, esta abordagem também é aplicada na USPLeste com os 180 alunos do curso de Sistemas de Informação. Influenciada pelo método proposto pela ACM (*Association for Computing Machinery*) e pelo IEEE (*Institute of Electrical and Electronics Engineers*) em seu currículo *Objects-first*, nossa abordagem evita o problema de se ensinar a programar sem o uso de objetos para depois exigir do estudante uma mudança de paradigma com a introdução de objetos. Elimina-se assim a necessidade de dizer "esqueça a forma que programamos até agora; agora vamos aprender o jeito correto".

Apresentando os conceitos de orientação a objetos desde o início do ensino de programação, a abordagem adotada aqui permite que o aluno inicie sua formação em programação de forma equilibrada. Aprendendo conceitos tanto algorítmicos quanto estruturais ao mesmo tempo em que toma contato com práticas fundamentais de programação como testes automatizados e o uso de nomes claros para os elementos do código, o leitor obtém uma visão básica porém ampla do que é mais importante para o desenvolvimento de software.

Esperamos que este livro seja útil tanto para alunos de cursos superiores de Informática quanto para profissionais do mercado e outros curiosos que queiram adquirir conhecimentos na área de desenvolvimento de software. Nosso objetivo é que este livro os ajude na fascinante empreitada de aprender a programar de forma elegante e eficaz e que ele permita um bom início rumo a uma formação sólida em programação que é uma condição fundamental que os desafios da informática do século XXI nos impõem.

Abraços e boa jornada!

São Paulo, março de 2006.

Alfredo Goldman, Fabio Kon e Paulo J. S. Silva

Capítulo 1

Teatro de Objetos

Vamos iniciar a nossa jornada ao fascinante mundo da orientação a objetos de forma dramática: com um Teatro de Objetos. Se você está usando este livro em um curso, reserve juntamente com seu professor uma parte da primeira aula para realizar a encenação descrita neste capítulo. Se você trabalha em uma empresa, reúna-se com colegas de trabalho ou com amigos interessados em programação para exercitar suas qualidades dramáticas. Liberte o artista que existe dentro de você!'

O objetivo do Teatro de Objetos é fazer com que os alunos vivenciem um jogo interativo do qual participam vários "objetos" realizando diferentes formas de ações e comunicações. Os conceitos de orientação a objetos empregados no teatro não são explicitamente explicados já no primeiro capítulo mas serão abordados ao longo do livro. Em um curso de Introdução á Ciência da Computação, normalmente a primeira metade da primeira aula é dedicada a uma conversa informal com os alunos explicando quais são os objetivos da disciplina (e do curso inteiro, se for o caso). É sempre interessante também conversar sobre contatos prévios que os alunos tiveram com informática e com programação. É bom deixar claro que este livro pode ser acompanhado por uma pessoa que nunca viu um computador na frente em sua vida, mas que aprender a programar não é uma tarefa fácil, é preciso se empenhar. Na segunda metade da aula, exercitamos as habilidades dramáticas: para ilustrar o funcionamento de um programa de computador complexo, vamos fazer de conta que somos partes de um programa de computador trabalhando em conjunto para atingir um certo objetivo. Se você estiver organizando a encenação com seus colegas, você pode fazer um certo suspense sobre qual é o objetivo (simular uma disputa de pênaltis) e sobre como ele será alcançado.

1.1 Disputa de pênaltis

A "peça" que encenaremos representará uma disputa de pênaltis entre dois times e contará com a participação de cerca de 26 atores desempenhando 8 papéis. Se você não tiver 26 atores disponíveis, você pode elaborar a sua própria adaptação da peça. Os papéis são:

- Técnico (2 atores)
- Juiz (1 ator)
- Bandeirinha (2 atores)
- Gandula (1 ator)

- Jogador. Os jogadores são divididos em dois tipos:
  - Goleiro (2 atores desempenham este papel)
  - Batedor de pênalti (10 atores)
- Torcedor. Os torcedores são divididos em dois tipos:
  - Torcedor educado (4 atores)
  - Torcedor mal-educado (4 atores)

O professor (ou o diretor da peça, que pode ser você) será responsável por escolher as pessoas que desempenharão cada papel. Se houver limitação de espaço, é conveniente que os 8 torcedores fiquem concentrados em uma área separada (por exemplo, em uma suas próprias carteiras, se o local for uma sala de aula) para não tumultuar muito o ambiente. Obviamente, o diretor pode também aumentar ou diminuir o número de torcedores e de batedores de pênalti. Para desempenhar o papel de torcedores, uma boa dica é o diretor escolher pessoas que pareçam bem barulhentas e faladoras (por exemplo, a turma do fundão numa sala de aula :-). Ao escolher os atores, o diretor deverá entregar um cartão preso com um barbante que ficará pendurado no pescoço do ator e conterá informações sobre o papel desempenhado pelo ator. As informações são:

1. Nome do papel
2. Mensagens que o personagem é capaz de entender
3. Atributos do personagem

Os três tipos de informação acima já devem vir pré-escritos à caneta no cartão mas os valores dos atributos do personagem devem ser escritos na hora a lápis pelo diretor. Alguns papéis, como o de juiz, não possuem nenhum atributo. Outros papéis podem possuir um ou mais atributos, o jogador, por exemplo, pode possuir como atributos o nome do time ao qual pertence e o número da sua camisa. No caso de o jogador ser um goleiro, o atributo "número da camisa" pode vir escrito a caneta como valendo 1.

Além do cartão que fica pendurado no pescoço do ator, cada ator recebe um script descrevendo o seu comportamento: para cada mensagem recebida pelo ator, o script descreve quais ações devem ser tomadas pelo ator.

O diretor não deve esquecer de trazer uma bola para esta atividade e deve tomar cuidado para que nenhuma janela seja quebrada durante a realização da atividade. O tempo total estimado para a realização da atividade é de 50 minutos. A maior parte do tempo é gasto explicando-se os procedimentos. A encenação em si, demora entre 5 e 10 minutos dependendo da qualidade dos batedores de pênalti e dos goleiros.

Eis a descrição detalhada dos dados que deverão aparecer nos cartões descritivos e no script (comportamento) de cada um dos 26 atores participantes da encenação. Para obter versões PDF dos cartões prontas para impressão, visite o sítio deste livro: `www.ime.usp.br/~kon/livros/Java`

1. Goleiro

   - Cartão de Identificação
     - Nome do Papel: Goleiro
     - Mensagens que entende: `SuaVez`, `Cobrança Autorizada`, `VenceuOTimeX`
     - Atributos: `Time:` , `Camisa número:` 1
   - Comportamento (*Script*)
     - mensagem: `SuaVez` ⇒ ação: posiciona-se na frente do gol e fica esperando pela cobrança do pênalti.
     - mensagem: `CobrançaAutorizada` ⇒ ação: concentra-se na bola que será chutada pelo adversário e faz de tudo para não deixar que a bola entre no gol. O goleiro não pode se adiantar antes do chute do adversário. Após a cobrança sair do gol para dar lugar ao goleiro adversário.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` é igual ao atributo `Time` no seu cartão de identificação, comemore; caso contrário, xingue o juiz (polidamente! :-).

2. Batedor de Pênalti

   - Cartão de Identificação
     - Nome do Papel: Batedor de Pênalti
     - Mensagens que entende: `SuaVez`, `CobrançaAutorizada`, `VenceuOTimeX`
     - Atributos: `Time:` , `Camisa número:`
   - Comportamento
     - mensagem: `SuaVez` ⇒ ação: posiciona-se na frente da bola e fica esperando pela autorização do juiz.
     - mensagem: `CobrançaAutorizada` ⇒ ação: chuta a bola tentando marcar um gol. Após a cobrança voltar para junto do seu técnico para dar lugar à próxima cobrança.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` é igual ao atributo `Time` no seu cartão de identificação, comemore; caso contrário, xingue o juiz (polidamente! :-).

3. Torcedor Educado

   - Cartão de Identificação
     - Nome do Papel: Torcedor Educado
     - Mensagens que entende: `Ação`, `VenceuOTimeX`
     - Atributos: `Time:` , `Camisa número:` 12
   - Comportamento
     - mensagem: `Ação` ⇒ ação: assista ao jogo emitindo opiniões inteligentes sobre o andamento da peleja e manifestando o seu apreço e desapreço pelo desenrolar da disputa.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` é igual ao atributo `Time` no seu cartão de identificação, comemore e faça um comentário elogioso sobre o seu time; caso contrário, elogie o adversário e parabenize o seu time pelo empenho.

4. Torcedor Mal-Educado

   - Cartão de Identificação
     - Nome do Papel: Torcedor Mal-Educado
     - Mensagens que entende: `Ação`, `VenceuOTimeX`
     - Atributos: `Time:` , `Camisa número: 12`
   - Comportamento
     - mensagem: `Ação` ⇒ ação: assista ao jogo emitindo opiniões duvidosas sobre o andamento da peleja e manifestando a sua raiva ou alegria pelo desenrolar do jogo.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` é igual ao atributo `Time` no seu cartão de identificação, xingue o adversário. Caso contrário, xingue o adversário desesperadamente (mas, por favor, não se esqueça que estamos em uma universidade).

5. Juiz

   - Cartão de Identificação
     - Nome do Papel: Juiz
     - Mensagens que entende: `Ação`, `Irregularidade`
   - Comportamento
     - mensagem: `Ação` ⇒ ação: coordene o andamento da disputa de pênaltis enviando mensagens `SuaVez` para o técnico do time batedor e para o goleiro defensor a cada nova batida. Quando os personagens estiverem a postos, emita a mensagem `CobrançaAutorizada`. Faça a contagem de gols e quando houver um vencedor, emita a mensagem `VenceuOTimeX` onde `TimeX` é o nome do time vencedor. A disputa de pênaltis é feita alternadamente, 5 cobranças para cada time. Se não houver um ganhador após as 5 cobranças, são feitas novas cobranças alternadamente até que haja um vencedor.
     - mensagem: `Irregularidade` ⇒ ação: se a mensagem foi enviada por um Bandeirinha, ignore a cobrança recém-efetuada e ordene que ela seja realizada novamente enviando a mensagem `RepitaCobrança` ao técnico apropriado.

6. Gandula

   - Cartão de Identificação
     - Nome do Papel: Gandula
     - Mensagens que entende: `CobrançaAutorizada`
   - Comportamento
     - mensagem: `CobrançaAutorizada` ⇒ ação: preste atenção à cobrança do pênalti. Após a conclusão da cobrança, pegue a bola e leve-a de volta à marca de pênalti.

7. Técnico

   - Cartão de Identificação
     - Nome do Papel: Técnico
     - Mensagens que entende: `SuaVez`, `RepitaCobrança`, `VenceuOTimeX`
     - Atributos: `Time:`
   - Comportamento
     - mensagem: `SuaVez` ⇒ ação: escolha um dos seus jogadores para efetuar a cobrança e envie a mensagem `SuaVez`. Não repita jogadores nas 5 cobranças iniciais.
     - mensagem: `RepitaCobrança` ⇒ ação: envie a mensagem `SuaVez` para o jogador que acabou de efetuar a cobrança.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` é igual ao atributo `Time` no seu cartão de identificação, comemore; caso contrário, diga que o seu time foi prejudicado pela arbitragem e que futebol é uma caixinha de surpresas.

8. Bandeirinha

   - Cartão de Identificação
     - Nome do Papel: Bandeirinha
     - Mensagens que entende: `Cobrança Autorizada`, `VenceuOTimeX`
   - Comportamento
     - mensagem: `CobrançaAutorizada` ⇒ ação: verifique se o goleiro realmente não avança antes de o batedor chutar a bola. Caso ele avance, envie uma mensagem `Irregularidade` para o Juiz.
     - mensagem: `VenceuOTimeX` ⇒ ação: se `TimeX` não é o nome do time da casa, distancie-se da torcida pois você acaba de se tornar um alvo em potencial.

Capítulo 2

História da Computação

Quais novidades veremos neste capítulo?

- a história da computação;
- evolução da arquitetura do computador;
- evolução das linguagens de programação.

A Computação tem sido considerada uma ciência independente desde meados do século XX. No entanto, desde a antiguidade, a humanidade busca formas de automatizar os seus cômputos e de criar máquinas e métodos para facilitar a realização de cálculos. Neste capítulo, apresentamos inicialmente uma linha do tempo que indica alguns dos principais eventos ocorridos na história da computação e na evolução da arquitetura dos computadores automáticos. Em seguida, apresentamos uma linha do tempo da evolução das linguagens de programação.

2.1 História da Computação e Arquitetura do Computador

- Ábaco (Soroban em japonês) (criado ~2000 anos atrás)
- Blaise Pascal, 1642 (*pai da calculadora*)
  - o primeiro computador digital
  - hoje, diríamos apenas que era uma calculadora super simplificada
  - capaz de somar
  - entrada através de discos giratórios
  - ajudou seu pai, coletor de impostos
- Leibniz
  - computador capaz de somar e multiplicar (inventou em 1671 e construiu em 1694)

  - criou o mecanismo de engrenagens do "vai-um" usado até hoje

- Joseph Marie Jacquard (1752 - 1834)
  - 1790: criou um sistema de tear semi-automático onde os desenhos – de flores, folhas e figuras geométricas – eram codificados em cartões perfurados
  - 1812: havia cerca de 11 mil teares de Jacquard na França
  - a máquina despertou muitos protestos de artesãos que temiam o desemprego que a máquina poderia causar
  - a máquina era capaz de desenhar padrões de alta complexidade como, por exemplo, um auto-retrato de Jacquard feito com 10 mil cartões perfurados

- Charles Babbage (professor de Matemática em Cambridge, Inglaterra)
  - 1812: notou que muito do que se fazia em Matemática poderia ser automatizado
  - iniciou projeto do "Difference Engine" (Máquina/Engenho/Engenhoca de Diferenças)
  - 1822: terminou um protótipo da máquina e obteve financiamento do governo para construí-la
  - 1823: iniciou a construção (usaria motor a vapor, seria totalmente automático, imprimiria o resultado e teria um programa fixo)
  - 1833: depois de 10 anos teve uma idéia melhor e abandonou tudo
  - Nova idéia: máquina **programável**, de propósito geral: "Analytical Engine" (Máquina Analítica)
    * manipularia números de 50 dígitos
    * memória de 1000 dígitos
    * *estações de leitura* leriam cartões perfurados similares ao de tear de Jacquard (nesta época, o auto-retrato de Jacquard pertencia a Babbage)
  - mas ele não conseguiu construí-lo
    * tecnologia mecânica da época era insuficiente
    * pouca gente via a necessidade para tal máquina
  - Ada Lovelace (*mãe da programação*) escreveu programas para o engenho analítico; inventou a palavra algoritmo em homenagem ao matemático Al-Khawarizmi (820 d.C.)
  - **Algoritmo**: seqüência de operações ou comandos que, aplicada a um conjunto de dados, permite solucionar classes de problemas semelhantes. Exemplos: algoritmo da divisão, da raiz cúbica, para resolução de equações do segundo grau, etc.
  - a máquina foi finalmente construída pelo governo inglês nos anos 1990 (e funciona!)

- Herman Hollerith, 1890
  - criou cartões perfurados para uso no censo americano
  - tecnologia levou à criação da International Business Machines (IBM)
  - até hoje dizemos que no final do mês os empregados recebem o “holerite” como sinônimo de contracheque

- Avanços nas calculadoras de mesa ⇒ Em 1890, as máquinas permitiam:
  - acumular resultados parciais
  - armazenamento e reentrada automática de resultados passados (memória)
  - imprimir resultados em papel
- MARK 1, criada em 1937 por Howard Aiken, professor de Matemática Aplicada de Harvard
  - calculadora eletromecânica com motor elétrico
  - pesava 5 toneladas, usava toneladas de gelo para refrigeração
  - multiplicava dois números de 23 dígitos em 3 segundos
- John Vincent Atanasoff, Universidade Estadual de Iowa
  - construiu o que é considerado o primeiro computador digital entre 1937 e 1942
  - calculadora com válvulas a vácuo (240 válvulas)
  - resolvia equações lineares, diferenciais e de balística
  - manipulava números binários
- Rumo à programabilidade
- Alan Turing,
  - Trabalhou para o exército inglês ajudando a quebrar o código criptográfico da máquina Enigma criada pelos alemães
  - Realizou importantes contribuições práticas e teóricas à Ciência da Computação
  - 1912: nasce em Londres
  - 1935: Ganha bolsa para realizar pesquisas no King's College, Cambridge
  - 1936: Elabora "Máquina de Turing", pesquisas em computabilidade
  - 1936-38: Princeton University. Ph.D. Lógica, Álgebra, Teoria dos Números
  - 1938-39: Cambridge. É apresentado à máquina Enigma dos alemães
  - 1939-40: "The Bombe", máquina para decodificação do Enigma criada em Bletchley Park
  - 1939-42: "quebra" Enigma do U-boat, aliados vencem batalha do Atlântico
  - 1943-45: Consultor-chefe anglo-americano para criptologia
  - 1947-48: Programação, redes neurais e inteligência artificial
  - 1948: Manchester University
  - 1949: Pesquisas sobre usos do computador em cálculos matemáticos avançados
  - 1950: Propõe "Teste de Turing" para inteligência de máquinas
  - 1952: Preso por homossexualidade, perde privilégios militares
  - 1953-54: Trabalho não finalizado em Biologia e Física; tem sua reputação e vida destruídas pelos militares ingleses

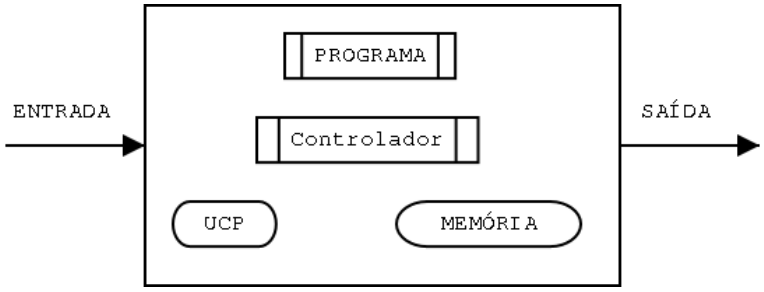


Figura 2.1: Arquitetura do ENIAC

  - 1954: Suicida-se em Wilmslow, Cheshire
  - Livro interessante sobre sua vida e obra: *Alan Turing: the Enigma* de Andrew Hodges, 2000
  - Sítio sobre a vida de Turing mantido pelo autor deste livro: `http://www.turing.org.uk/turing`

- ENIAC (Electronic Numerical Integrator and Computer), 1945

  - por alguns, considerado o primeiro computador eletrônico
  - números de 10 dígitos decimais
  - 300 multiplicações ou 5000 somas por segundo
  - 17486 válvulas, a queima de válvulas era quase que diária
  - 6000 comutadores manuais e centenas de cabos usados na programação
  - programação era muito difícil

- Arquitetura do ENIAC (ver Figura 2.1)

  - programa especificado manualmente em "hardware" com conexões semelhantes àquelas que as velhas telefonistas utilizavam
  - memória de dados separada do controle e separada do programa
  - o controle é formado por circuitos eletroeletrônicos

- John Von Neumann, matemático, 1945

  - estudo abstrato de modelos de computação levou à arquitetura do computador moderno
  - o programa deve ser guardado no mesmo lugar que os dados: na memória
  - Arquitetura de Von Neumann (ver Figura 2.2)
  - hoje em dia, há vários tipos de memória ROM, RAM, flash RAM, etc.)
  - o controlador em memória, levou à idéia de sistema operacional que temos hoje

  - Nunca diga nunca...

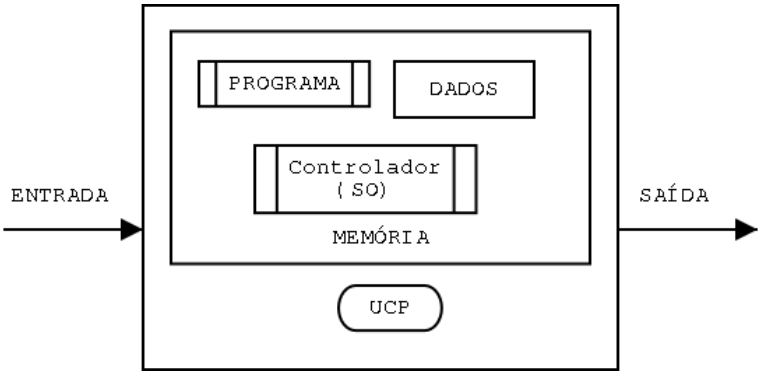


Figura 2.2: Arquitetura de Von Neumann

  * Aiken declarou em 1947 que nunca haveria necessidade de mais de um ou dois computadores programáveis e que os projetos neste sentido deveriam ser abandonados
  * Bill Gates declarou na década de 1980 que a Microsoft nunca desenvolveria um sistema operacional de 32 bits

- Anos 50
  - 1953: IBM vende 15 máquinas baseadas no modelo de Neumann
  - transistores
  - memória magnética ("magnetic core memory")
- Anos 60
  - circuitos impressos / circuitos integrados (*chips*)
  - crescimento segundo lei de Moore, que diz que o número de transistores em circuitos integrados duplica a cada 18 meses. Esta lei continua valendo até hoje
  - computação limitada a poucos computadores de grande porte
- Anos 70
  - indo contra o modelo centralizador da IBM, geração sexo, drogas e rock-and-roll da California exige a democratização da informática
  - revista esquerdista da Universidade da Califórnia em Berkeley *People's Computer Company* defende a criação de computadores pessoais e de cooperativas de informação
  - Steve Jobs cria a Apple em garagem em ~1975 e investe lucros do Apple II em shows de Rock (82)
  - nasce a Microsoft
  - governo da Califórnia apóia microinformática

- Anos 80
  - IBM lança PC (1981)
  - Apple lança MacIntosh (1984)
  - Xerox inventa e Apple comercializa interface baseada em janelas ("Windows")
  - Microsoft cresce comercializando o sistema operacional simplista MS-DOS para IBM-PCs (o DOS era uma versão simplificada do CPM que, por sua vez, era uma versão simplificada do UNIX)
  - algumas empresas começam a esconder código-fonte do software (antes era sempre aberto)
  - começa o movimento do software livre e software aberto
- Anos 90
  - Microsoft pega carona na explosão de vendas de PCs, utiliza técnicas de marketing agressivas (consideradas por alguns como moralmente questionáveis) para controlar o mercado de software, estabelecendo quase um monopólio em certas áreas
  - popularização da Internet e criação da Web
  - intensifica-se o movimento do software livre
  - nasce o Linux e uma nova forma de desenvolvimento de software baseada em comunidades distribuídas através da Internet
  - no final da década, o governo americano percebe o perigo que a Microsoft representa e inicia batalha judicial contra a empresa (em 2002, após a vitória dos conservadores na eleição nos EUA, o governo termina um processo judicial que havia iniciado; as sanções à empresa são mínimas)
- Século XXI
  - computadores de mão, telefones celulares, iPod
  - sistemas embutidos
  - computação ubíqua e pervasiva
  - grande crescimento da empresa Google através de serviços inovadores na Web

2.2 Evolução das Linguagens de Programação

Paralelamente à evolução do hardware dos computadores eletrônicos, ocorreu também a evolução do software e das linguagens de programação utilizadas para desenvolvê-lo. Inicialmente, as linguagens estavam bem próximas do funcionamento dos circuitos do hardware; paulatinamente, as linguagens foram se aproximando da linguagem natural utilizada pelos humanos em seu dia-a-dia.

- A máquina de Babbage só poderia ser programada com a troca física de engrenagens
- 1945, no ENIAC, a programação era feita mudando chaves e trocando a posição de cabos

- 1949-50, primeira linguagem binária, a programação era feita mudando os "comandos" de zero a um, e vice-versa
- 1951, Grace Hooper cria o primeiro compilador, A0, programa que transforma comandos em 0s e 1s
- 1957, primeira linguagem de programação de alto nível: FORTRAN (*FORmula TRANslating*) (John Backus da IBM)
- 1958, criação de um padrão universal de linguagem: ALGOL 58 (*ALGOritmic Language*) (origem da maioria das linguagens modernas). Primeira linguagem estruturada
- 1958, John McCarthy do MIT cria o LISP (*LISt Processing*), inicialmente projetada para uso em inteligência artificial. Nela tudo se baseia em listas. Ainda é usada hoje em dia
- 1959, FORTRAN era eficaz para manipulação de números, mas não para entrada e saída: foi criada COBOL (*COmmon Bussines Oriented Language*)
- 1964, criação do BASIC (*Beginners All-purpose Symbolic Instruction Code*)
- 1965, criação de uma linguagem específica para simulação (SIMULA I) por Ole-Johan Dahl and Kristen Nygaard da Universidade de Oslo. É considerada a base das linguagens orientadas a objetos
- 1966, criação da linguagem Logo para desenhos gráficos (a linguagem da tartaruga)
- 1967, Simula 67, uma linguagem de uso geral incluindo todos os conceitos fundamentais de orientação a objetos
- 1968, criação da linguagem PASCAL por Niklaus Wirth. Principal interesse: linguagem para o ensino. Combinou as melhores características de Cobol, Fortran e Algol, foi uma linguagem bem utilizada
- 1970, PROLOG, linguagem para programação lógica
- 1972, criação da linguagem C (Denis Ritchie). Supriu as deficiências da linguagem Pascal e teve sucesso quase imediato
- 1972-1980, linguagem Smalltalk, desenvolvida por Alan Kay, da Xerox, utilizando as melhores características de LISP, Simula 67, Logo e ambientes gráficos como Sketchpad; Orientação a Objetos ganha força
- 1983, criadas extensões de C incluindo suporte para OO: C++ e Objective-C
- 1987, linguagens baseadas em scripts, como Perl, desenvolvida por Larry Wall. Ferramentas de UNIX como sed e awk não eram suficientes
- 1994, Java é lançada pela Sun como a linguagem para a Internet
- 2000, lançamento do C# pela Microsoft, combinando idéias das linguagens Java e C++
- 2005, lançamento da versão 1.5 de Java

Neste livro, apresentamos, de forma muito sucinta, algumas das principais linguagens de programação. Para referências mais completas visite `www.princeton.edu/~ferguson/adw/programming_languages.shtml` e `en.wikipedia.org/wiki/Programming_language`.

Capítulo 3

Conversor de Temperaturas

Quais novidades veremos neste capítulo?

- primeiro programa em Java.

3.1 Analogia entre dramatização da disputa de pênaltis e Programação Orientada a Objetos

Terminologia Teatral	Terminologia de Programação Orientada a Objetos
personagem (papel)	classe (tipo)
ator	objeto
envio de mensagem	envio de mensagem, chamada de método ou chamada de função

Na dramatização, podíamos enviar uma mensagem (dizer alguma coisa) para um ator. Na programação orientada a objetos, podemos enviar uma mensagem para (ou chamar um método de) um objeto.

Os cartões de identificação definem os papéis dos atores e os scripts especificam o comportamento dos atores no decorrer da peça. A linguagem Java permite que especifiquemos a mesma coisa. Um cartão de identificação tem 3 partes, essas mesmas 3 partes aparecem na definição de uma classe em Java. Por exemplo, o bandeirinha em Java seria mais ou menos assim:

```
class Bandeirinha
{
  CobrançaAutorizada
  {
    // Verifica se o goleiro realmente não avança antes de o batedor...
  }
  VenceuOTime (NomeDoTime)
  {
    // Se NomeDoTime não é o nome do time da casa, distancie-se da torcida...
  }
}
```

3.2 Um exemplo real em Java: um conversor de Celsius para Fahrenheit

Sempre o primeiro passo antes de programar é analisar o problema.

$$\frac{F-32}{9} = \frac{C}{5} \Rightarrow F - 32 = \frac{9}{5}C \Rightarrow F = \frac{9}{5}C + 32$$

Traduzindo esta fórmula para Java temos `F=9*C/5+32`. A seguir iremos criar diversas classes para realizar a conversão entre Celsius e Fahrenheit.

1. **Primeira tentativa:** programa em Java para converter 40 graus Celsius para Fahrenheit.

```
class Conversor
{
  int celsiusParaFahrenheit()
  {
    return 9 * 40 / 5 + 32;
  }
}
```

   - para executar este conversor dentro do DrJava[1], podemos digitar o seguinte na janela do interpretador (chamada de *interactions*):

```
Conversor c1 = new Conversor();
c1.celsiusParaFahrenheit()
```

     o DrJava imprimirá o valor devolvido pelo método `celsiusParaFahrenheit` do objeto `c1`.

   - limitação: sempre converte a mesma temperatura.

2. **Segunda tentativa:** conversor genérico de temperaturas Celsius -> Fahrenheit. É capaz de converter qualquer temperatura de Fahrenheit para Celsius.

```
class Conversor2
{
  int celsiusParaFahrenheit(int c)
  {
    return 9 * c / 5 + 32;
  }
}
```

   - para executar este conversor, podemos digitar o seguinte na janela do interpretador:

```
Conversor2 c2 = new Conversor2();
c2.celsiusParaFahrenheit(100)
```

     o DrJava imprimirá o valor devolvido pelo método `celsiusParaFahrenheit` do objeto `c2`.

[1] Para mais informações sobre o DrJava consulte o Apêndice A.

- limitação: essa classe manipula apenas números inteiros. Mas, em geral, temperaturas são números reais, fracionários, então números inteiros não são suficientes. Quando o computador opera com números inteiros, os números são truncados, ou seja, 30.3 se torna 30 e 30.9 também se torna 30. Devido a esta limitação, se tivéssemos escrito a fórmula como $\frac{9}{5} * C + 32$, o cálculo seria errado, uma vez que $\frac{9}{5} = 1$ se considerarmos apenas a parte inteira da divisão. Assim, o programa calcularia apenas $1 * C + 32$.

Quando precisamos trabalhar com números reais, usamos números de ponto flutuante (*floating point numbers*). Esse nome se deriva do fato de que internamente os números são representados de forma parecida a potências de 10 onde, variando o expoente da potência, movemos (flutuamos) o ponto decimal para a esquerda ou para a direita. De *floating point* vem o tipo `float` da linguagem Java. Números do tipo `float` são armazenados em 4 bytes e tem uma precisão de 23 bits (o que equivale a aproximadamente 7 casas decimais). No entanto, quase sempre são utilizados números de ponto flutuante com precisão dupla que são chamados de `double`. Em Java, um `double` ocupa 8 bytes e tem precisão de 52 bits, o que equivale a aproximadamente 16 casas decimais. Daqui para frente sempre que precisarmos de números fracionários, vamos utilizar o tipo `double`. Mas lembre-se de que as variáveis do tipo `double` são apenas uma aproximação do número real, como podemos ver no exemplo abaixo:

```
> double x = 1.0
1.0
> x = x / 59049
1.6935087808430286E-5
> x = x * 59049
0.9999999999999999
```

Após dividir o número 1.0 pelo número $3^{10} = 59049$ e depois multiplicá-lo por este mesmo número, obtivemos um valor diferente do inicial. Por este motivo, devemos ter muito cuidado com os erros de arredondamento presentes quando utilizamos números de ponto flutuante.

3. **Terceira tentativa:** conversor genérico usando `double`

```
class Conversor3
{
  double celsiusParaFahrenheit(double c)
  {
    return 9.0 * c / 5.0 + 32.0;
  }
}
```

- para executar este conversor, podemos digitar o seguinte na janela do interpretador:

```
Conversor3 c3 = new Conversor3();
c3.celsiusParaFahrenheit(37.8);
```

- limitação: só faz conversão em um dos sentidos.

4. **Quarta e última versão:** conversão de mão dupla

```
class Conversor4
{
  double celsiusParaFahrenheit(double c)
  {
    return 9.0 * c / 5.0 + 32.0;
  }
  double fahrenheitParaCelsius(double f)
  {
    return 5.0 * (f - 32.0) / 9.0;
  }
}
```

   - para executar este conversor, podemos digitar o seguinte na janela do interpretador:

```
Conversor4 c4 = new Conversor4();
c4.celsiusParaFahrenheit(37.8);
c4.FahrenheitParaCelsius(-20.3);
```

   Note que, para que o seu programa fique bem legível e elegante, é muito importante o alinhamento dos abre-chaves `{` com os fecha-chaves `}` correspondentes. Em programas mais complexos, esse correto alinhamento (indentação) ajuda muito a tornar o programa mais claro para seres humanos.

Exercícios

1. Crie uma classe `Conversor5` que inclua também a escala Kelvin (K). Esta classe deve conter conversores entre as três escalas de temperatura (Celsius, Fahrenheit e Kelvin), totalizando seis funções. A relação entre as três escalas é dada por:

$$\frac{F-32}{9} = \frac{C}{5} = \frac{K-273}{5}$$

2. Iremos agora construir uma classe que calcula o valor de um número ao quadrado e ao cubo. Para tal, crie uma classe que contenha dois métodos. O primeiro método deve receber um número e devolver o seu quadrado e o segundo método deve receber um número e devolver o seu valor ao cubo. Escolha nomes para a classe e métodos que facilitem a compreensão de seu funcionamento.

3. Neste exercício iremos simular um jogo de tênis. A partida de tênis é composta por diversos papéis: jogador, juiz de cadeira, juiz de linha, treinador, gandula, torcedor.

   Para cada um destes papéis, descreva as mensagens que os atores de cada papel devem receber, e seu comportamento para cada uma delas. Utilize como modelo a descrição do papel `Bandeirinha` apresentada no início deste capítulo.

Capítulo 4

Testes Automatizados

Quais novidades veremos neste capítulo?

- comando `if` e `else`;
- comparações == (igualdade) e != (diferença);
- definição de variáveis inteiras e de ponto flutuante;
- impressão de texto;
- comentários.

Desde o início, a computação sempre esteve sujeita erros. O termo *bug*, para denotar erro, tem uma origem muito anterior (vem de um inseto que causava problemas de leitura no fonógrafo de Thomas Edison em 1889). Várias outras histórias reais, ou nem tanto, também apareceram no início da informática. Infelizmente, é muito difícil garantir que não existam erros em programas. Uma das formas de se garantir que certos erros não vão ocorrer é testando algumas situações.

Apesar da célebre afirmação de Edsger Dijkstra [1] de que testes podem apenas mostrar a presença de erros e não a sua ausência, eles podem ser os nossos grandes aliados no desenvolvimento de programas corretos. Intuitivamente, quanto mais testes fizermos em um programa e quanto mais abrangentes eles forem, mais confiantes podemos ficar com relação ao seu funcionamento. Por outro lado, podemos usar o próprio computador para nos ajudar, isto é, podemos criar testes automatizados. Em vez de fazermos os testes "na mão", faremos com que o computador seja capaz de verificar o funcionamento de uma seqüência de testes. Veremos neste capítulo como desenvolver testes automatizados, passo a passo, para os conversores de temperatura vistos anteriormente.

No início da computação não havia uma preocupação muito grande com os testes, que eram feitos de forma manual pelos próprios programadores. Os grandes testadores eram os usuários finais. É interessante notar que isto acontece com alguns produtos ainda hoje. Com o aparecimento da Engenharia de Software ficou clara a necessidade de se efetuarem testes, tanto que em várias empresas de desenvolvimento de software existe a figura do testador, responsável por tentar encontrar erros em sistemas. Hoje existe uma tendência para se

[1] Um dos mais influentes membros da geração dos criadores da Ciência da Computação. A página `http://www.cs.utexas.edu/users/EWD` contém cópias de vários de seus manuscritos.

considerar que testes automatizados são muito importantes, devendo ser escritos mesmo antes de se escrever o código propriamente dito, técnica esta chamada de *testes a priori*.

Veremos como testar os diversos conversores de temperatura. Como testar o nosso primeiro programa em Java (Conversor).

```
Conversor c1 = new Conversor()

// a resposta esperada é o equivalente a 40C em F
if (c1.celsiusParaFahrenheit() == 104)
  System.out.println("Funciona");
else
  System.out.println("Não funciona");
```

Note que para fazer o teste utilizamos o comando condicional `if else`. O formato genérico deste comando é o seguinte.

```
if (CONDIÇÃO)
  COMANDO-1;
else
  COMANDO-2;
```

Se a `CONDIÇÃO` é verdadeira, o `COMANDO-1` é executado, caso contrário, o `COMANDO-2` é executado.

A classe `Conversor2` possui um método que aceita um parâmetro, veja o teste abaixo:

```
Conversor2 c2 = new Conversor2();

// cria duas variáveis inteiras
int entrada = 40;
int resposta = 104;

// a resposta esperada é o equivalente à entrada C em F
if (c2.celsiusParaFahrenheit(entrada) == resposta)
  System.out.println("Funciona");
else
  System.out.println("Não funciona");
```

Note que, para realizar o teste acima, definimos duas variáveis inteiras chamadas de `entrada` e `resposta`. A linha

```
int entrada = 40;
```

faz na verdade duas coisas. Primeiro, ela declara a criação de uma nova variável (`int entrada;`) e, depois, atribui um valor inicial a esta variável (`entrada = 40;`). Na linguagem Java, se o valor inicial de uma variável não é atribuído, o sistema atribui o valor 0 à variável automaticamente. Note que isso não é necessariamente verdade em outras linguagens como, por exemplo, C e C++.

Podemos também testar o `Conversor2` para outros valores. Por exemplo, para as entradas (e respostas): 20 (68) e 100 (212).

```
entrada = 20; // como as variáveis já foram declaradas acima, basta usá-las
resposta = 68;
if (c2.celsiusParaFahrenheit(entrada) == resposta)
  System.out.println("Funciona");
else
```

```
    System.out.println("Não funciona");

  entrada = 100;
  resposta = 212;
  if (c2.celsiusParaFahrenheit(entrada) == resposta)
    System.out.println("Funciona");
  else
    System.out.println("Não funciona");
```

No programa acima o texto *Funciona* será impresso na tela a cada sucesso, o que poderá causar uma poluição visual caso tenhamos dezenas ou centenas de testes. O ideal para um testador é que ele fique silencioso caso os testes dêem certo e chame a atenção caso ocorra algum erro. Podemos então mudar o programa para:

```
Conversor2 c2 = new Conversor2();

int entrada = 40;
int resposta = 104;
if (c2.celsiusParaFahrenheit(entrada) != resposta)
  System.out.println("Não funciona para 40");

entrada = 20;
resposta = 68;
if (c2.celsiusParaFahrenheit(entrada) != resposta)
  System.out.println("Não funciona para 20");

entrada = 100;
resposta = 212;
if (c2.celsiusParaFahrenheit(entrada) != resposta)
  System.out.println("Não funciona para 100");

System.out.println("Fim dos testes");
```

Note que o comando `if` acima foi utilizado sem a parte do `else`, o que é perfeitamente possível. Adicionamos também uma linha final para informar o término dos testes. Ela é importante no caso em que todos os testes dão certo para que o usuário saiba que a execução dos testes foi encerrada.

Uma forma de simplificar os comandos de impressão é usar a própria entrada como parâmetro, o que pode ser feito da seguinte forma:

```
System.out.println("Não funciona para " + entrada);
```

Criaremos agora os testes para o `Conversor4`. Mas, agora, devem ser testados os seus dois métodos. Introduziremos um testador automático criando uma classe com apenas um método que faz o que vimos.

```
class TestaConversor4
{
  int testaTudo()
  {
    Conversor4 c4 = new Conversor4();
    double celsius = 10.0;
    double fahrenheit = 50.0;

    if (c4.celsiusParaFahrenheit(celsius) != fahrenheit)
      System.out.println("C–> F não funciona para " + celsius);
```

```
    if (c4.fahrParaCelsius(fahrenheit) != celsius)
      System.out.println("F–> C não funciona para " + fahrenheit);
    celsius = 20.0;
    fahrenheit = 68.0;
    if (c4.celsiusParaFahrenheit(celsius) != fahrenheit)
      System.out.println("C–> F não funciona para " + celsius);
    if (c4.fahrParaCelsius(fahrenheit) != celsius)
      System.out.println("F–> C não funciona para " + fahrenheit);
    celsius = 101.0;
    fahrenheit = 213.8;
    if (c4.celsiusParaFahrenheit(celsius) != fahrenheit)
      System.out.println("C–> F não funciona para " + celsius);
    if (c4.fahrParaCelsius(fahrenheit) != celsius)
      System.out.println("F–> C não funciona para " + fahrenheit);
    System.out.println("Final dos testes");
    return 0;
  }
}
```

Comentários: Você deve ter notado que, no meio de alguns exemplos de código Java deste capítulo, nós introduzimos uns comentários em português. Comentários deste tipo são possíveis em praticamente todas as linguagens de programação e servem para ajudar os leitores do seu código a compreender mais facilmente o que você escreveu.
Comentários em Java podem ser inseridos de duas formas. O símbolo `//` faz com que tudo o que aparecer após este símbolo até o final da linha seja ignorado pelo compilador Java; este é o formato que utilizamos neste capítulo e que utilizaremos quase sempre neste livro.
A segunda forma é através dos símbolos `/*` (que indica o início de um comentário) e `*/` (que indica o final de um comentário). Neste caso, um comentário pode ocupar várias linhas. Esta técnica é também útil para desprezarmos temporariamente um pedaço do nosso programa. Por exemplo, se queremos testar se uma classe funciona mesmo se removermos um de seus métodos, não é necessário apagar o código do método. Basta acrescentar um `/*` na linha anterior ao início do método, acrescentar um `*/` logo após a última linha do método e recompilar a classe. Posteriormente, se quisermos reinserir o método, basta remover os símbolos de comentários.
Outro uso para este último tipo de comentários é quando desejamos testar uma versão alternativa de um método. Comentamos a versão antiga do método, inserimos a versão nova e testamos a classe. Podemos aí decidir qual versão utilizar e, então, remover a que não mais será usada.

Agora, vamos treinar o que aprendemos com alguns exercícios. Em particular, os exercícios 3, de refatoração, e 4, que indica quais casos devem ser testados, são muito importantes.

Exercícios

1. Escreva uma classe `TestaConversor3` para testar a classe `Conversor3`.

2. **Importante**: Refatore a classe `TestaConversor4` de modo a eliminar as partes repetidas do código. *Dica:* Crie um método que realiza o teste das duas funções.

3. **Importante**: Podemos criar testes para objetos de uma classe que não sabemos como foi implementada. Para tal, basta conhecermos suas entradas e saídas. Escreva um teste automatizado para a seguinte classe:

```
class Contas
{
  double calculaQuadrado(double x);
  double calculaCubo(double x);
}
```

   Ao escrever o teste, pense em testar vários casos com características diferentes. Em particular, não deixe de testar os casos limítrofes que é, geralmente, onde a maioria dos erros tendem a se concentrar. Na classe `Contas` acima, os casos interessantes para teste seriam, por exemplo:

   - $x = 0$;
   - x é um número positivo pequeno;
   - x é um número negativo pequeno;
   - x é um número positivo grande;
   - x é um número negativo grande.

   Além disso, seria interessante que, em alguns dos casos acima, x fosse um número inteiro e em outros casos, um número fracionário.

4. *Programas confiáveis:* Se conhecemos uma implementação e sabemos que a mesma funciona de maneira confiável, ela pode servir de base para o teste de outras implementações. Caso duas implementações diferentes produzam resultados distintos para os mesmos dados de entrada, podemos dizer que pelo menos uma das duas está errada;
   Supondo que a classe `Contas` do exercício anterior é confiável, escreva um teste automatizado que utiliza esta classe para testar a classe `ContasNãoConfiavel` dada abaixo:

```
class ContasNãoConfiável
{
  double calculaQuadrado(double x);
  double calculaCubo(double x);
}
```

Resoluções

- Exercício 2

```
class TesteConversor4Refatorado
{
  int testePontual(double celsius, double fahrenheit)
  {
    Conversor c4 = new Conversor4();
```

```
    if (c4.celsiusParaFahrenheit(celsius) != fahrenheit)
      System.out.println("C-> F não funciona para " + celsius);
    if (c4.fahrParaCelsius(fahrenheit) != celsius)
      System.out.println("F-> C não funciona para " + fahrenheit);
    return 0;
  }

  int testaTudo()
  {
    double celsius = 10.0;
    double fahrenheit = 50.0;
    testePontual(celsius, fahrenheit);
    celsius = 20.0;
    fahrenheit = 68.0;
    testePontual(celsius, fahrenheit);
    celsius = 101.0;
    fahrenheit = 213.8;
    testePontual(celsius, fahrenheit);
    System.out.println("Final dos testes");
    return 0;
  }
}
```

- Exercício 3

```
class TesteContas
{
  int teste()
  {
    Contas contas = new Contas();
    double valor = 4.0;

    if (contas.calculaQuadrado(valor) != 16.0)
      System.out.println("Não funciona para calcular o quadrado de 4");
    if (contas.calculaCubo(valor) != 64.0)
      System.out.println("Não funciona para calcular 4 ao cubo");
    return 0;
  }
}
```

Capítulo 5

Métodos com Vários Parâmetros

Quais novidades veremos neste capítulo?

- Métodos com vários parâmetros;
- Atributos;
- Métodos que devolvem nada (`void`);
- Escolha de bons nomes.

No último capítulo, vimos um método que recebe vários parâmetros. Apesar de não estarmos apresentando nenhuma novidade conceitual, vamos reforçar a possibilidade da passagem de mais de um parâmetro.

Ao chamarmos um método que recebe múltiplos parâmetros, assim como no caso de métodos de um parâmetro, devemos passar valores do mesmo tipo que aqueles que o método está esperando. Por exemplo, se um objeto (ator) tem um método (entende uma mensagem) que tem como parâmetro um número inteiro, não podemos enviar a este objeto um número `double`. Apesar de ser intuitivo, também vale ressaltar que a ordem dos parâmetros é importante, de modo que na chamada de um método devemos respeitar a ordem que está na sua definição.

Vamos começar com o cálculo da área de uma circunferência, onde é necessário apenas um parâmetro, o raio:

```
class Círculo1
{
  double calculaÁrea(double raio)
  {
    return 3.14159 * raio * raio;
  }
}
```

Nota Lingüística: Em Java, o nome de variáveis, classes e métodos pode conter caracteres acentuados. Muitos programadores evitam o uso de acentuação mesmo ao usar nomes em português. Este costume advém de outras linguagens mais antigas, como C, que não permitem o uso de caracteres acentuados. Você pode usar caracteres acentuados se quiser mas lembre-se que, se você tiver uma variável chamada `área`, ela será diferente de outra chamada `area` (sem acento); ou seja, se você decidir usar acentos, deverá usá-los consistentemente sempre.

Podemos sofisticar o exemplo calculando também o perímetro, com o seguinte método:

```
double calculaPerímetro(double raio)
{
  return 3.14159 * 2.0 * raio;
}
```

O seguinte trecho de código calcula e imprime a área e o perímetro de uma circunferência de raio 3.0:

```
Círculo1 c = new Círculo1();

System.out.println(c.calculaÁrea(3.0));
System.out.println(c.calculaPerímetro(3.0));
```

Vejamos agora uma classe `Retângulo` que define métodos para o cálculo do perímetro e da área de um retângulo. Neste caso são necessários dois parâmetros.

```
class Retângulo1
{
  int calculaÁrea(int lado1, int lado2)
  {
    return lado1 * lado2;
  }
  int calculaPerímetro(int lado1, int lado2)
  {
    return 2 * (lado1 + lado2);
  }
}
```

As chamadas para os métodos podem ser da seguinte forma:

```
Retângulo1 r = new Retângulo1();

System.out.println(r.calculaÁrea(2, 3));
System.out.println(r.calculaPerímetro(2, 3));
```

5.1 Atributos

Assim como no exemplo do teatrinho dos objetos onde os jogadores tinham como característica o time e o número da camisa (atributos), também podemos ter algo semelhante para o caso dos retângulos. Podemos fazer com que os dados sobre os lados sejam características dos objetos. Isto é feito da seguinte forma:

```
class Retângulo2
{
  int lado1;
  int lado2;

  int calculaÁrea()
  {
    return lado1 * lado2;
  }
  int calculaPerímetro()
  {
    return 2 * (lado1 + lado2);
  }
}
```

Na classe acima, `lado1` e `lado2` são atributos que farão parte de todas as instâncias da classe `Retângulo2`. Um atributo, por ser definido fora dos métodos, pode ser acessados a partir de qualquer método da classe onde ele é definido. Atributos são também chamados de propriedades ou variáveis de instância.

No entanto, no exemplo acima, ficou faltando algum método para carregar os valores dos lados nas variáveis `lado1` e `lado2`. Poderíamos escrever um método para cada atributo ou então, apenas um método para definir o valor de ambos como no exemplo seguinte:

```
void carregaLados(int l1, int l2) // este método não devolve nenhum valor
{
  lado1 = l1;
  lado2 = l2;
}
```

Nota Lingüística: `void` em inglês significa vazio, nada ou nulidade. Assim, quando temos um método que não devolve nenhum valor, como, por exemplo, o método `carregaLados` acima, colocamos a palavra `void` antes da definição do método para indicar que ele não devolverá nada. Neste caso, não é necessário incluir o comando `return` no corpo do método. Mesmo assim, podemos usar um comando de retorno vazio para provocar o fim prematuro da execução do método, como em: `return ;` . O uso do `return` vazio nem sempre é recomendado pois, em alguns casos, pode gerar código pouco claro, mas é uma opção permitida pela linguagem.

O funcionamento de um objeto da classe `Retângulo2` pode ser verificado com o código abaixo:

```
Retângulo2 r = new Retângulo2();

r.carregaLados(3, 5);
System.out.println(r.calculaÁrea());
System.out.println(r.calculaPerímetro());
```

Para não perdermos o hábito, também veremos como testar esta classe. Neste caso, temos duas opções: criar diversos objetos do tipo `Retângulo2`, um para cada teste, ou carregar diversos lados no mesmo objeto. Abaixo, a cada chamada de `TestePontual`, um novo `Retângulo2` é criado:

```
class TestaRetângulo
{
  void testePontual(int l1, int l2)
  {
    Retângulo2 r = new Retângulo2();

    r.carregaLados(l1, l2);
    if (r.calculaÁrea() != l1 * l2)
      System.out.println("Não funciona área para lados "
                         + l1 + " e " + l2);
    if (r.calculaPerímetro() != 2 * (l1 + l2))
      System.out.println("Não funciona perímetro para lados "
                         + l1 + " e " + l2);
  }

  void testaTudo ()
  {
    testePontual(10, 20);
    testePontual(1, 2);
    testePontual(3, 3);
  }
}
```

Da mesma forma que os lados foram atributos para a classe `Retângulo2`, podemos fazer o mesmo para a classe `Círculo1`.

```
class Círculo2
{
  double raio;

  void carregaRaio(double r)
  {
    raio = r;
  }
  double calculaÁrea()
  {
    return 3.14159 * raio * raio;
  }
  double calculoPerímetro()
  {
    return 3.14159 * 2.0 * raio;
  }
}
```

Assim como vimos anteriormente podemos também utilizar objetos de uma classe sem conhecer a sua implementação. Por exemplo, suponha que temos acesso a uma classe `Cálculo` que possui o seguinte método:

```
int calculaPotência(int x, int n);
```

Para calcular a potência poderíamos ter o seguinte trecho de código:

```
Cálculo c = new Cálculo();

System.out.println("2 elevado a 5 é igual a: " + c.calculaPotência(2, 5));
```

5.2 A importância da escolha de bons nomes

A característica mais importante em um programa de computador é a clareza do seu código. Quanto mais fácil for para um programa ser entendido por um ser humano que o leia, melhor será o código. Portanto, é fundamental que os nomes das variáveis, classes e métodos sejam escolhidos com muito cuidado e critério. Programadores inexperientes (e alguns não tão inexperientes assim mas descuidados) tendem a não dar importância a este ponto que é essencial para o desenvolvimento de software de boa qualidade.

Os nomes das variáveis devem indicar claramente o seu significado, isto é, devem explicar o que é o conteúdo que elas guardam. Por exemplo, na classe `Círculo1` descrita no início deste capítulo, a escolha do nome `raio` para a variável é perfeita porque ela não deixa a menor dúvida sobre o que a variável irá guardar. Se ao invés de `raio`, o programador tivesse escolhido `x` como nome da variável, isto seria péssimo pois o nome `x` não traria informação nenhuma para o leitor do programa sobre o seu significado. Se o programador tivesse dado o nome `r` para a variável, a escolha não chegaria a ser péssima, pois o `r` traz alguma informação, mesmo que sutil. Mas, mesmo assim, não seria uma solução tão boa quanto nomear como `raio`, pois este não deixa a menor dúvida.

Portanto, é importante não ter preguiça de pensar em bons nomes e de escrever nomes um pouco mais longos. Por exemplo, o nome `númeroDeAlunos` é muito melhor do que `nda` e melhor do que `numAlun`. Quanto menor o esforço mental do leitor para compreender o programa, maior serão os ganhos a médio e longo prazos. A economia de alguns segundos que obtemos com o uso de nomes abreviados pode facilmente se tornar em horas de dor de cabeça no futuro.

A escolha dos nomes de classes e métodos também deve ser feita criteriosamente. Normalmente as classes representam tipos de objetos do mundo real ou do mundo virtual que estamos criando dentro do computador. Portanto, o mais comum é usarmos substantivos como nome para classes (embora esta não seja uma regra obrigatória). Os nomes das classes tem que explicar muito bem qual o tipo de objeto que ela irá representar. Por exemplo, chamar uma classe de `Aluno` e depois usar objetos desta classe para guardar as notas de um aluno e calcular sua média, não é uma boa escolha. Provavelmente, neste caso, `NotasDeAluno` seria um nome melhor.

Os métodos representam ações que são realizadas, normalmente manipulando os atributos da classe. Portanto, normalmente utiliza-se verbos no imperativo para nomear os métodos (novamente, esta não é uma regra obrigatória). Por exemplo, os métodos `calculaÁrea`, `calculaPerímetro`, `carregaLados` e `testaTudo` atendem bem a este critério.

Portanto, antes de escrever sua próxima classe, método ou variável, não se esqueça: **nomes são importantes!**

Exercícios

1. Neste exercício, construiremos uma classe que calcula a média aritmética de 4 notas e diz se o dono das notas foi aprovado, está de recuperação ou foi reprovado. Por exemplo,

   - Entrada:

     ```
     8.7, 7.2, 9.3, 7.4
     5.2, 3.4, 6.5, 2.1
     3.4, 5.1, 1.1, 2.0
     ```

- Saída:

```
Média: 8.15 -> aprovado.
Média: 4.3 -> recuperação.
Média: 2.9 -> reprovado.
```

Para isso, crie uma classe `Aluno` com métodos que carreguem as 4 notas em variáveis `p1`, `p2`, `p3`, `p4` e um método responsável por calcular a média aritmética das notas e dar o "veredito".

2. Escreva uma classe `Olá` com um único método `cumprimenta` que, a cada chamada, cumprimenta o usuário de uma entre 3 maneiras diferentes. *Dica:* use um atributo para, dependendo de seu valor, escolher qual das maneiras será usada; depois de imprimir a mensagem, altere o valor do atributo.

3. Construa a classe `Inteiro` que representa um número inteiro. Essa classe deve ter os seguintes atributos e métodos:

Classe `Inteiro`

– Atributos:

  * `int valor`
    Valor do inteiro representado.

– Métodos para interação com o usuário da classe:

  * `void carregaValor(int v)`
    Muda o valor representado por este objeto. O novo valor deve ser `v`.
  * `int devolveValor()`
    Devolve o valor representado por este objeto.
  * `int devolveValorAbsoluto()`
    Devolve o valor absoluto do valor representado por este objeto.
  * `void imprime()`
    Imprime algo que representa este objeto. Sugestão: imprima o seu valor.

Exemplo de uso no DrJava:

```
> Inteiro i = new Inteiro();
> i.carregaValor(14);
> i.devolveValor()
14
> i.carregaValor(-473158);
> i.devolveValor()
-473158
> i.devolveValorAbsoluto()
473158
> i.imprime();
```

```
Valor: -473158.
```

4. Acrescente à classe `Inteiro` algumas operações matemáticas, implementando os seguintes métodos:

Classe `Inteiro`

– Mais métodos para interação com o usuário da classe:

* `int soma(int v)`
  Soma v ao valor deste objeto (valor + v). Este objeto passa a representar o novo valor, que também deve ser devolvido pelo método.
* `int subtrai(int v)`
  Subtrai v do valor deste objeto (valor - v). Este objeto passa a representar o novo valor, que também deve ser devolvido pelo método.
* `int multiplicaPor(int v)`
  Multiplica o valor deste objeto por v (valor * v). Este objeto passa a representar o novo valor, que também deve ser devolvido pelo método.
* `int dividePor(int divisor)`
  Verifica se divisor é diferente de zero. Se não, imprime uma mensagem de erro e não faz nada (devolve o valor inalterado). Se for, divide o valor deste objeto por v (valor / divisor, divisão inteira). Este objeto passa a representar o novo valor, que também deve ser devolvido pelo método.

Exemplo de uso no DrJava:

```
> Inteiro i = new Inteiro();
> i.carregaValor(15);
> i.subtrai(20)
-5
> i.devolveValor()
-5
```

Se quiser, você também pode fazer versões desses métodos que não alteram o valor representado, apenas devolvem o valor da conta.

5. Crie uma classe `TestaInteiro` que verifica o funcionamento correto da classe `Inteiro` em diversas situações. Lembre-se de verificar casos como a divisão por diversos valores. Ao escrever os testes, você notará que a classe `Inteiro` tem uma limitação importante no método `dividePor`.

6. Crie uma classe `Aluno2` para calcular a média aritmética de 4 notas. Mas no lugar de um método que recebe 4 parâmetros, a classe deve conter 4 métodos `recebeNotaX`, onde `X` = 1, 2, 3 ou 4, para receber as notas das provas, de modo que cada método receba apenas uma nota. A classe deve conter ainda um método `imprimeMédia` que imprime a média final do aluno, dizendo se ele foi aprovado ou reprovado. Em seguida, escreva uma classe `TestaAluno2` que verifica se a classe `Aluno2` calcula as médias corretamente.

Exemplo de utilização da classe:

```
> Aluno2 aluno = new Aluno2();
> aluno.recebeNota1(5.0);
> aluno.recebeNota2(7.0);
> aluno.recebeNota3(9.0);
> aluno.recebeNota4(7.0);
> aluno.imprimeMedia();
Média: 7.0 -> aprovado.
```

Capítulo 6

`if else` Encaixados

Quais novidades veremos neste capítulo?

- novidade: `if else` encaixados;
- exercício para reforçar o que aprendemos até agora.

No Capítulo 4, vimos pela primeira vez o conceito de desvio condicional através dos comandos `if` e `else`. Entretanto, vimos exemplos iniciais onde apenas um comando simples era executado, no caso, comandos de impressão. Na prática, comandos `if` e `else` podem ser encaixados de forma a criar estruturas muito complexas. Para isto, veremos inicialmente como criar blocos de comandos através do seguinte exemplo:

```
if (CONDIÇÃO)
{  // início do bloco
   COMANDO-1;
   COMANDO-2;
      ...
   COMANDO-n;
}
```

Neste trecho de código, caso a `CONDIÇÃO` seja verdadeira, os comandos, de 1 a n, são executados seqüencialmente. Veremos a seguir que é comum que alguns destes comandos sejam também comandos de desvio condicional.

Vamos iniciar programando uma classe para representar um triângulo retângulo. Ela contém um método que, dados os comprimentos dos lados do triângulo, verifica se o mesmo é retângulo ou não.

```
class TrianguloRetângulo
{
  void verificaLados(int a, int b, int c)
  {
    if (a * b * c != 0) // nenhum lado pode ser nulo
    {
      if (a*a == b*b + c*c)
        System.out.println("Triângulo retângulo.");
      if (b*b == a*a + c*c)
        System.out.println("Triângulo retângulo.");
```

```
      if (c*c == a*a + b*b)
        System.out.println("Triângulo retângulo.");
    }
  }
}
```

O método acima pode ser chamado da seguinte forma:

```
TrianguloRetângulo r = new TrianguloRetângulo();
r.verificaLados(1, 1, 1);
r.verificaLados(3, 4, 5);
```

Limitações:

1. mesmo que um `if` seja verdadeiro, ele executa os outros `if`s. Em particular, se tivéssemos um triângulo retângulo para o qual vários desses `if`s fossem verdadeiros, ele imprimiria esta mensagem várias vezes (neste exemplo específico, isto não é possível);

2. este método só imprime uma mensagem se os dados correspondem às medidas de um triângulo retângulo, se não é um triângulo retângulo, ele não imprime nada. Através do uso do `else` podemos imprimir mensagens afirmativas e negativas:

```
class TrianguloRetângulo2
{
  void verificaLados(int a, int b, int c)
  {
    if (a * b * c != 0) // nenhum lado pode ser nulo
    {
      if (a*a == b*b + c*c)
        System.out.println("Triângulo retângulo.");
      else if (b*b == a*a + c*c)
        System.out.println("Triângulo retângulo.");
      else if (c*c == a*a + b*b)
        System.out.println("Triângulo retângulo.");
      else
        System.out.println("Não é triângulo retângulo.");
    }
    else
      System.out.println("Não é triângulo pois possui lado de comprimento nulo.");
  }
}
```

Caso sejam necessários outros métodos, como um para o cálculo de perímetro, é interessante colocar os lados como atributos da classe.

```
class TrianguloRetângulo3
{
  int a, b, c;
  void carregaLados(int l1, int l2, int l3)
  {
    a = l1;
    b = l2;
    c = l3;
```

```
  }
  int calculaPerímetro()
  {
    return a + b + c;
  }
  void verificaLados()
  {
    if (a * b * c != 0) // nenhum lado pode ser nulo
    {
      if (a*a == b*b + c*c)
        System.out.println ("Triângulo retângulo.");
      else if (b*b == a*a + c*c)
        System.out.println ("Triângulo retângulo.");
      else if (c*c == a*a + b*b)
        System.out.println ("Triângulo retângulo.");
      else
        System.out.println("Não é triângulo retângulo.");
    }
    else
      System.out.println("Não é triângulo pois possui lado de comprimento nulo.");

  }
}
```

Exercícios

1. Você foi contratado por uma agência de viagens para escrever uma classe em Java para calcular a conversão de reais para dólar de acordo com a taxa de compra e a taxa de venda. Para isso, escreva uma classe `ConversorMonetário` que inclua os seguintes métodos:

   (a) `defineTaxaCompra()` e `defineTaxaVenda()`;

   (b) `imprimeTaxas()` que imprime o valor das 2 taxas de conversão;

   (c) `vendeDólar()` que recebe uma quantia em dólares e devolve o valor correspondente em reais;

   (d) `compraDólar()` que recebe uma quantia em dólares e devolve o valor correspondente em reais.

   Em seguida, escreva uma classe `TestaConversorMonetário` que define diferentes taxas de compra e venda de dólares e, para cada taxa de conversão, realiza operações de compra e venda.

2. Escreva uma classe `Baskara` que possui 3 atributos do tipo `double` correspondentes aos coeficientes a, b e c de uma equação do segundo grau. Escreva um método para carregar valores nestes atributos e, em seguida, escreva os 4 métodos seguintes:

   (a) `delta()` deve calcular o δ da fórmula de Baskara;

   (b) `númeroDeRaízesReais()` deve devolver um inteiro indicando quantas raízes reais a equação possui;

   (c) `imprimeRaízesReais()` deve imprimir as raízes reais;

(d) `imprimeRaízesImaginárias()` deve imprimir as raízes imaginárias.

Para calcular a raiz quadrada, você pode utilizar o método `java.lang.Math.sqrt(double x)`, que recebe um `double` como parâmetro e devolve outro `double`. Você pode supor que o primeiro coeficiente, *a*, é diferente de 0.

3. Crie uma classe contendo um método que, dado um ponto determinado pelas suas coordenadas `x` e `y`, reais, imprime em qual quadrante este ponto está localizado. O primeiro quadrante corresponde aos pontos que possuem `x` e `y` positivos, o segundo quadrante a `x` positivo e `y` negativo e assim por diante. Para resolver este exercício, será necessário utilizar os operadores < e >. Sua utilização é similar à do operador == utilizado até agora.

4. **[Desafio!]** *São apresentadas a você doze esferas de aparência idêntica. Sabe-se que apenas uma delas é levemente diferente das demais em massa, mas não se sabe qual e nem se a massa é maior ou menor. Sua missão é identificar essa esfera diferente e também dizer se ela é mais ou menos pesada. Para isso você tem apenas uma balança de prato (que só permite determinar igualdades/desigualdades). Ah, sim, pequeno detalhe: o desafio é completar a missão em não mais que três pesagens.*

   Escreva um programa que resolve esse desafio. O seu programa deve dar uma resposta correta sempre e também informar as três ou menos pesagens que permitiram concluir a resposta. Como esse problema é um tanto complicado, recomendamos que você implemente o modelo descrito no quadro a seguir.

   Cada esfera deve ser representada por um inteiro entre 1 e 12. Para representarmos a esfera diferente, usaremos, além do identificador inteiro, uma variável booleana que receberá o valor `true` se a esfera for mais pesada ou o valor `false` se a esfera for mais leve.

   Importante: note que, para que o problema tenha sentido, o método resolveDesafioDasDozeEsferas não deve de modo algum acessar os atributos esferaDiferente e maisPesada para inferir a resposta. Quem dá pistas para este método sobre o valor desses atributos são os métodos pesa#x#.

   Lembre-se de que você também pode implementar métodos adicionais, se achar necessário ou mais elegante. Ou ainda, se você já se sente seguro(a), você pode implementar a(s) sua(s) própria(s) classe(s).

   Exemplo de uso no DrJava:

```
> DesafioDasEsferas dde = new DesafioDasEsferas();
> dde.defineEsferaDiferente(4, false);
Início do desafio: esfera 4 mais leve.
> dde.resolveDesafioDasDozeEsferas();
Pesagem 1: 1 2 3 4 5 x 6 7 8 9 10.
Resultado: (1) lado direito mais pesado.
Pesagem 2: 1 2 x 3 4.
Resultado: (-1) lado esquerdo mais pesado.
Pesagem 3: 1 x 2.
Resultado: (0) balança equilibrada.
Resposta: esfera 3 mais leve.
[Resposta errada!]
```

Note que a resposta está errada! (Além disso, as pesagens não permitem dar a resposta certa.)

Classe `DesafioDasEsferas`

– Atributos:

* `int esferaDiferente`
  Identificador da esfera com peso diferente.
* `boolean maisPesada`
  Diz se a esfera diferente é ou não mais pesada que as demais (isto é, true para mais pesada e false para mais leve).
* `int numPesagens`
  Representa o número de pesagens realizadas.

– Métodos para interação com o usuário:

* `void defineEsferaDiferente(int esfera, boolean pesada)`
  Determina qual é a esfera diferente (parâmetro esfera), e se ela é mais pesada ou não (parâmetro pesada). Além disso, reinicia as pesagens, isto é, o número de pesagens realizadas volta a ser zero.
* `void resolveDesafioDasDozeEsferas()`
  Resolve o Desafio das Doze das Esferas. Este método deve imprimir as 3 (ou menos) pesagens feitas, e no final a resposta correta. Este método deve se utilizar dos métodos para uso interno descritos abaixo. Dica: na implementação deste método você também usará uma quantidade grande de ifs e elses encaixados.

– Métodos para uso interno da classe:

* `int pesa1x1(int e1, int d1)`
* `int pesa2x2(int e1, int e2, int d1, int d2)`
* `int pesa3x3(int e1, int e2, int e3, int d1, int d2, int d3)`
* `int pesa4x4(int e1, int e2, int e3, int e4, int d1, int d2, int d3, int d4)`
* `int pesa5x5(int e1, int e2, int e3, int e4, int e5, int d1, int d2, int d3, int d4, int d5)`
* `int pesa6x6(int e1, int e2, int e3, int e4, int e5, int e6, int d1, int d2, int d3, int d4, int d5, int d6)`

  Os métodos acima (no formato pesa#x#) funcionam de forma semelhante. Eles representam as possíveis pesagens e devolvem o resultado. Os parâmetros representam as esferas que são pesadas, os começados por e (ou seja, e1, e2, ...) representam esferas que vão para o prato esquerdo e os começados por d (d1, d2, ...) são as do prato direito. Lembrando, cada esfera é representada por um inteiro entre 1 e 12. Então, por exemplo, para comparar as esferas 1, 7 e 3 com as esferas 4, 5 e 6, basta chamar pesa3x3(1,7,3,4,5,6). Os métodos devem devolver -1 se a balança pender para o lado esquerdo, 0 se os pesos forem iguais ou 1 se a balança pender para o lado direito. Esses métodos também devem incrementar o número de pesagens realizadas.

Capítulo 7

Programas com Vários Objetos

Quais novidades veremos neste capítulo?

- Programas com vários objetos.

Até agora, todos os programas que vimos lidavam com apenas um objeto. No entanto, podemos ter programas que lidam com vários objetos. Estes objetos podem pertencer todos à mesma classe ou a classes diferentes. Exemplo:

1. Vários objetos do mesmo tipo

```
Flor rosa = new Flor();
Flor margarida = new Flor();
Flor florDeLaranjeira = new Flor();

rosa.cor("vermelha");
rosa.aroma("muito agradável");
margarida.aroma("sutil");
florDeLaranjeira.aroma("delicioso");
```

2. Vários objetos de tipos (classes) diferentes:

```
Cachorro floquinho = new Cachorro();
Gato mingau = new Gato();
Rato topoGiggio = new Rato();
Vaca mimosa = new Vaca();

floquinho.lata();
mingau.mie();
topoGiggio.comaQueijo();
mingau.persiga(topoGiggio)
floquinho.persiga(mingau);
mimosa.passePorCima(floquinho, mingau, topoGiggio);
```

Vejamos agora um exemplo de utilização de objetos de 3 tipos diferentes em conjunto. Neste exemplo, teremos 3 classes representando prismas, quadrados e triângulos retângulos e criaremos uma instância de quadrado, uma de triângulo retângulo e duas de prismas.

Note, no exemplo abaixo, que utilizamos um padrão diferente para nomear os métodos que servem para atribuir valores aos atributos. Segundo este padrão, muito utilizado por programadores avançados em linguagens como C++ e Smalltalk, o nome do método é exatamente o nome do atributo correspondente. Por exemplo, o método `void altura(double a)` é utilizado para definir o valor do atributo `altura` e assim por diante. Ao escrever seus programas, você é livre para escolher entre qualquer um dos padrões existentes, mas é importante que você seja coerente, ou seja, após escolher o padrão de nomeação, aplique-o consistentemente em todo o seu código.

```
class Prisma
{
  double altura;
  double áreaDaBase;

  void altura(double a)
  {
    altura = a;
  }

  void áreaDaBase(double a)
  {
    áreaDaBase = a;
  }

  double volume()
  {
    return áreaDaBase * altura;
  }
}

class Quadrado
{
  double lado;

  void lado(double l)
  {
    lado = l;
  }

  double área()
  {
    return lado * lado;
  }
}

class TrianguloRetângulo
{
  double cateto1;
  double cateto2;
```

```
  // note a indentação supercompacta!
  void cateto1(double c) { cateto1 = c; }
  void cateto2(double c) { cateto2 = c; }

  double área()
  {
    return cateto1 * cateto2 / 2.0;
  }
}
```

Agora, utilizando o interpretador, podemos criar objetos destes vários tipos e utilizá-los em conjunto:

```
Quadrado q = new Quadrado();
TrianguloRetângulo tr = new TrianguloRetângulo();
Prisma prismaBaseQuadrada = new Prisma();
Prisma prismaBaseTriangular = new Prisma();

q.lado(10.0);
tr.cateto1(20.0);
tr.cateto2(30.0);

prismaBaseQuadrada.altura(3.0);
prismaBaseTriangular.altura(1.0);

prismaBaseQuadrada.áreaDaBase(q.área());
prismaBaseTriangular.áreaDaBase(tr.área());

if(prismaBaseQuadrada.volume() > prismaBaseTriangular.volume())
    System.out.println("O prisma de base quadrada tem maior volume");
else if(prismaBaseTriangular.volume() > prismaBaseQuadrada.volume())
    System.out.println("O prisma de base triangular tem maior volume");
else
    System.out.println("Ambos os prismas têm o mesmo volume");
```

Nota sobre o interpretador do DrJava: para conseguir digitar todos os `if`s encaixados no interpretador do DrJava sem que ele tente interpretar cada linha em separado, é preciso utilizar Shift+Enter em vez de apenas Enter no final de cada linha dos `if`s encaixados. Apenas no final da última linha (a que contém o `println` final) é que se deve digitar apenas Enter para que o DrJava então interprete todas as linhas de uma vez.

Exercícios

1. Utilizando a classe `Conversor5` definida no exercício 1 do Capítulo 3, escreva uma classe contendo três métodos, onde cada método recebe uma temperatura `x` utilizando uma escala de temperaturas e imprime os valores de `x` nas demais escalas de temperatura.

2. Escreva uma classe `Rendimentos` que contenha os seguintes métodos a fim de contabilizar o total de rendimentos de uma certa pessoa em um certo ano:

- `rendimentosDePessoaFísica(double);`
- `rendimentosDePessoaJurídica(double);`
- `rendimentosDeAplicaçõesFinanceiras(double);`
- `rendimentosNãoTributáveis(double);`
- `double totalDeRendimentosTributáveis();`

Em seguida, escreva uma classe `TabelaDeAlíquotas` que possui:

- um método `alíquota()` que recebe como parâmetro o total de rendimentos tributáveis de uma pessoa e devolve um número entre 0 e 1.0 correspondente à alíquota de imposto que a pessoa deverá pagar e
- um método `valorADeduzir()` que recebe como parâmetro o total de rendimentos tributáveis de uma pessoa e devolve o valor a deduzir no cálculo do imposto.

Para escrever esta classe, utilize a tabela do IR 2006 abaixo:

Rendimentos Tributáveis	Alíquota	Parcela a deduzir
Até R$ 13.968,00		
De R$ 13.968,01 a R$ 27.912,00	0.15	R$ 1.904,40
acima de R$ 27.912,00	0.275	R$ 5.076,90

Agora escreva uma classe `CalculadoraDeImposto` que possui um único método que recebe como parâmetro o valor dos rendimentos tributáveis de uma pessoa e devolve o valor do imposto a ser pago.

Finalmente, escreva um trecho de código (para ser digitado no interpretador) que utiliza `Rendimentos` para definir os vários rendimentos de uma pessoa e `CalculadoraDeImposto` para calcular o imposto a pagar.

3. Suponha que você tenha as seguintes classes:

```
class A
{
  double a (int meses, double taxa)
  {
    return Math.pow((taxa + 100) / 100, meses) - 1;
  }
}

class B
{
  final double TAXA = 1.2;

  void b (double valorEmprestado, int meses)
  {
    A a = new A();
    double valorDaDívida = valorEmprestado + (a.a(meses, TAXA)*valorEmprestado);
```

```
    System.out.println("Dívida de " + valorDaDívida + " real(is), " +
                       "calculada com taxa de " + TAXA + "% ao mês.");
  }
}
```

(a) O que fazem os métodos `a` (da classe `A`) e `b` (da classe `B`)? Não precisa entrar em detalhes. Dica: para saber o que `Math.pow` faz consulte a página `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html`;

(b) Os nomes `a` e `b` (dos métodos) e `A` e `B` (das classes) são péssimos. Por quê? Que nomes você daria? Sugira, também, outro nome para a variável objeto (criada no interpretador);

(c) Acrescente alguns comentários no código do método `b`;

(d) Seria mais fácil digitar o valor 1.2 quando necessário, em vez de criar uma constante `TAXA` e utilizá-la. Então, por que isso foi feito? Cite, pelo menos, dois motivos. A palavra chave `final` faz com que o valor da variável `TAXA`, uma vez atribuído, não possa ser alterado.

Capítulo 8

Laços e Repetições

Quais novidades veremos neste capítulo?

- A idéia de laços em linguagens de programação;
- O laço `while`;
- O operador que calcula o resto da divisão inteira: `%`.

8.1 Laços em linguagens de programação

Vamos apresentar para vocês um novo conceito fundamental de programação: o *laço*. Mas o que pode ser isso? Um nome meio estranho, não? Nada melhor do que um exemplo para explicar.

Vamos voltar ao nosso velho conversor de temperatura. Imagine que você ganhou uma passagem para Nova Iorque e que os EUA não estão em guerra com ninguém. Você arruma a mala e se prepara para a viagem. Antes de viajar você resolve conversar com um amigo que já morou nos EUA. Ele acaba lhe dando uma dica: guarde uma tabelinha de conversão de temperaturas de Fahrenheit para Celsius. Ela será muito útil, por exemplo, para entender o noticiário e saber o que vestir no dia seguinte. Você então se lembra que já tem um conversor pronto. Basta então usá-lo para montar a tabela. Você chama então o DrJava e começa uma nova sessão interativa.

```
Welcome to DrJava.
> Conversor4 c = new Conversor4()
> c.fahrenheitParaCelsius(0)
-17.77777777777778
> c.fahrenheitParaCelsius(10)
-12.222222222222221
> c.fahrenheitParaCelsius(20)
-6.666666666666667
> c.fahrenheitParaCelsius(30)
-1.1111111111111112
```

```
> c.fahrenheitParaCelsius(40)
4.444444444444445
> c.fahrenheitParaCelsius(50)
10.0
> c.fahrenheitParaCelsius(60)
15.555555555555555
> c.fahrenheitParaCelsius(70)
21.11111111111111
> c.fahrenheitParaCelsius(80)
26.666666666666668
> c.fahrenheitParaCelsius(90)
32.22222222222222
> c.fahrenheitParaCelsius(100)
37.77777777777778
> c.fahrenheitParaCelsius(110)
43.333333333333336
>
```

Pronto, agora é só copiar as linhas acima para um editor de textos, retirar as chamadas ao método `fahrenheitParaCelsius` (pois elas confundem) e imprimir a tabela.

Será que existe algo de especial nas diversas chamadas do método `fahrenheitParaCelsius` acima? Todas elas são muito parecidas e é fácil adivinhar a próxima se sabemos qual a passada. Ou seja, a lei de formação das diversas chamadas do método é simples e bem conhecida. Não seria interessante se fosse possível escrever um trecho de código compacto que representasse essa idéia? Para isso servem os laços: eles permitem a descrição de uma seqüência de operações repetitivas.

8.2 O laço `while`

O nosso primeiro laço será o `while`, a palavra inglesa para *enquanto*. Ele permite repetir uma seqüência de operações enquanto uma *condição* se mantiver verdadeira. Mais uma vez, um exemplo é a melhor explicação. Experimente digitar as seguintes linhas de código no painel de interações do DrJava (lembre-se que para digitarmos as 5 linhas do comando `while` abaixo, é necessário usarmos Shift+Enter em vez de apenas Enter no final das 4 linhas iniciais do `while`):

```
Welcome to DrJava.
> int a = 1;
> while (a <= 10)
{
  System.out.println("O valor atual de a é: " + a);
  a = a + 1;
}
```

o resultado será o seguinte:

```
O valor atual de a é: 1
O valor atual de a é: 2
O valor atual de a é: 3
O valor atual de a é: 4
O valor atual de a é: 5
O valor atual de a é: 6
O valor atual de a é: 7
O valor atual de a é: 8
O valor atual de a é: 9
O valor atual de a é: 10
>
```

Vamos olhar com calma o código acima. Primeiro criamos uma variável inteira chamada `a`. O seu valor inicial foi definido como 1. A seguir vem a novidade: o laço `while`. Como dissemos antes, ele faz com que o código que o segue (e está agrupado usando chaves) seja executado enquanto a condição `a <= 10` for verdadeira. Inicialmente `a` vale 1, por isso este é o primeiro valor impresso. Logo depois de imprimir o valor de `a`, o seu valor é acrescido de 1, passando a valer 2. Neste momento o grupo de instruções que segue o `while` terminou. O que o computador faz é voltar à linha do `while` e verificar a condição novamente. Como `a` agora vale 2, ele ainda é menor que 10. Logo as instruções são executadas novamente. Elas serão executadas *enquanto* a condição for verdadeira, lembra? Mais uma vez, o valor atual de `a` é impresso e incrementado de 1, passando a valer 3. De novo o computador volta à linha do `while`, verifica a condição (que ainda é verdadeira) e executa as instruções dentro das chaves. Esse processo continua até que o `a` passe a valer 11, depois do décimo incremento. Neste instante, a condição torna-se falsa e na próxima vez que a condição do `while` é verificada, o computador pula as instruções dentro das chaves do `while`. Ufa, é isso! Ainda bem que é o computador que tem todo o trabalho! Uma das principais qualidades do computador é a sua capacidade de efetuar repetições. Ele faz isso de forma automatizada e sem se cansar. O laço é uma das formas mais naturais de aproveitarmos essa característica da máquina.

Agora vamos ver como esse novo conhecimento pode nos ajudar a montar a nossa tabela de conversão de forma mais simples e flexível. Se pensarmos bem, veremos que as operações realizadas para calcular as temperaturas para tabela são semelhantes ao laço apresentado. Só que no lugar de simplesmente imprimir os diferentes valores de uma variável, para gerar a tabela chamamos o método `fahrenheitParaCelsius` várias vezes. Vamos agora adicionar um método novo à classe `Conversor4`, que terá a função de imprimir tabelas de conversão para diferentes faixas de temperatura. O código final seria:

```
class Conversor5
{
  /**
   * Converte temperatura de Celsius para Fahrenheit.
   */
  double celsiusParaFahrenheit(double celsius)
  {
    return celsius * 9.0 / 5.0 + 32;
  }

  /**
   * Converte temperatura de Fahrenheit para Celsius.
   */
  double fahrenheitParaCelsius(double fahr)
```

```
    {
      return (fahr - 32.0) * 5.0 / 9.0;
    }

    /**
     * Imprime uma tabela de conversão Faranheit => Celsius.
     */
    void imprimeTabelaFahrenheitParaCelsius(double inicio, double fim)
    {
      double fahr = inicio;
      double celsius;

      while (fahr <= fim)
      {
        celsius = fahrenheitParaCelsius(fahr);
        System.out.println(fahr + "F = " + celsius + "C");
        fahr = fahr + 10.0;
      }
    }
  }
```

Muito melhor, não?

8.3 Números primos

Vejamos agora um novo exemplo. Todos devem se lembrar o que é um número primo: um número natural que possui exatamente dois divisores naturais distintos, o 1 e o próprio número. Vamos tentar escrever uma classe capaz de reconhecer e, futuramente, gerar números primos.

Como podemos reconhecer números primos? A própria definição nos dá um algoritmo. Dado um candidato a primo `x`, basta verificar se algum inteiro entre 2 e `x` - 1 divide `x`. Então para ver se um número é primo podemos usar um laço que verifica se a divisão exata ocorreu.

Porém, ainda falta um detalhe. Como podemos verificar se uma divisão entre números inteiros é exata. Já sabemos que se dividirmos dois números inteiros em Java a resposta é inteira. E o resto da divisão? Felizmente, há um operador especial que devolve o resto da divisão, é o operador `%`. Vejamos alguns exemplos:

```
Welcome to DrJava.
> 3 / 2
1
> 3 % 2
1
> 5 / 3
1
> 5 % 3
2
> int div = 7 / 5
> int resto = 7 % 5
> div
1
```

```
> resto
2
> div*5 + resto
7
>
```

Deu para pegar a idéia, não?

Agora vamos escrever uma classe contendo um método que verifica se um inteiro é primo ou não, imprimindo a resposta na tela. O nome que daremos à nossa classe é `GeradorDePrimos`. A razão para esse nome ficará clara no próximo capítulo.

```
class GeradorDePrimos
{
  /**
   * Imprime na tela se um número inteiro positivo é primo ou não.
   */
  void verificaPrimalidade(int x)
  {
    // Todos os números inteiros positivos são divisíveis por 1.
    int númeroDeDivisores = 1;
    // O primeiro candidato a divisor não trivial é o 2.
    int candidatoADivisor = 2;

    // Testa a divisão por todos os números menores ou iguais a x.
    while (candidatoADivisor <= x)
    {
      if (x % candidatoADivisor == 0)
        númeroDeDivisores = númeroDeDivisores + 1;
      candidatoADivisor = candidatoADivisor + 1;
    }

    // Imprime a resposta.
    if (númeroDeDivisores == 2)
      System.out.println(x + " é primo.");
    else
      System.out.println(x + " não é primo.");
  }
}
```

Será que esta é a forma mais eficiente de implementar este método? Será que podemos alterar o código para que ele forneça a resposta mais rapidamente? O que aconteceria se quiséssemos verificar a primalidade de 387563973. Pense um pouco sobre isso e depois dê uma olhada no exercício 6 deste capítulo.

Exercícios

1. Crie uma classe `Fatorial` com um método `calculaFatorial(int x)` que calcula o fatorial de `x` se este for um número inteiro positivo e devolve `-1` se `x` for negativo.

   Adicione o método `testaCalculaFatorial()` que testa o método `calculaFatorial(int x)` para diferentes valores de `x`.

2. Crie uma classe `Média` contendo um método `calculaMédia(int n)` que devolve a média dos valores 1, 2, 3, ..., n, onde n é o valor absoluto de um número fornecido ao método.

   Adicione o método `testaCalculaMédia()` que testa o método `calculaMédia(int n)` para diferentes valores de `n`.

3. Adicione as seguintes funcionalidades à classe `Conversor5` vista neste capítulo:

   (a) Crie o método `imprimeTabelaCelsiusParaFahrenheit`, que converte no sentido oposto do método `imprimeTabelaFahrenheitParaCelsius`.

   (b) Adicione um parâmetro aos métodos acima que permita a impressão de uma tabela com passos diferentes de 10.0. Ou seja, o passo entre a temperatura atual e a próxima será dado por esse novo parâmetro.

4. Escreva uma classe `Fibonacci`, com um método `imprimeNúmerosDeFibonacci (int quantidade)`, que imprime os primeiros `quantidade` números da seqüência de Fibonacci. A seqüência de Fibonacci é definida da seguinte forma.

   - $F_1 = 1$;
   - $F_2 = 1$;
   - $F_n = F_{n-1} + F_{n-2}$, para todo inteiro positivo $n > 2$.

   O método deve então imprimir $F_1, F_2, F_3, \ldots, F_{quantidade}$.

5. Abaixo, apresentamos uma pequena variação do método `verificaPrimalidade`. Ela não funciona corretamente em alguns casos. Você deve procurar um exemplo no qual esta versão não funciona e explicar o defeito usando suas próprias palavras. Note que a falha é sutil, o que serve como alerta: programar é uma tarefa difícil, na qual pequenos erros podem gerar resultados desastrosos. Toda atenção é pouca!

```
/**
 * Imprime na tela se um número inteiro positivo é primo ou não.
 */
void verificaPrimalidade(int x)
{
  // Todos os números inteiros positivos são divisíveis por 1.
  int númeroDeDivisores = 1;
  // O primeiro candidato a divisor não trivial é o 2.
  int candidatoADivisor = 2;

  // Testa a divisão por todos os números menores ou iguais a x.
  while (candidatoADivisor <= x)
  {
    candidatoADivisor = candidatoADivisor + 1;
    if (x % candidatoADivisor == 0)
      númeroDeDivisores = númeroDeDivisores + 1;
  }

  // Imprime a resposta.
  if (númeroDeDivisores == 2)
```

```
      System.out.println(x + " é primo.");
    else
      System.out.println(x + " não é primo.");
  }
```

6. O laço no nosso `verificaPrimalidade` é executado mais vezes do que o necessário. Na verdade poderíamos parar assim que `candidatoADivisor` chegar a `x/2` ou mesmo ao chegar à raiz quadrada de `x`. Pense como mudar o programa levando em consideração estes novos limitantes.

7. Escreva uma classe Euclides, com um método `mdc` que recebe dois números inteiros `a1` e `a2`, estritamente positivos, com `a1 >= a2`, e devolve o máximo divisor comum entre eles, utilizando o algoritmo de Euclides.

   Breve descrição do algoritmo de Euclides (para maiores detalhes, consulte seu professor de Álgebra):

   - Dados `a1` e `a2`, com `a1` >= `a2`, quero o m.d.c.(`a1`, `a2`).
   - Calcule `a3` = `a1` % `a2`.
   - Se `a3` = 0, fim. A solução é `a2`.
   - Calcule `a4` = `a2` % `a3`.
   - Se `a4` = 0, fim. A solução é `a3`.
   - Calcule `a5` = `a3` % `a4`.
   - ...

   Nota importante: o operador binário % calcula o resto da divisão de n por m, quando utilizado da seguinte maneira: n % m. Curiosidade: ele também funciona com números negativos! Consulte seu professor de Álgebra ;-)

Capítulo 9

Expressões e Variáveis Lógicas

Quais novidades veremos neste capítulo?

- Condições como expressões lógicas;
- Variáveis booleanas;
- Condições compostas e operadores lógicos: `&&`, `||` e `!`;
- Precedência de operadores.

9.1 Condições como expressões

Já vimos que em Java e outras linguagens de programação, as condições exercem um papel fundamental. São elas que permitem que diferentes ações sejam tomadas de acordo com o contexto. Isso é feito através dos comandos `if` e `while`.

Mas, o que são condições realmente? Vimos apenas que elas consistem geralmente em comparações, usando os operadores `==`, `>=`, `<=`, `>`, `<` e `!=`, entre variáveis e/ou constantes. Uma característica interessante em linguagens de programação é que as condições são na verdades expressões que resultam em verdadeiro ou falso. Vamos ver isso no DrJava:

```
Welcome to DrJava.
> 2 > 3
false
> 3 > 2
true
> int a = 2
> a == 2
true
> a >= 2
true
```

```
> a < a + 1
true
>
```

Vejam que cada vez que digitamos uma condição o DrJava responde `true` (para verdadeiro) ou `false` (para falso).

Para entender bem o que ocorre, é melhor imaginar que em Java as condições são expressões que resultam em um dos dois valores lógicos: "verdadeiro" ou "falso". Neste sentido, Java também permite o uso de variáveis para guardar os resultados destas contas, como vemos abaixo.

```
> boolean comp1 = 2 > 3
> comp1
false
> int a = 3
> boolean comp2 = a < a + 1
> comp2
true
>
```

Com isso, acabamos de introduzir mais um tipo de variável, somando-se aos tipos `int` e `double` já conhecidos: o tipo `boolean`, que é usado em variáveis que visam conter apenas os valores booleanos (verdadeiro ou falso). O nome é uma homenagem ao matemático inglês George Boole (1815-1864). Em português este tipo de variável é chamada de variável *booleana*

Agora que começamos a ver as comparações como expressões que calculam valores booleanos, torna-se mais natural a introdução dos operadores lógicos. Nós todos já estamos bem acostumados a condições compostas. Algo como "eu só vou à praia se tiver sol *e* as ondas estiverem boas". Nesta sentença a conjunção *e* une as duas condições em uma nova condição composta que é verdadeira somente se as duas condições que a formam forem verdadeiras.

Em Java o "*e*" lógico é representado pelo estranho símbolo `&&`. Ou seja, uma condição do tipo `1 <= a <= 10` seria escrita em Java como `a >= 1 && a <= 10`. Da mesma forma temos um símbolo para o *ou* lógico. Ele é o símbolo `||`. Isso mesmo, duas barras verticais. Por fim, o símbolo `!` antes de uma expressão lógica nega o seu valor. Por exemplo, a condição "a não é igual a 0"poderia ser escrita em Java como `!(a == 0)`[1].

Tabelas da verdade contém todos os resultados que podem ser obtidos ao se aplicar uma operação lógica sobre variáveis booleanas. Abaixo apresentamos as tabelas da verdade para os operadores `&&`, `||` e `!`.

`&&` (*e*)	true	false
true	true	false
false	false	false

`||` (*ou*)	true	false
true	true	true
false	true	false

`!` (*não*)	true	false
	false	true

[1] Daí vem a explicação para o fato do sinal de diferente conter o ponto de exclamação.

Por fim, podemos montar expressões compostas unindo, através dos operadores descritos acima, condições simples ou respostas de expressões lógicas anteriores que foram armazenadas em variáveis booleanas. Mais uma vez um exemplo vale mais que mil palavras.

```
Welcome to DrJava.
> (2 > 3) || (2 > 1)
true
> boolean comp1 = 2 > 3
> comp1
false
> comp1 && (5 > 0)
false
> !(comp1 && (5 > 0))
true
> int a = 10
> (a > 5) && (!comp1)
true
> boolean comp2 = (a > 5) && comp1
> comp2
false
>
```

Também podemos "misturar" operadores aritméticos e comparadores, sempre que isso faça sentido. Por exemplo,

```
> (a - 10) > 5
false
> a - (10 > 5)
koala.dynamicjava.interpreter.error.ExecutionError: Bad type in subtraction
>
```

Note que a última expressão resultou em um erro. Afinal de contas, ela pede para somar, a uma variável inteira, o resultado de uma expressão cujo valor é booleano, misturando tipos. Isto não faria sentido em Java.

Outra coisa que pode ser feita é a criação de métodos que devolvem um valor booleano. Assim a resposta dada por esses métodos pode ser usada em qualquer lugar onde uma condição faça sentido, como um `if` ou um `while`. Por exemplo, se alterarmos o método `verificaPrimalidade`, dado no capítulo anterior, para devolver a resposta (se o número é primo ou não), em vez de imprimir na tela, teríamos o seguinte método `éPrimo`:

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean éPrimo(int x)
{
  if (x < 2)
    return false;

  // Todos os números inteiros positivos são divisíveis por 1.
```

```
    int númeroDeDivisores = 1;
    // O primeiro candidato a divisor não trivial é o 2.
    int candidatoADivisor = 2;

    // Testa a divisão por todos os números menores ou iguais a x.
    while (candidatoADivisor <= x)
    {
      if (x % candidatoADivisor == 0)
        númeroDeDivisores = númeroDeDivisores + 1;
      candidatoADivisor = candidatoADivisor + 1;
    }

    if (númeroDeDivisores == 2)
      return true;
    else
      return false;
  }
```

Note que, desta forma, podemos escrever algo do tipo:

```
if (!éPrimo (x))
   // faça alguma coisa
```

A construção acima deixa muito claro o significado da expressão, pois a lemos como "se não é primo x", o que é bem próximo do que seria uma frase falada em português: "se x não é primo". Quanto mais próximo for o seu código da linguagem falada, mais fácil será para outras pessoas o compreenderem; e a clareza do código é um dos principais objetivos do bom programador.

9.2 Precedência de operadores

Como acabamos de apresentar vários operadores novos, devemos estabelecer a precedência entre eles. Lembre-se que já conhecemos as regras de precedência dos operadores aritméticos há muito tempo. Já a precedência dos operadores lógicos é coisa nova. A tabela abaixo apresenta os operadores já vistos, listados da precedência mais alta (aquilo que deve ser executado antes) à mais baixa:

operadores unários	- !
operadores multiplicativos	* / %
operadores aditivos	+ -
operadores de comparação	== != > < >= <=
"*e*" lógico	&&
"*ou*" lógico	||
atribuição	=

Entre operadores com mesma precedência, as operações são computadas da esquerda para a direita.

Note, porém, que, nos exemplos acima, abusamos dos parênteses mesmo quando, de acordo com a tabela de precedência, eles são desnecessários. Sempre é bom usar parênteses no caso de expressões lógicas (ou mistas), pois a maioria das pessoas não consegue decorar a tabela acima. Assim, mesmo que você tenha uma ótima memória, o seu código torna-se mais legível para a maioria dos mortais.

9.3 Exemplos

Primeiro, vamos retomar o método `verificaLados` da classe `TianguloRetângulo3` vista no Capítulo 6. Nele, testamos se não há lado de comprimento nulo. Entretanto, parece mais natural e correto forçar todos os lados a terem comprimento estritamente positivo:

```
if ((a > 0) && (b > 0) && (c > 0))
{
  // Aqui vão os comandos para verificar a condição pitagórica.
}
```

Podemos também usar condições compostas para escrever uma versão mais rápida do método `éPrimo` do capítulo anterior.

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean éPrimo(int x)
{
  if (x < 2)
    return false;

  // Todos os números inteiros positivos são divisíveis por 1.
  int númeroDeDivisores = 1;
  // O primeiro candidato a divisor não trivial é o 2.
  int candidatoADivisor = 2;

  // Testa a divisão por todos os números menores ou iguais a x/2 ou
  // até encontrar o primeiro divisor.
  while ((candidatoADivisor <= x/2) && (númeroDeDivisores == 1))
  {
    if (x % candidatoADivisor == 0)
      númeroDeDivisores = númeroDeDivisores + 1;
    candidatoADivisor = candidatoADivisor + 1;
  }

  if (númeroDeDivisores == 1)
    return true;
  else
    return false;
}
```

Melhor ainda podemos finalmente escrever a classe `GeradorDePrimos` de forma completa. O método mais interessante é o `próximoPrimo` que devolve o primeiro número primo maior do que o último gerado. Este exemplo já é bem sofisticado, vocês terão que estudá-lo com calma. Uma sugestão: tentem entender o que o programa faz, um método por vez. O único método mais complicado é o `éPrimo`, mas este nós já vimos.

```
class GeradorDePrimos
{
  // Limite inferior para busca de um novo primo.
  int limiteInferior = 1;

  /**
```

```
    * Permite mudar o limite para cômputo do próximo primo.
    */
   void carregaLimiteInferior(int limite)
   {
     limiteInferior = limite;
   }

  /**
    * Verifica se um número inteiro positivo é primo ou não.
    */
   boolean éPrimo(int x)
   {
     if (x < 2)
       return false;

     // Todos os números inteiros positivos são divisíveis por 1.
     int númeroDeDivisores = 1;
     // O primeiro candidato a divisor não trivial é o 2.
     int candidatoADivisor = 2;

     // Testa a divisão por todos os números menores ou iguais a x/2 ou
     // até encontrar o primeiro divisor.
     while ((candidatoADivisor <= x/2) && (númeroDeDivisores == 1))
     {
       if (x % candidatoADivisor == 0)
         númeroDeDivisores = númeroDeDivisores + 1;
       candidatoADivisor = candidatoADivisor + 1;
     }

     if (númeroDeDivisores == 1)
       return true;
     else
       return false;
   }

   /**
    * A cada chamada, encontra um novo primo maior que limiteInferior.
    */
   int próximoPrimo()
   {
     // Busca o primeiro primo depois do limite.
     limiteInferior = limiteInferior + 1;
     while (!éPrimo(limiteInferior))
       limiteInferior = limiteInferior + 1;

     return limiteInferior;
   }
 }
```

Note o uso do atributo `limiteInferior` no exemplo acima. Ele guarda uma informação importante: o valor do último número primo encontrado. É importante que ele seja um atributo dos objetos do tipo `GeradorDePrimos` e não uma variável local do método `próximoPrimo()` para que o seu valor sobreviva às sucessivas chamadas ao método `próximoPrimo()`. Se ele fosse uma variável local, a cada nova execução do

método, seu valor seria zerado.

Não deixem de brincar um pouco com objetos da classe `GeradorDePrimos` para entender melhor como ela funciona! Agora um desafio para vocês: usando o método `próximoPrimo`, escrevam um novo método `void imprimePrimos (int quantidade)` que imprime uma dada quantidade de números primos a partir do `limiteInferior`. Experimentem executar o método no DrJava passando 50 como parâmetro.

Exercícios

1. Escreva uma classe `TrianguloRetângulo` com um método denominado `defineLados(double x1, double x2, double x3)` que recebe três valores e verifica se eles correspondem aos lados de um triângulo retângulo. Em caso afirmativo, o método devolve `true`, caso contrário ele devolve `false`. Note que o programa deve verificar quais dos três valores corresponde à hipotenusa. Construa duas versões do método, uma contendo três `if`s e outra contendo apenas um `if`! Em seguida, crie um novo método que verifica se ambos os métodos retornam o mesmo resultado para diferentes combinações de `x1`, `x2` e `x3`.

2. Escreva uma classe `Brincadeiras` que possua 3 atributos inteiros. Escreva um método para carregar valores nestes atributos e, em seguida, escreva os seguintes métodos:

   (a) `troca2Primeiros()` que troca os valores dos dois primeiros atributos. Por exemplo, se antes da chamada do método o valor dos atributos é <1, 2, 3>, depois da chamada, eles deverão valer <2, 1, 3>.

   (b) `imprime()` que imprime o valor dos 3 atributos.

   (c) `imprimeEmOrdemCrescente()` que imprime o valor dos 3 atributos em ordem crescente.

3. A linguagem Java oferece operadores que, se usados corretamente, ajudam na apresentação e digitação do código, tornando-o mais enxuto. Veremos neste exercício dois deles: os operadores de incremento e de decremento. Verifique o funcionamento desses operadores usando os métodos da classe abaixo.

```
class Experiência
{
  void verIncremento(int n)
  {
    int x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("x++              -> " + x++);
    System.out.println("Novo valor de x  -> " + x);

    x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("++x              -> " + ++x);
    System.out.println("Novo valor de x  -> " + x);
  }
  void verDecremento(int n)
  {
    int x = n;
    System.out.println("Número inicial x -> " + x);
    System.out.println("x--              -> " + x--);
    System.out.println("Novo valor de x  -> " + x);
```

```
        x = n;
        System.out.println("Número inicial x -> " + x);
        System.out.println("--x              -> " + --x);
        System.out.println("Novo valor de x  -> " + x);
    }
}
```

Entenda bem o código e observe os resultados. Em seguida, tire suas conclusões e compare-as com as conclusões de seus colegas.

Além dos operadores de incremento e decremento, também existem os seguintes operadores resumidos: `+=`, `-=`, `*=` e `/=`. Eles são úteis quando queremos efetuar uma operação em uma variável e guardar o resultado na mesma. Isto é, `a = a * 2;` é completamente equivalente a `a *= 2;`.

Capítulo 10

Mergulhando no `while`

Quais novidades veremos neste capítulo?

- Reforço em `while`;
- O comando `do...while`.

10.1 Um pouco mais sobre primos

Vamos iniciar este capítulo com dois exercícios. Primeiro, que tal modificarmos o método de geração de primos para que ele use o fato de que os únicos candidatos a primos maiores do que 2 são ímpares? Uma complicação interessante é que o `limiteInferior` para o próximo primo pode ser modificado pelo usuário a qualquer momento chamando `carregaLimiteInferior`. Isso deve ser contemplado na solução. Aqui vai a resposta:

```
/**
 * A cada chamada, calcula um novo primo seguindo ordem crescente.
 */
int próximoPrimo()
{
  // Move o limite inferior na direção do próximo primo.
  // Temos que considerar que o limite inferior pode ser par
  // porque ele pode ser modificado a qualquer momento com uma
  // chamada a carregaLimiteInferior.
  if (limiteInferior == 1)
    limiteInferior = 2;
  else if (limiteInferior % 2 == 0)
    limiteInferior = limiteInferior + 1;
  else
    limiteInferior = limiteInferior + 2;

  // Encontra o próximo primo
  while (!éPrimo(limiteInferior))
    limiteInferior = limiteInferior + 2;
```

```
    return limiteInferior;
  }
```

Nosso próximo desafio é criar uma nova classe `ManipuladorDeInteiros`. Ela deve conter o método `fatoraInteiro` que deve imprimir a decomposição em primos de um inteiro positivo maior ou igual a 2. Uma dica importante é usar o `GeradorDePrimos` Antes de ler solução colocada abaixo, tente com afinco fazer o exercício sozinho.

```
  class ManipuladorDeInteiros {
    /**
     * Fatora em primos um inteiro > 1.
     */
    void fatoraInteiro(int x)
    {
      System.out.print(x + " =");

      // Usa um gerador de primos para encontrar os primos menores ou iguais a x.
      GeradorDePrimos gerador = new GeradorDePrimos();
      int primo = gerador.próximoPrimo();

      // Continua fatorando o número até que x se torne 1.
      while (x > 1)
      {
        if (x % primo == 0)
        {
          System.out.print(" " + primo);
          x = x / primo;
        }
        else
          primo = gerador.próximoPrimo();
      }

      // Imprime um fim de linha no final.
      System.out.println();
    }
  }
```

Um exemplo de uso do nosso `ManipuladorDeInteiros`:

```
Welcome to DrJava.
> ManipuladorDeInteiros m = new ManipuladorDeInteiros()
> m.fatoraInteiro(5)
5 = 5
> m.fatoraInteiro(10)
10 = 2 5
> m.fatoraInteiro(18)
18 = 2 3 3
> m.fatoraInteiro(123456)
123456 = 2 2 2 2 2 2 3 643
> m.fatoraInteiro(12345678)
```

```
12345678 = 2 3 3 47 14593
> m.fatoraInteiro(167890)
167890 = 2 5 103 163
>
```

Obs: Note que na solução usamos uma rotina de impressão nova: `System.out.print`. Ela é muito parecida com `System.out.println` com a diferença de que não muda a linha ao terminar de imprimir.

10.2 Uma biblioteca de funções matemáticas.

Terminamos com um exercício clássico. Vamos mostrar como construir uma pequena biblioteca de funções matemáticas avançadas. Com será que o computador consegue calcular senos, cossenos, logaritmos? O segredo para implementar essas funções em Java é um bom conhecimento de cálculo e laços.

Usando cálculo, sabemos que essas funções matemáticas "complicadas" possuem expansões de Taylor. Estas expansões transformam uma função numa série de polinômios[1], que podem ser facilmente calculados usando laços. Vejamos a expansão de algumas dessas funções:

- $sen(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^k x^{(2k+1)}}{(2k+1)!} + \ldots$
- $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + \frac{(-1)^k x^{(2k)}}{(2k)!} + \ldots$
- $ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + \frac{(-1)^{(k-1)} x^k}{k} + \ldots$

  Isso funciona bem sempre que $|x| < 1$.

O segredo para usar essas fórmulas no computador é continuar somando até que o módulo do próximo termo seja muito pequeno e por isso possa ser desprezado.

Antes de apresentarmos aqui a solução que consideramos ideal, faça com cuidado e atenção os exercícios 1 (implementação das funções `double pot(double x, int y)` e `double fat(double x)`) e 2 (implementação da função `double sen(double x)` usando as funções do exercício 1.

Agora, após termos feito os exercícios 1 e 2, iremos criar uma nova classe, que chamaremos `Matemática`, com métodos para calcular funções como as apresentadas acima. Abaixo vemos a classe com uma função que calcula $sen(x)$ implementada. Compare esta forma de implementar com as formas usadas no exercício. Qual é mais rápida? Qual é mais fácil de entender?

```
class Matemática
{
  // Controla o significado de "pequeno".
  double epsilon = 1.0e-8;

  double sen(double x)
  {
    int k = 1;
    double termo = x;
    double seno = termo;
```

[1] Mais informações sobre séries de Taylor podem ser encontradas em livros de Cálculo.

```
    while (termo*termo > epsilon*epsilon)
    {
      // É muito mais fácil construir o próximo termo usando o anterior.
      k = k + 2;
      termo = -termo * x * x / (k - 1) / k;
      seno = seno + termo;
    }
    return seno;
  }
}
```

Um exemplo de uso:

```
Welcome to DrJava.
> m = new Matemática();
> m.sen(0.3)
0.2955202066613839
> m.sen(0.5)
0.4794255386164159
> m.sen(3.141596/2.0)
0.9999999999925767
>
```

10.3 do...while

Para complementar os laços possíveis em Java, vejamos uma pequena variação do `while`. Nele a condição é testada sempre antes de execução do corpo de comandos que compõe o laço. Já o laço `do...while` tem a condição testada apenas no final. Conseqüentemente, no caso do `do...while`, existe a garantia que o conteúdo no interior do laço será executado pelo menos uma vez, enquanto no `while` este pode nunca ser executado. Na prática, a existência destes dois tipos de laços é uma mera conveniência sintática, já que um pode ser facilmente substituído pelo outro.

Vejamos um exemplo de utilização do `do...while`:

```
int fatorial(int x)
{
  int resultado = 1;
  do
  {
    resultado = resultado * x;
    x = x - 1;
  } while (x > 1)
  return resultado;
}
```

Exercícios

1. Implemente na classe `Matemática` as funções `double pot(double x, int y)` que calcula x^y e `double fat(double x)` que calcula $x!$ [2].

2. Implemente na classe `Matemática` a função `double sen(double x)` utilizando-se das funções `double pot(double x, int y)` e `int fat(int x)` do ítem anterior.

3. Implemente na classe `Matemática` funções para calcular $cos(x)$ e $ln(1+x)$. Note que para implementar o $ln(1+x)$ deve-se criar uma função `double ln(double x)` e no interior da função definir uma variável local `x2 = 1 - x` de modo que se possa calcular $ln(1+x2)$.

4. Escreva uma classe `TestaMatemática` que utiliza os métodos matemáticos `java.lang.Math.sen()`, `java.lang.Math.cos()`, `java.lang.Math.pow()` e `java.lang.Math.log()`, disponíveis na biblioteca Java, para testar os respectivos métodos implementado por você nos exercícios anteriores. Os métodos de `TestaMatemática` devem receber um parâmetro `double epsilon` que determina a diferença máxima aceitável entre o valor devolvido pela sua implementação com relação ao da implementação da biblioteca Java. Qualquer dúvida sobre a utilização destes métodos, consulte a documentação online do Java[3].

5. O enunciado deste exercício é bem mais complexo que a solução, por isso não tenha medo! Imagine um quadrado em um plano e uma reta paralela a um dos lados do quadrado: a projeção do quadrado sobre a reta tem exatamente o mesmo comprimento que o lado do quadrado. Imagine agora que este quadrado seja girado sobre o plano; a projeção do quadrado sobre a reta tem um novo tamanho. Crie uma classe Projetor que possua um método gira que aceite como parâmetro o número de graus que o quadrado deve girar em relação à sua posição anterior e imprima na tela o tamanho da projeção do quadrado sobre a reta. Note que se o usuário executar o método duas vezes, com os parâmetros "22" e "35", sua classe deve responder qual o tamanho da projeção para inclinações do quadrado de 22 e 57 graus.

   - Escreva 3 soluções para este exercício: uma que você considere elegante e clara, uma com um único método e uma com o máximo número possível de métodos. Utilize os métodos `sen()` e `cos()` desenvolvidos neste capítulo.
   - Utilize agora os métodos `java.lang.Math.cos()` e `java.lang.Math.sin()` disponíveis na biblioteca Java, que calculam, respectivamente, o coseno e o seno do ângulo passado como parâmetro em radianos (graus * PI/180 = radianos). Compare os resultados com os obtidos com nossas implementações de `sen()` e `cos()`.

[2] Apesar do cálculo de fatorial só usar inteiros, com o tipo `int` pode se calcular até 16!, e com o tipo `long` até 20!. Com o tipo `double` é possível calcular fatoriais maiores, mas com menos dígitos significativos.

[3] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html`.

Capítulo 11

Caracteres e Cadeias de Caracteres

Quais novidades veremos neste capítulo?

- Introdução do tipo `char`;
- Uma classe da biblioteca padrão: `String`.

11.1 Um tipo para representar caracteres

Até o momento já vimos diferentes tipos de variáveis, como os inteiros (`int`) e os reais (`double`). Além disto, também vimos as variáveis booleanas, que podem ter apenas dois valores, verdadeiro ou falso (`boolean`). Parece intuitivo que as linguagens de programação também ofereçam variáveis para a manipulação de caracteres. No caso de Java temos o tipo `char`. Vejamos um exemplo de uso:

```
class Caractere1
{
  void verificaResposta(char ch)
  {
    if ((ch == 's') || (ch =='S'))
      System.out.println("A resposta foi sim");
    else if ((ch == 'n') || (ch == 'N'))
      System.out.println("A resposta foi não");
    else
      System.out.println("Resposta inválida");
  }
}
```

No exemplo acima podemos ver que para se representar um caractere usamos aspas simples (’). Também podemos ver que os caracteres minúsculos são diferentes do mesmo caractere maiúsculo.

Um outro exemplo um pouco mais elaborado pode ser visto abaixo:

```
class Caracteres
{
```

```
  void imprimeCaracteres(char ch, int n)
  {
    int i = 0;
    while (i < n)
    {
      System.out.print(ch);
      i = i + 1;
    }
  }
}
```

Neste exemplo, são impressos diversos caracteres do mesmo tipo. Observe abaixo como podemos adicionar um novo método para desenhar letras grandes:

```
class Caracteres
{

  void imprimeCaracteres(char ch, int n)
  {
    int i = 0;
    while (i < n)
    {
      System.out.print(ch);
      i = i + 1;
    }
  }
  void novaLinha()
  {
    System.out.println();
  }
  void imprimeCaracteresNL(char ch, int n)
  {
    imprimeCaracteres(ch, n);
    novaLinha();
  }
  void desenhaE()
  {
    imprimeCaracteresNL('*',20);
    imprimeCaracteresNL('E', 15);
    imprimeCaracteresNL('E', 14);
    imprimeCaracteresNL('E', 3);
    imprimeCaracteresNL('E', 3);
    imprimeCaracteresNL('E', 13);
    imprimeCaracteresNL('E', 13);
    imprimeCaracteresNL('E', 3);
    imprimeCaracteresNL('E', 3);
    imprimeCaracteresNL('E', 14);
    imprimeCaracteresNL('E', 15);
    imprimeCaracteresNL('*',20);
  }
}
```

A saída do programa é:

```
********************
EEEEEEEEEEEEEEE
EEEEEEEEEEEEEE
EEE
EEE
EEEEEEEEEEEEE
EEEEEEEEEEEEE
EEE
EEE
EEEEEEEEEEEEEE
EEEEEEEEEEEEEEE
**********************
```

Para desenharmos letras onde é necessário intercalar espaços e letras em uma única linha, a implementação fica um pouco mais longa. Os diferentes métodos que imprimem caracteres pulando ou sem pular linha serão usados. Por exemplo, para a letra U, a primeira linha deve ser impressa como:

```
imprimeCaracteres('U', 3);
imprimeCaracteres(' ', 9);  // espaço também é um caractere
imprimeCaracteresNL('U", 3);
```

11.2 Cadeias de caracteres (`Strings`)

Uma forma de escrevermos palavras no computador seria usando grupos de caracteres, entretanto isto depende de um conceito mais avançado que ainda não vimos. A nossa outra opção é usar uma classe pronta, que já vem com a linguagem Java, a classe `String`.

Nos nossos primeiros exemplos, já havíamos feito algumas operações com strings, por exemplo:

```
System.out.println("O triângulo é retângulo");
System.out.println("A raiz de " + 4 + " é igual a " + 2 + ".");
```

Agora nós veremos com mais detalhes esta classe `String` . Podemos ver que o operador `+` tem um significado natural o de concatenação. Logo, as seguintes operações são válidas:

```
String a = "abc";
String b = "cdf";
String c;

c = a + b;
System.out.println(c);
```

Podemos também concatenar números a uma `String`:

```
String a = "O resultado é";
int i = 10;
String c;
```

```
c = a + i;
System.out.println(c);
```

Além disto, existem alguns métodos pré-definidos na classe `String`. Entre eles podemos citar:

- `char charAt(int index)` - devolve o caractere na posição `index`.
  Os índices em uma `String` vão de zero ao seu tamanho menos um. Exemplo:

```
String s = "mesa";
System.out.println(s.charAt(0)); // Imprime  m
System.out.println(s.charAt(3)); // Imprime  a
```

- `boolean endsWith(String suffix)` - verifica se a `String` acaba com o sufixo dado. Usado, entre outras coisas, para verificar as extensões dos arquivos. Por exemplo, verificar se o nome de um arquivo termina com `".java"`.

- `int indexOf(char ch)` - devolve o índice da primeira ocorrência de `ch` na `String` ou -1 caso o caractere não ocorra na `String`. Exemplo:

```
String s1 = "EP1.java";
System.out.println(s1.indexOf('.')); // imprime 3
System.out.println(s1.indexOf('x')); // imprime -1
```

- `int length()` - devolve o tamanho da `String`. Exemplo:

```
String s1 = "mesa";
System.out.println(s1.length()); // imprime 4
```

- `String toUpperCase()` - devolve a `String` convertida para letras maiúsculas. Exemplo:

```
String s1 = "mesa";
System.out.println(s1.toUpperCase()); // imprime MESA
```

- `int compareTo(String outra)` - compara duas `String`s.
  Devolve um número positivo se a `outra` for menor, 0 se forem iguais, e um negativo caso contrário. Exemplo:

```
String s1 = "mesa";
String s2 = "cadeira";
System.out.println(s1.compareTo(s2)); // imprime 10 que é > 0
```

Exercícios

1. Escreva uma classe `Linha` que possua um método `imprimeLinha` que, ao ser chamado, imprime uma linha de caracteres `X` na diagonal, na tela de interações do DrJava. Use laços `while`. DICA: você vai precisar do método `System.out.print()`, que imprime seu argumento na tela mas não passa para a linha seguinte; imprima linhas com número crescente de espaços no começo e o caractere `X` no final.

2. Escreva uma classe contendo um método que devolve o número de ocorrências da vogal `a` em uma frase contida em uma `String`. Crie um teste para verificar o funcionamento desta classe.

3. Implemente uma classe com um método que determina a freqüência relativa de vogais em uma `String`. Considere que as letras maiúsculas e minúsculas não estão acentuadas. Crie um teste para verificar o funcionamento desta classe.

Capítulo 12

A Memória e as Variáveis

Quais novidades veremos neste capítulo?

- Organização da Memória do Computador;
- O que são variáveis.

12.1 A Memória do Computador

Nos computadores modernos, a memória armazena tanto os programas que estão sendo executados quanto os dados por eles manipulados. Grosso modo, podemos dividir a memória em principal e secundária. Memória principal é aquela normalmente armazenada em *chips* conectados à placa mãe do computador; o acesso aos seus dados é muito rápido mas, em geral, seus dados são perdidos quando o computador é desligado. Já a memória secundária é armazenada em discos rígidos ou flexíveis, CDs, DVDs, etc. O acesso aos seus dados é mais lento mas eles são armazenados de forma persistente, ou seja, não necessitam de energia elétrica para serem mantidos. Os dados gravados em um CD, por exemplo, mantêm-se inalterados por vários anos até que a degradação do material comece a causar perda de informações.

Quando ativamos um ambiente de desenvolvimento de software como o DrJava ou o Eclipse em nosso computador, o programa do ambiente que estava gravado no disco, é lido e copiado para a memória principal do computador. A partir de então, as informações que digitamos no ambiente são armazenadas na memória do computador como dados. Quando selecionamos a opção "Salvar", os dados da memória principal são copiados para o disco.

Podemos visualizar abstratamente a memória do computador como sendo uma rua muito longa com milhões de casas. O endereço de cada casa é dado pelo número da rua no qual ela se encontra e cada casa armazena um certo conteúdo. Nos computadores modernos, em cada endereço de memória é armazenado um *byte* formado por 8 *bits*. O computador utiliza internamente números binários e com 8 bits consegue armazenar número de 0 a 255 (ou de 00000000 até 11111111 em binário). Combinando duas casas (bytes) adjacentes, o computador consegue armazenar números de 16 bits, ou de 0 a 32767. Um `int` em Java ocupa 4 bytes e usa uma codificação que é capaz de representar números de -2.147.483.648 até 2.147.483.647. Utilizando mais bytes adjacentes e

outras formas de codificação, é possível armazenar na memória qualquer tipo de informação que podemos imaginar: números positivos e negativos inteiros e fracionários, letras, textos, imagens, sons, vídeo, etc.

Quando dizemos que um computador tem 1GB de memória (1 Giga Byte), queremos dizer que sua memória é composta por $2^{30} = 1.073.741.824$ bytes. Podemos então imaginá-la como sendo uma rua com pouco mais de 1 bilhão de casas (comprida né?), cada uma composta por um byte. Nestes bytes, o computador armazena os programas sendo executados, o código-fonte dos programas Java que digitamos no DrJava, os bytecodes correspondentes às classes Java depois que nós as compilamos, o conteúdo das variáveis que fazem parte de nossos objetos, etc.

12.2 O que são as Variáveis?

Variáveis são nomes associados a dados em um programa de computador. No caso de Java, isso inclui os atributos de objetos e de classes, parâmetros de métodos e variáveis locais. Quando um programa é executado, cada variável é associada, em tempo de execução, a um determinado endereço da memória. Se definimos uma variável `i` do tipo `int`, ela pode ser associada, por exemplo, ao endereço de memória 3.745.908. A partir daí, quando atribuímos um valor a esta variável, este valor é armazenado nos quatro bytes (porque um `int` Java ocupa 4 bytes) a partir do endereço 3.745.908 da memória.

Em Java, uma variável associada a um tipo primitivo (`int`, `float`, `double`, `boolean` e `char`) tem um comportamento levemente diferente de variáveis associadas a objetos complexos como Strings, arrays ou objetos pertencentes a classes definidas pelo programador. Os dados de tipo primitivo são todos muito simples, ocupando pouco espaço de memória. Neste caso, é fácil guardar nas posições de memória correspondentes a estas variáveis o seu valor. Logo, se `a` e `b` são variáveis de mesmo tipo primitivo, a operação `a = b` copia em `a` o valor que está em `b`, ou seja, copia o conteúdo dos bytes da memória correspondentes à variável `b` para os bytes da memória correspondentes à variável `a`. Dizemos que as variáveis `a` e `b` armazenam diretamente o seu conteúdo.

Isso explica por que a mudança em um parâmetro de tipo primitivo dentro de um método não se reflete fora dele. O parâmetro alterado dentro do método não tem nenhuma ligação com a variável que estava em seu lugar no momento da chamada do método (além, é claro, da cópia inicial do seu valor quando o método é chamado).

Já para objetos complexos, como cadeias de caracteres, arrays e qualquer objeto de uma classe definida pelo programador, a situação é bem diferente. Estes objetos podem ser muito complicados, ocupando muito espaço de memória. Neste caso, o Java não guarda nas variáveis uma cópia do objeto, mas sim o endereço de memória no qual este objeto está armazenado. Por este motivo, fala-se que em Java as variáveis associadas a objetos complexos são *referências* para estes objetos. No momento em que um objeto precisa ser usado, o interpretador Java o localiza a partir de uma referência para ele armazenada em uma variável que, por sua vez, está também armazenada em um endereço de memória. Esta pequena diferença gera algumas surpresas. Considere que temos uma classe chamada `Fruta` com um atributo `preço` e métodos `carregaPreço` e `imprimePreço`. O que ocorre quando executamos o seguinte código?

```
Fruta a = new Fruta();
a.carregaPreço(10);
Fruta b = a;
b.carregaPreço(100);
a.imprimePreço();
```

Observamos que o preço da fruta `a` mudou. Isto ocorre porque, após a terceira linha do trecho acima, `a` e `b` passam a ser referências para o mesmo objeto na memória; como dois apelidos para mesma pessoa. Qualquer coisa que ocorre com o objeto usando um de seus apelidos se reflete para a visão do outro apelido. Esse raciocínio explica porque alterações em objetos complexos dentro de um método se propagam para fora. O parâmetro associado a esse objeto também é um apelido.

A diferença entre tipos primitivos e objetos também fica evidente quando lidamos com comparações. Quando se comparam dois tipos primitivos com o operador ==, os valores dos mesmos são verificados. O mesmo não acontece com objetos, o que se comparam são as referências. Veja o exemplo abaixo:

```
> int a = 5
> int b = 2 + 3
> a == b
true
> String s1 = "Bom dia"
> String s2 = "Bom"
> s2 = s2 + " " + "dia"
> s1 == s2
false
```

Vemos no exemplo acima que o operador == tem um significado diferente com relação a objetos. A comparação de igualdade verifica se as duas referências se referem ao mesmo objeto. Na última linha do exemplo acima, tanto `s1` quanto `s2` se referem ao texto `"Bom dia"` mas as variáveis `s1` e `s2` são referências a posições diferentes da memória, se referem a dois objetos distintos; portanto, `s1 == s2` é `false`.

Exercícios

1. O comparador == não pode ser usado para comparar o conteúdo de objetos. Neste caso, usa-se o método `equals`, presente em todos os objetos. Este método recebe como entrada uma outra `String` e retorna `true` se as `String`s são iguais e `false` caso contrário. Faça uma classe com um método que verifica se duas `String`s têm o mesmo conteúdo.

2. Vimos neste capítulo que a mudanças no valor de um parâmetro de tipo primitivo dentro de um método não se reflete fora dele. Mas no caso de objetos, alterações em seu conteúdo dentro de um método são refletidas fora deste.

   Suponha que você deseje criar um método que modifica os valores de um par de inteiros de modo que estas modificações se reflitam fora do métodos. Um bom exemplo é um método que recebe como entrada as coordenadas de ponto cartesiano e desloca este ponto em (+1,+1). Pense numa possível solução para este problema. Em seguida, implemente sua solução e verifique se ela funciona.

3. Verifique o seguinte trecho de código:

   ```
   > String s1 = "Bom dia."
   > String s2 = s1
   > s1 = "Boa noite."
   ```

Qual o valor da variável `s2` após a execução deste trecho de código? Pense cuidadosamente na resposta e em seguinte verifique-a utilizando o DrJava. Tente explicar o comportamento verificado.

Capítulo 13

Manipulando Números Utilizando Diferentes Bases

Quais novidades veremos neste capítulo?

- Como extrair os dígitos de um número;
- Como converter números de uma base para outra.

Neste capítulo, aprenderemos como extrair e processar os dígitos de um número inteiro e como realizar a conversão de um número entre diferentes bases. Mas antes, faremos uma introdução aos principais sistemas de numeração utilizados para representar números inteiros.

13.1 Sistemas de numeração

Números inteiros podem ser representados utilizando diferentes sistemas de numeração. Por exemplo, no dia-a-dia, representamos os números utilizando o sistema decimal, isto é, utilizamos 10 algarismos para representar os números, dados por $\{0,1,2,3,4,5,6,7,8,9,0\}$. Um número é formado pela concatenação destes algarismos, onde a posição dos algarismos determina um valor pelo qual aquele algarismo deve ser multiplicado. Para o sistema decimal, também chamado de base 10, este valor é de 10^{pos}, onde *pos* é a posição do algarismo, contada a partir da direita, que possui *pos* igual a 0 (zero). Por exemplo, o número 1256 tem o seu valor dado por:

$$1 * 10^3 + 2 * 10^2 + 5 * 10^1 + 6 * 10^0 = 1000 + 200 + 50 + 6 = 1256$$

Já os computadores representam os dados internamente utilizando o sistema binário, cujos números são representados pelos algarismo {0, 1}. O motivo para a utilização de uma base binária se deve ao fato dos computadores trabalharem utilizando apenas dois níveis de voltagem, por exemplo, $+5V$ e $0V$. Voltando ao exemplo anterior, o número 1256 seria representado na base binária por 10011101000_2, pois:

$$10011101000_2 = 2^10 + 2^7 + 2^6 + 2^5 + 2^3 = 1024 + 128 + 64 + 32 + 8 = 1256$$

Apesar da representação binária ser aquela utilizada internamente pelo computador, esta é de difícil leitura para os seres humanos, devido às longas seqüências de algarismos 0 e 1. Para aliviar este problema, duas outras bases são bastantes utilizadas na computação, a base octal e a hexadecimal.

A base octal corresponde à concatenação de 3 bits, que combinados dão origem a 8 diferentes valores, representados pelos algarismos de 0 a 7. Já a base hexadecimal corresponde à concatenação de 4 bits, gerando 16 combinação diferentes. Como possuímos apenas 10 algarismo numéricos, utilizamos as letras do alfabeto $\{A,B,C,D,E,F\}$. O número 1256 nestas bases seria então dado por:

$$2350_8 = 2*8^3+3*8^2+5*8^1 = 2*512+3*64+5*8 = 1256$$

$$4E8_{16} = 4*16^2+14*16^1+8*16^0 = 4*256+14*16+8 = 1256$$

Reparem que a representação hexadecimal é mais compacta que a decimal, o que é esperado, dado que utilizamos mais algarismos para representar os números. Veremos agora como realizar a conversão de números inteiros entre diferentes sistemas de numeração.

13.2 Conversão entre sistemas de numeração

Analise agora, com cuidado, a implementação da classe Conversão que converte números binários para decimais e vice-versa. Este exemplo também demonstra como realizar a extração dos dígitos de um número inteiro. Não é muito fácil entender o funcionamento de seus métodos; sugerimos fortemente, então, que você tente rastrear[1] a execução dos dois métodos.

```
class Conversão
{
  int binárioParaDecimal(int n)
  {
    int dec = 0;
    int pot2  = 1;

    while (n != 0){
      /* processa um dígito binário */
      dec = dec + n % 10 * pot2;
      n = n / 10;
      pot2 = pot2 * 2;
    }
    return dec;
  }

  int decimalParaBinário(int n)
  {
    int dig;
    int bin = 0;
    int pot = 1;
```

[1] Rastrear a execução de um programa significa construir uma tabela contendo colunas correspondentes às variáveis do programa e simular a sua execução indicando os valores que as variáveis recebem ao longo do tempo. Se o programa imprimir texto (usando `System.out.println`) esse texto tem que ser indicado como *saída* e se os métodos devolvem algum valor (com o comando `return`) este valor tem que ser destacado.

```
        while (n > 0) {
          /* extrai próximo dígito binário menos significativo (mais à direita) */
          dig = n % 2;
          /* remove esse dígito do n */
          n = n / 2;
          /* adiciona o dígito como o mais significativo até o momento */
          bin = bin + dig * pot;
          pot = pot * 10;
        }
        return bin;
      }
    }
```

Exercícios

1. Crie uma classes `Contador` com um método que recebe um número natural `n` e devolve seu número de dígitos. Em seguida, escreva uma classe `TestaContador` que testa o método para diferentes valores de `n`. Não se esqueça de testar o caso do número ser igual a zero!

2. Dado um número, verificar se o mesmo possui dois dígitos consecutivos iguais. Para resolver este problema podemos usar um técnica denominada indicador de passagem, para isto, inicialmente vamos supor que o número não contém dígitos iguais. Verificaremos cada par de dígitos consecutivos, caso algum deles seja igual, saberemos que ele contém dígitos consecutivos iguais. Em outras palavras, vamos inicializar uma variável booleana com falso e testar a condição para todos os dígitos consecutivos; se forem iguais mudamos a condição. Ao final, a resposta vai corresponder ao estado final desta condição.

3. Dado um número natural `n`, verificar se `n` é palíndromo. Um número palíndromo é um número que lido de trás para frente é o mesmo quando lido normalmente, por exemplo:

   - 78087
   - 1221
   - 11111
   - 3456 não é palindromo!!!

   Duas maneiras de se resolver estes problemas são apresentadas abaixo. Escreva as soluções para cada uma delas.

   - A forma mais fácil é construir o número inverso e compará-lo com o original.
   - A outra solução consiste em supor inicialmente que o número é palíndromo e em seguida verificar se a condição de igualdade é válida para os extremos.

   Se o número for negativo, considere apenas o seu valor absoluto (isso é apenas uma convenção nossa para este exercício). Por exemplo, -2002 deve ser considerado palíndromo.

   Curiosidade: números palíndromos também são conhecidos por *capicuas*.

4. Uma propriedade de números naturais é a seguinte: um número sempre é maior do que o produto dos seus dígitos. Faça uma classe com dois métodos: `int calculaProd(int n)`, que calcula o produto dos dígitos de um número natural `n` e `boolean verificaProp(int n)`, que verifica se a propriedade é válida para um número `n` dado.

5. Crie métodos para converter uma `String` contendo um número em algarismos romanos em um inteiro e vice-versa. Crie testes para estes métodos.

6. Crie uma classe contendo um método que recebe um inteiro e o imprime representado em notação científica. Por exemplo,

```
> Inteiro i = new Inteiro();
> i.carregaValor(1356);
> i.imprimeEmNotacaoCientifica();
1,356e3
> i.carregaValor(-102);
> i.imprimeEmNotacaoCientifica();
-1,02e2
> i.carregaValor(7);
> i.imprimeEmNotacaoCientifica();
7,0e0
> i.carregaValor(900200);
> i.imprimeEmNotacaoCientifica();
9,002e5
```

7. Implemente a operação de divisão de dois números inteiros utilizando apenas laços e os operadores `+` e `-`.

Capítulo 14

Arrays (vetores)

Quais novidades veremos neste capítulo?

- *arrays* (vetores);
- programas independentes em Java (método `main`).

Muitas vezes, precisamos que um objeto guarde um grande número de informações. Por exemplo, se precisamos calcular a temperatura média em um dado mês poderíamos ter uma classe similar à seguinte:

```
class TemperaturasDoMês
{
    double t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12
           t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23,
           t24, t25, t26, t27, t28, t29, t30, t31;
     // etc.
}
```

onde cada variável guarda a temperatura média de um dia do mês. Isso é claramente indesejável. Imagine ainda se quiséssemos uma classe para guardar as temperaturas médias de todos os dias do ano. Precisaríamos de 365 variáveis? Felizmente não!

A linguagem Java possui o conceito de *array* que é uma estrutura de dados que permite guardar uma seqüência de valores (números, caracteres ou objetos quaisquer) de uma forma única e organizada. Utilizando um *array*, a classe anterior ficaria assim:

```
class TemperaturasDoMês
{
    double[] temperaturas = new double[31];
    //etc.
}
```

Note que, no exemplo acima, a linha que define o *array* `temperaturas` faz duas operações simultaneamente. Esta linha poderia ser separada em dois passos:

- `double[] temperaturas;` define um novo *array* chamado temperaturas que irá conter valores do tipo double. Por enquanto, o *array* está vazio.

- `temperaturas = new double[31];` especifica que o *array* guardará exatamente 31 valores do tipo double. Neste instante, o ambiente Java reserva a memória necessária para guardar estes 31 valores.

> **Nota Lingüística:** a tradução padrão de *array* para o português é *vetor*. No entanto, a linguagem Java contém um tipo de objeto chamado `Vector` que é semelhante a *arrays*, mas não é igual. Para evitar confusões entre *arrays* e Vectors, preferimos não traduzir a palavra *array* para vetor neste livro.

Vejamos agora um exemplo simples de utilização de *arrays*.

```
class BrincadeirasComArrays
{
    String[] diasDaSemana = new String[7];
    int[] quadrados = new int[10];

    void defineDiasDaSemana()
    {
        diasDaSemana[0] = "domingo";
        diasDaSemana[1] = "segunda-feira";
        diasDaSemana[2] = "terça-feira";
        diasDaSemana[3] = "quarta-feira";
        diasDaSemana[4] = "quinta-feira";
        diasDaSemana[5] = "sexta-feira";
        diasDaSemana[6] = "sábado";
    }

    void calculaQuadrados()
    {
        int i = 0;
        while (i < 10)
        {
            quadrados[i] = i*i;
            i++;
        }
    }

    void listaDiasDaSemana()
    {
        int i = 0;
        while (i < 7)
        {
            System.out.println(diasDaSemana[i]);
            i++;
        }
    }

    void listaQuadrados()
    {
        int i = 0;
        while (i < 10)
        {
```

```
            System.out.println(i + " ao quadrado é " + quadrados[i]);
            i++;
        }
    }
}
```

O atributo `length`

Arrays são na verdade um tipo especial de objeto em Java. Qualquer *array* já vem com um atributo pré-definido, chamado `length`, que contém o comprimento do *array*. Desta forma, o método `calculaQuadrados` acima poderia ser reescrito para

```
    void calculaQuadrados()
    {
        int i = 0;
        while (i < quadrados.length)
        {
            quadrados[i] = i*i;
            i++;
        }
    }
```

O valor do atributo length é definido automaticamente pelo ambiente Java, o programador não pode alterá-lo. Assim, `quadrados.length = 2` é uma operação ilegal.

Inicialização de *arrays*

Existe também a opção de inicializar um *array* no momento em que ela é declarada. Assim, podemos inicializar *arrays* de inteiros e de Strings conforme o exemplo a seguir:

```
int[] primos = {2, 3, 5, 7, 11, 13, 17, 19, 23};

String[] planetas = {"Mercúrio", "Vênus", "Terra", "Marte", "Júpiter", "Saturno",
                     "Urano", "Netuno", "Plutão"};
```

14.1 Criação de programas Java

Até este capítulo, todos os exemplos de códigos que vimos, a rigor, não eram "programas", eles eram apenas classes Java que podiam ser usadas dentro do interpretador do DrJava. Mas, e se quiséssemos criar um programa para ser utilizado por alguém que não possui o DrJava em sua máquina? Neste caso, precisamos criar um programa a ser executado ou na linha de comando do sistema operacional ou dando um "clique duplo"com o mouse em cima do ícone do programa. Para fazer isso, basta que a classe principal do programa possua um método `main` como no exemplo a seguir.

```
class BrincadeirasComArrays
{
    // aqui vão os demais métodos e atributos da classe

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

```
    {
        BrincadeirasComArrays b = new BrincadeirasComArrays();
        b.defineDiasDaSemana();
        b.calculaQuadrados();
        b.listaDiasDaSemana();
        b.listaQuadrados();
        b.imprimeArray(arg);
    }

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

Para executar o seu programa após compilar a classe, basta abrir um *shell* (um interpretador de comandos do sistema operacional; no unix pode ser, por exemplo, o bash; no windows pode ser, por exemplo, o command) e digitar

```
java BrincadeirasComArrays um dois três
```

onde, `java` é o nome do interpretador Java, `BrincadeirasComArrays` é o nome da classe que será carregada e cujo método `main` será executado e `um dois três` são apenas um exemplo de 3 argumentos que estamos passando, poderia ser qualquer outra coisa.

Neste exemplo, o programa `BrincadeirasComArrays` geraria a seguinte saída:

```
domingo
segunda-feira
terça-feira
quarta-feira
quinta-feira
sexta-feira
sábado
0 ao quadrado é 0
1 ao quadrado é 1
2 ao quadrado é 4
3 ao quadrado é 9
4 ao quadrado é 16
5 ao quadrado é 25
6 ao quadrado é 36
7 ao quadrado é 49
8 ao quadrado é 64
9 ao quadrado é 81
```

```
um
dois
três
```

portanto, para que uma classe possa ser executada a partir da linha de comando do sistema, é necessário que ele possua um método com a seguinte assinatura:

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

o nome do parâmetro não precisa ser exatamente `arg`, qualquer nome funciona; mas o seu tipo tem que ser obrigatoriamente um *array* de `String`s. A partir do DrJava é possível executar o método `main` pressionando a tecla F2.

Exercícios

> **Testes:** até este capítulo, em diversos exercícios pedimos explicitamente para você criar testes para classes e métodos escritos. Fizemos isto para enfatizar a necessidade de escrever testes, um processo que deve ser realizado de modo automático. A partir deste capítulo não iremos mais solicitar em cada exercício que você escreva testes. Você deverá, a partir de agora, fazer isso naturalmente.

1. Escreva uma classe `Simples` contendo um método que recebe um *array* de inteiros como parâmetro e que inicializa todos os elementos do *array* com um valor, também dado como parâmetro. O método deve devolver o tamanho do *array*. A assinatura do método deve ser a seguinte:

   ```
   int inicializaArray(int[] a, int v);
   ```

   Escreva agora um método que recebe um *array* de inteiros como parâmetro e imprime o seu conteúdo:

   ```
   void imprimeArray(int[] a);
   ```

   Crie agora um método que, dado um inteiro, verifica se ele está presente no *array*.

   ```
   boolean estáNoArray(int []a, int v);
   ```

   Finalmente, escreva um programa que cria um *array*, cria um objeto `Simples` e chama os seus três métodos.

2. Crie um método `double[] freqüênciaRelativa(int[] v, int n)` que recebe um vetor contendo números inteiros no intervalo [0, n] e devolve um vetor contendo a freqüência relativa de cada um destes números.

3. Crie um método que, dados dois vetores `a` e `b`, verifica se o vetor de menor tamanho é uma subseqüência do vetor de tamanho maior.
   Ex: O vetor [9, 5] é uma subseqüência de [3, 9, 5, 4, -1].

4. Crie uma classe contendo um método que, dado um vetor `v` de inteiros, imprime o segmento de soma máxima.
   Ex: No vetor [-1, 5, -4, 7, 2, -3] o segmento de soma máxima é [5, -4, 7, 2]

Capítulo 15

`for`, leitura do teclado e conversão de `Strings`

Quais novidades veremos neste capítulo?

- `for` (mais um comando de repetição);
- Leitura de dados do teclado;
- Conversão de `Strings` para números.

15.1 O comando `for`

Você deve ter notado que, quando implementamos laços com o comando `while`, é muito comum que sigamos um padrão bem definido. Considere os dois exemplos a seguir que vimos no capítulo anterior:

```
void listaDiasDaSemana()
{
  int i = 0;
  while (i < 7)
  {
    System.out.println(diasDaSemana[i]);
    i++;
  }
}

void calculaQuadrados()
{
  int i = 0;
  while (i < quadrados.length)
  {
    quadrados[i] = i*i;
    i++;
  }
```

```
}
```

Em ambos os casos, a estrutura do laço é a seguinte:

```
int i = 0; // define onde o laço se inicia

// define o critério para continuarmos dentro do laço
while (i < quadrados.length)
{
  quadrados[i] = i*i; //a operação propriamente dita
  i++;  //comando de atualização para passarmos para a próxima iteração
}
```

Ou seja, um formato genérico para laços com `while` seria o seguinte:

```
inicialização;
while (condição para continuar)
{
  comando;
  atualização;
}
```

Após muito observar este padrão, projetistas de linguagens de programação decidiram criar um novo comando para implementação de laços onde todas estas partes relacionadas com o laço fossem organizadas de uma forma melhor. Assim, surgiu o comando `for` que, nas linguagens C, C++ e Java, adquirem o seguinte formato:

```
for (inicialização; condição para continuar; atualização )
{
  comando;
}
```

onde as chaves são apenas necessárias se comando for composto. Com o `for`, podemos implementar exatamente o mesmo que com o `while` mas, no caso do `for`, todas as operações relacionadas com o laço ficam na mesma linha, o que facilita a visualização e o entendimento de quem olha para o código.

15.2 Leitura do teclado

No capítulo anterior, aprendemos como escrever um programa em Java que pode ser executado a partir da linha de comando, sem a necessidade de utilizar um ambiente como o DrJava para executá-lo. Aprendemos também a receber dados como parâmetro através da linha de comando utilizando o parâmetro `args` do método `main`. No entanto, existem casos em que gostaríamos de ler valores digitados pelo usuário através do teclado durante a execução do programa. Até a versão 1.4, Java oferecia algumas formas de se fazer isso mas nenhuma delas era muito simples. A partir da versão 1.5 foi introduzida a classe `Scanner` que possui vários métodos para a leitura do teclado.

Observe a classe seguinte:

```
import java.util.*;

class UsoScanner
{
```

```
  public static void main(String [] args)
  {
    Scanner sc = new Scanner(System.in);
    int x = sc.nextInt();
    System.out.println(" O inteiro lido foi " + x);
    double d = sc.nextDouble();
    System.out.println(" O double lido foi " + d);
    String s = sc.next();
    System.out.println(" A string lida foi " + s);
  }
}
```

Nela podemos ver que, para a leitura de dados a partir do teclado, primeiro precisamos instanciar um objeto do tipo `Scanner`, para em seguida podermos usar métodos como `nextInt` para a leitura de um número do tipo `int`. Caso o dado lido não seja do tipo esperado, uma exceção[1] é lançada.

Versões anteriores a JDK 1.5

Até a versão 1.4 não havia forma simples de se efetuar a leitura a partir do teclado. Para facilitar a vida de iniciantes em programação Java com versões mais antigas do JDK, podemos utilizar uma classe que esconde as partes mais complicadas da leitura através do teclado. Neste livro utilizaremos a classe `SavitchIn` escrita pelo Prof. Walter Savitch da Universidade da California em San Diego. A classe `SavitchIn` pode ser encontrada no sítio do Prof. Savitch [2].

Após compilar o SavitchIn.java para produzir o SavitchIn.class, basta copiá-lo para o mesmo diretório onde os seus arquivos .class serão guardados para que você possa utilizar os métodos da classe `SavitchIn`. Uma descrição detalhada de todos os métodos da `SavitchIn` está disponível na Web[3]. Mas vamos nos concentrar apenas nos seguintes métodos:

```
public class SavitchIn
{
    public static String readLine();
    public static int readLineInt(); // supõe que há apenas um inteiro
na linha
    public static double readLineDouble(); // apenas um double na linha
    /************************************************************ **
     * In the following method
     * If the input word is "true" or "t", then true is returned.
     * If the input word is "false" or "f", then false is returned.
     * Uppercase and lowercase letters are considered equal. If the
     * user enters anything else (e.g., multiple words or different
     * words), then the user is asked to reenter the input.
     ************************************************************/
    public static boolean readLineBoolean();
}
```

[1] Exceções (`Exceptions`) são representadas por um tipo especial de objeto e utilizadas em Java para o tratamento de erros. Este recurso não é exclusivo de Java, sendo utilizado também em outras linguagens, como o C++. Mais informações sobre o uso de exceções em Java podem ser encontradas no site `http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html`.

[2] http://www.cs.ucsd.edu/users/savitch/java/SavitchIn.txt

[3] http://www.sinc.sunysb.edu/Stu/fmquresh/SavitchIn.html

A palavra `static` na declaração de cada método indica que este método pode ser utilizado mesmo que não existam instâncias da classe. Para utilizá-lo basta fazer algo como no exemplo seguinte:

```
for (i = 1; i <= 10; i++)
{
    int num = SavitchIn.readLineInt();
    System.out.println (i + "o número digitado: " + num);
}
```

Nota: Por uma limitação do DrJava, a classe SavitchIn ainda não funciona na janela Interactions. Segundo os desenvolvedores do DrJava, esta limitação será solucionada no futuro. Portanto, por enquanto, você deve utilizar a SavitchIn em programas a serem executados na linha de comando.

15.3 Conversão de `String` para números

Você deve ter notado que o método `main` recebe sempre `String`s como parâmetro. Mas e se quisermos receber valores numéricos? Neste caso temos que converter o String que recebemos em um número. Isso pode ser feito através da classe `Integer` ou da classe `Double` que contém métodos para efetuar estas conversões:

```
String meuStringInteiro = "10";
int meuInteiro = Integer.parseInt(meuStringInteiro);

String meuStringReal = "3.14159265";
double meuReal = Double.parseDouble(meuStringReal);
```

Exercícios

1. Usando o comando `for`, escreva um método que recebe um *array* de `double`s como parâmetro e imprime o seu conteúdo.

2. Usando o comando `for`, escreva um método `somaArrays` que recebe 3 *arrays* de mesmo comprimento como parâmetro e que calcula a soma dos dois primeiros e a armazena no terceiro. A soma deve ser implementada como soma vetorial, ou seja, soma-se a primeira posição de um vetor com a primeira posição do segundo, armazenando-se o resultado na primeira posição do terceiro e assim por diante.

   Nota: o terceiro *array* que é passado como parâmetro é chamado de **parâmetro de saída**. Quando o método se inicia, ele já contém um *array* que é passado por quem chamou o método `somaArrays`, mas o seu conteúdo inicial não é relevante. Isso funciona em Java apenas quando passamos uma referência de objeto como parâmetro, como vimos no Capítulo 12.

3. Escreva um método que recebe um array de `doubles` como parâmetro e imprime a média dos valores nele contidos.

4. Escreva dois métodos (`min` e `max`) que recebem um array de inteiros como parâmetro e devolvem, respectivamente, um inteiro correspondente ao menor e ao maior elemento do array.

5. Escreva uma classe ProdutoEscalar contendo os seguintes métodos:

(a) método que recebe um array "vazio"como parâmetro e que lê valores `double` do teclado para serem inseridos no array.

(b) método que recebe 2 arrays de `double` como parâmetro e devolve o valor do produto escalar (soma dos produtos das posições de mesmo índice dos dois arrays).

(c) método `main` que, usando os métodos anteriores, lê do teclado o comprimento dos vetores, lê do teclado o conteúdo de dois vetores com o tamanho comprimento, calcula o produto escalar e imprime o resultado.

Capítulo 16

Laços Encaixados e Matrizes

Quais novidades veremos neste capítulo?

- repetições encaixadas;
- matrizes (arrays multidimensionais).

16.1 Laços encaixados

Em algumas situações, pode ser necessário implementar um laço dentro de outro laço. Chamamos esta construção de laços encaixados, laços aninhados ou repetições encaixadas. Eis um exemplo bem simples:

```
int i, j;
for (i = 0; i < 5; i++)
    for (j = 0; j < 3; j++)
        System.out.println("i = " + i + ", j = " + j);
```

A idéia é que, para cada iteração do `for` mais externo, o interpretador Java executa o `for` mais interno com todas as suas iterações. No exemplo acima, o `println` é, portanto, executado 5*3 = 15 vezes.

Agora um exemplo mais complexo: um programa usado para calcular a média de várias turmas na prova `P1` de uma disciplina:

```
class MediasNaP1
{
  public static void main (String[] arg)
  {
    int númeroDeAlunos, númeroDeTurmas, turma, aluno;
    double nota, soma;
    Scanner sc = new Scanner(System.in);

    System.out.print("Você quer calcular as médias de quantas turmas? ");
    númeroDeTurmas = sc.nextInt ();
    for (turma = 1; turma <= númeroDeTurmas; turma++)
    {
```

```
        soma = 0;
        nota = 0;
        System.out.println("Digite as notas na P1 da turma " + turma);
        System.out.println("Finalize a seqüência de notas digitando -1.");
        for (númeroDeAlunos = 0; nota >= 0; númeroDeAlunos++)
        {
          nota = sc.nextDouble();
          if (nota >= 0)
            soma += nota;
          }
        System.out.println("A média da turma é " + soma /
                           (númeroDeAlunos - 1));
      }
    }
}
```

16.2 Matrizes (*arrays* multidimensionais)

Uma matriz de 1 dimensão, também chamada de vetor, é implementada em Java usando um array unidimensional, como vimos no Capítulo 14. Matrizes de 2 ou mais dimensões também podem ser representadas em Java através de arrays multidimensionais. Veja um exemplo:

```
// define uma matriz bidimensional
int[][] matriz;

// aloca espaço para uma matriz de 5 linhas e 7 colunas
matriz = new int[5][7];
matriz[0][0] = 3;
matriz[1][1] = matriz[3][4] + matriz [1][2];
```

É muito comum utilizarmos repetições encaixadas quando precisamos percorrer os elementos de matrizes de duas dimensões ou mais:

```
int i, j;
int [][] tabuada = new int[11][11];
for (i = 0; i <= 10; i++)
    for (j = 0; j <= 10; j++)
        tabuada[i][j] = i * j;
```

este trecho de código cria uma matriz de duas dimensões contendo a tabuada de 0 a 10. Note que a primeira linha e a primeira coluna não precisariam ser inicializadas com uma multiplicação, poderíamos atribuir o valor 0 diretamente a elas.

Nota interessante: para Java, na verdade uma matriz bidimensional é um array de arrays. Ou seja, `tabuada` é um array onde cada posição é do tipo array de inteiros. Dito isto, pense como podemos fazer para descobrir quantas linhas e quantas colunas possui uma matriz bidimensional. A resposta é a seguinte:

```
void imprimeDimensões(int[][] matriz)
  {
    System.out.println("A matriz possui " +
                       matriz.length      + " linhas e " +
```

```
                        matriz[0].length    + " colunas.");
    }
```

16.3 Exemplo: LIFE, o jogo da vida

O LIFE é um jogo simples de simulação de processos biológicos criado pelo matemático John Conway. O "ambiente" onde se passa a simulação é uma grade quadriculada onde são colocadas "células" vivas; cada quadrado da grade pode conter ou não uma célula viva. A partir de um estado inicial (que pode ser gerado aleatoriamente, por exemplo), o estado seguinte da grade é determinado através de 3 regras bem simples:

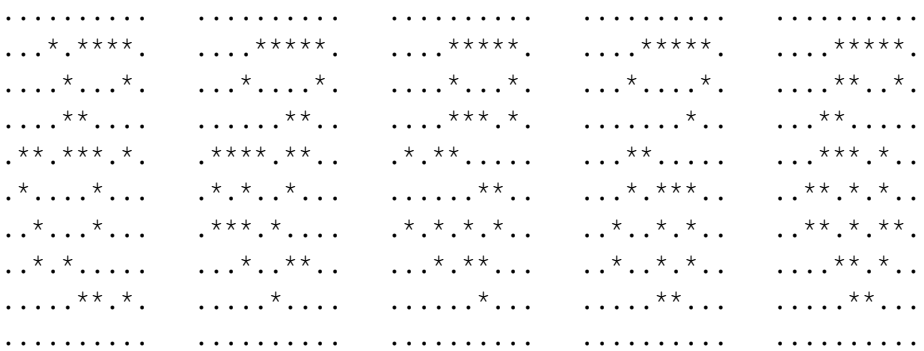
- Uma célula viva com menos de 2 vizinhos morre.
- Uma célula viva com mais de 3 vizinhos morre.
- Uma célula viva aparece quando tem 3 vizinhos vivos exatamente.

O processo de simulação é iterativo, ou seja, as regras são aplicadas ao estado inicial que produz um segundo estado. A este segundo estado são aplicadas as regras novamente e assim sucessivamente, criando novas "gerações" de células ao longo do tempo. Veja um exemplo de implementação do LIFE abaixo. Estude atentamente cada trecho do programa e depois o execute no DrJava.

```
class Life
{
  int MAX = 10;  // Tamanho da matriz
  int [][] matriz = new int[MAX][MAX];

  void inicializa()
  {
    int i, j;

    for (i = 1; i < MAX - 1; i++)
       for (j = 1; j < MAX - 1; j++)
           matriz[i][j] = (int) (Math.random() * 1.5);
           // o Math.random gera um número em [0,1], multiplicando
           // por 2/3 conseguiremos 2/3 das casas com zeros e 1/3 com 1s
           // o (int) transforma o double obtido em inteiro

    // Os mais observadores podem perceber que as bordas da matriz não
    // foram inicializadas. Fizemos isso para deixá-las zeradas (o Java
    // faz isto automaticamente quando da alocação da matriz (new int).
  }

  void imprimeTabuleiro()
  {
    int i, j;

    for (i = 0; i < MAX; i++)
    {
      for (j = 0; j < MAX; j++)
        if (matriz[i][j] == 1)
```

```
            System.out.print("*");
          else
            System.out.print(".");
        System.out.println();
      }
      System.out.println();
    }

    int vizinhos(int i, int j)
    {
      return matriz[i-1][j-1] + matriz[i-1][j] + matriz[i-1][j+1] +
             matriz[i][j-1] + matriz[i][j+1] +
             matriz[i+1][j-1] + matriz[i+1][j] + matriz[i+1][j+1];
    }

    int [][] iteração()
    {
      int [][] aux = new int[MAX][MAX];
      int i, j;

      for (i = 1; i < MAX - 1; i++)
         for (j = 1; j < MAX - 1; j++)
            {
              if (matriz[i][j] == 1)  // se está viva
              {
                 if ((vizinhos(i, j) < 2) || (vizinhos(i, j) > 3))
                   aux[i][j] = 0;      // morre
                 else
                   aux[i][j] = 1;      // continua viva
              }
              else                     // se não está viva
               {
                if (vizinhos(i, j) == 3)
                  aux[i][j] = 1;       // aparece vida
                else
                  aux[i][j] = 0;       // continua como estava
              }
            }
       return aux; // devolve a matriz com a nova iteração
    }

    void simulaVida(int quant)
    {
      int i;

      // faremos a simulação de quantos ciclos
      for(i = 0; i < quant; i++)
      {
        imprimeTabuleiro();
        matriz = iteração(); // a matriz da iteração anterior é recolhida
                             // pelo coletor de lixo.
      }
    }
}
```

Observe abaixo, cinco iterações do Life, da esquerda para a direita.



Exercícios

1. Escreva um método para preencher uma matriz com números inteiros lidos do teclado. O método recebe a matriz como parâmetro.

2. Escreva uma classe com os seguintes métodos:

   - Um método que recebe 3 matrizes do tipo `double` como parâmetro, calcula a soma das duas primeiras e guarda o resultado na terceira.
   - Um método que recebe 3 matrizes do tipo `double` como parâmetro, calcula o produto das duas primeiras e guarda o resultado na terceira.
   - Para os dois métodos anteriores, escreva uma segunda versão que recebe apenas 2 matrizes como parâmetro e devolve o resultado através do comando `return`.

3. Adicionar à classe do exercício anterior um método que recebe duas matrizes quadradas do tipo `double` e devolve um `boolean` indicando se uma é inversa da outra. Lembrando da imprecisão das contas com ponto flutuante considere que um número é igual a zero se ele pertencer a $[-10^{-8}, 10^{-8}]$. De forma análoga considere que um número é igual a 1 se ele pertencer a $[1-10^{-8}, 1+10^{-8}]$.

4. Dizemos que uma matriz A_{nxn} é um *quadrado latino* de ordem `n` se em cada linha e em cada coluna aparecem todos os inteiros 1,2,3,...,n (ou seja, cada linha e coluna é permutação dos inteiros 1,2,...,n). Escreva uma classe com os seguintes métodos:

   (a) `imprimeMatriz()`, que imprime uma matriz;

   (b) `estáNoArray()` que recebe como parâmetros um inteiro `j` e um array `a` de inteiros e verifica se `j` ocorre no array;

   (c) `temTodos()` que, usando o método acima, recebe como parâmetros um array `a` com `n` inteiros e verifica se em `a` ocorrem todos os inteiros de 1 a `n` e

   (d) `éLatino()` que, utilizando o método acima, verifica se uma dada matriz inteira A_{nxn} é um quadrado latino de ordem `n`.

Veja um exemplo de uso de um objeto da classe nas seguintes iterações. Aproveitamos o exemplo abaixo para mostrar como podemos definir matrizes constantes (usando chaves).

```
Welcome to DrJava.
> QuadradoLatino a = new QuadradoLatino();
> int [][] matriz1 = {{1}};
> a.imprimeMatriz(matriz1)
1
> a.éLatino(matriz1)
true
> int [][] matriz2 = {{1,2,3}, {2,3,1}, {3,1,2}};
> a.imprimeMatriz(matriz2)
1 2 3
2 3 1
3 1 2
> a.éLatino(matriz2)
true
> int [][] matriz3 = {{1,2}, {1,2}};
> a.imprimeMatriz(matriz3)
1 2
1 2
> a.éLatino(matriz3)
false
```

5. Dizemos que uma matriz A_{nxn} é um *quadrado mágico* de ordem `n` se o valor da soma dos elementos em cada linha e em cada coluna é o mesmo. Escreva uma classe que inclua métodos para verificar se uma dada matriz (recebida como parâmetro em um de seus métodos) é um quadrado mágico ou não.

6. Faça uma classe com dois métodos estáticos:

   - `int pertence(int el, int v[], int tam)` que verifica se um inteiro `el` ocorre em um *array* `v[]` com `tam` elementos. Se ele ocorre, o método devolve a posição da primeira ocorrência, caso contrário devolve -1.
   - `void freqüência(int v[])` que imprime a freqüência absoluta dos elementos em v.

     Dica: para calcular a freqüência absoluta são necessários dois vetores, um com os elementos distintos e outro com o número de ocorrências. Percorra o *array* `v` verificando, para cada posição, se o número armazenado já apareceu.

7. Um jogo de palavras cruzadas pode ser representado por uma matriz A_{mxn} onde cada posição da matriz corresponde a um quadrado do jogo, sendo que 0 indica um quadrado branco e -1 indica um quadrado preto. Indicar na matriz as posições que são início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras. Para isso, numere consecutivamente tais posições.
   Exemplo: Dada a matriz:

0	-1	0	-1	-1	0	-1	0
0	0	0	0	-1	0	0	0
0	0	-1	-1	0	0	-1	0
-1	0	0	0	0	-1	0	0
0	0	-1	0	0	0	-1	-1

A saída deverá ser:

1	-1	2	-1	-1	3	-1	4
5	6	0	0	-1	7	0	0
8	0	-1	-1	9	0	-1	0
-1	10	0	11	0	-1	12	0
13	0	-1	14	0	0	-1	-1

8. (Difícil) Escreva uma classe com um método que leia um inteiro `n` e as posições de `n` rainhas em um tabuleiro de xadrez e determina se duas rainhas se atacam.

Capítulo 17

Busca e Ordenação

Quais novidades veremos neste capítulo?

- Busca em array;
- Ordenação de arrays.

17.1 Busca

Como já vimos, arrays podem guardar muita informação embaixo de uma única variável. Uma das operações mais comuns que podemos fazer em um array é buscar um elemento nele. Isso pode ser útil para saber se um elemento já está lá para recuperar outras informações dos objetos armazenados. Pense, por exemplo, na busca de um CD pelo título. Sabendo onde o CD está, podemos recuperar outras informações, como as faixas que o compõem, sua duração, etc.

Mas como podemos buscar algo? Vamos primeiro aprender a forma mais simples de busca: a busca seqüencial. Neste caso, "varre-se" o array do início ao fim até encontrarmos o elemento desejado. Isso pode ser feito usando um simples laço `while`. Considere que `biblioteca` é um array de objetos da classe `Livro`. O array está cheio de livros válidos, ou seja, tem elementos de 0 a `biblioteca.length` - 1. Considere ainda que esses objetos disponibilizam um método `pegaNome` que devolve o nome do livro. Veja abaixo um exemplo de código para buscar um livro de nome "Amor em tempos de cólera".

```
int i = 0;
while((i < biblioteca.length) &&
      (biblioteca[i].pegaNome().equals("Amor em tempos de cólera") == false))
  i = i + 1;
```

Ao final, temos duas possibilidades:

- `i < bibliotecas.length`: Neste caso o livro está na posição `i` do array. De posse desse índice podemos manipular o livro encontrado como quisermos.
- `i == bibliotecas.length`: Já, aqui, o livro não foi encontrado pois passamos do final do array.

Por fim, devemos destacar que há uma certa sutileza no código acima. Note que no segundo caso não poderíamos acessar o array na posição `i`. Isso não ocorre devido à ordem das condições do `while`. Como a primeira condição é falsa, o computador não precisa testar a segunda para saber que a expressão é falsa, já que a condição é um "e-lógico".

17.2 Pondo ordem na casa

Uma das tarefas mais comuns em programas é ordenar coisas. Muitas vezes é mais fácil trabalhar com os elementos de um array ordenados. Um bom exemplo é a busca. Caso o array esteja ordenado pela "chave" de busca, como veremos no próximo capítulo, é possível fazer algo bem mais eficiente do que a busca seqüencial descrita acima.

Mas como podemos ordenar os elementos de um array?

Para fazer isso vamos nos lembrar de algo que universitários fazem muito: jogar baralho. Quando temos várias cartas na mão, muitas vezes acabamos por ordená-las. Pensem e discutam um pouco como vocês fazem isso. Existe alguma lógica por trás da maneira que você move as cartas para lá e para cá? Será que você poderia descrever um algoritmo para ordenar as cartas?

Agora vamos fazer a brincadeira ficar um pouco mais complicada. Seja qual for o algoritmo que você descreveu acima, tente pensar como adaptá-lo para incorporar duas restrições. Primeiro, considere que você não pode olhar todas as cartas de uma vez, isto é, no máximo duas cartas podem ficar visíveis ao mesmo tempo. Vale, porém, anotar a posição onde uma carta do seu interesse está. Segundo, considere que você não pode simplesmente mover as cartas de um lugar para outro, mas somente pode trocar duas cartas de posição.

É bem provável que a solução imaginada poderia ser descrita em Java por um dos algoritmos abaixo. Neles deseja-se ordenar os elementos no array de inteiros `números`. Cada algoritmo é escrito como um método.

1. **Seleção direta**. Neste algoritmo, a cada passo buscamos o menor elemento no array e o levamos para o início. No passo seguinte buscamos o segundo menor elemento e assim por diante.

```
void seleçãoDireta(int []números)
{
  int i, j, índiceDoMínimo, temp;
  int fim = números.length;

  for (i = 0; i < fim - 1; i++)
  {
    // Inicialmente o menor elemento já visto é o primeiro elemento.
    índiceDoMínimo = i;
    for (j = i + 1; j < fim; j++)
    {
      if (números[j] < números[indiceDoMínimo])
        índiceDoMínimo = j;
    }
    // Coloca o menor elemento no início do subvetor atual. Para isso, troca
    // de lugar os elementos nos índices i e índiceDoMínimo.
    temp = números[i];
    números[i] = números[índiceDoMínimo];
    números[índiceDoMínimo] = temp;
  }
}
```

2. **Inserção direta**. A cada passo aumenta a parte ordenada do array de uma posição, inserindo um novo elemento na posição correta.

```
void inserçãoDireta(int []números)
{
  int i, j, númeroAInserir;
  int fim = números.length;

  // Cada passo considera que o array à esquerda de i está ordenado.
  for (i = 1; i < fim; i++)
  {
    // Tenta inserir mais um número na porção inicial do array que
    // já está ordenada empurrando para direita todos os elementos
    // maiores do que númeroAInserir.
    númeroAInserir = números[i];
    j = i;
    while((j > 0) && (números[j-1] > númeroAInserir))
    {
      números[j] = números[j-1];
      j--;
    }
    números[j] = númeroAInserir;
  }
}
```

3. **Método da bolha**. Esse método testa se a seqüência está ordenada e troca o par de elementos que gerou o problema. Pense no array como um tubo de ensaio vertical onde os elementos mais leves sobem à superfície como bolhas.

```
void bolha(int []números)
{
  int i, j, temp;
  int fim = números.length;

  for(i = fim - 1; i > 0; i--)
    // Varre o array desde o início procurando erros de ordenação.
    // Como a cada passagem o maior elemento sobe até sua
    // posição correta, não há necessidade de ir até o final.
    for (j = 1; j <= i; j++)
      // Se a ordem está errada para o par j-1 e j
      if (números[j-1] > números[j])
      {
        // Troca os dois de lugar
        temp = números[j-1];
        números[j-1] = números[j];
        números[j] = temp;
      }
}
```

Obs.: Sempre que pedimos para o `System.out.print` ou `System.out.println` imprimirem um array ou outro objeto qualquer de uma classe definida por nós, ele imprime apenas um número estranho (que representa o lugar onde o objeto está armazenado na memória). Deste modo é interessante escrever um método

`imprimeArray` para que o computador mostre todos os elementos válidos de seu array. Esse método é fácil, basta percorrer o array e pedir a cada passo que o computador imprima o elemento atual (ou a parte dele que nos interessa ver).

Exercício

1. Os algoritmos de ordenação neste capítulo foram desenvolvidos utilizando números. Se quisermos ordenar uma seqüência de nomes, por exemplo, {Renault, McLaren, Ferrari, Toyota, Williams, Honda, Red Bull, BMW Sauber}, é preciso modificar estes métodos. Escreva uma versão do método da bolha que ordena um array de `String`.

2. Vamos descobrir como os algoritmos acima descritos se comportam à medida que aumentamos o tamanho do array a ser ordenado? Para cada um dos algoritmos acima execute o algoritmo utilizando como entrada arrays de diferentes tamanhos. O tempo gasto para realizar a ordenação cresce linearmente com o tamanho do array?

3. Algoritmos de ordenação são freqüentemente utilizados junto com outros algoritmos para torná-los mais eficientes.

   - Escreva uma classe que contenha um método `atribuiArray` que recebe um array de inteiros e o armazena como um atributo da classe. Escreva então um método `int nRepetições(int x)` que devolve o número de vezes que `x` aparece neste array
   - Escreva uma outra classe similar à anterior cujo método `atribuiArray` ordena os elementos do array antes de armazená-lo como atributo da classe.

   Forneça a cada uma das classes vetores de diferentes tamanhos contendo números aleatórios. Para cada caso, chame o método `int nRepetições(int x)` utilizando diferentes valores de `x`.

   Podemos argumentar que a primeira classe é mais eficiente no caso em que são realizadas poucas consultas a `nRepetições` e a segunda classe é mais eficiente quando são realizadas muitas consultas. Justifique esta argumentação.

Capítulo 18

Busca Binária e Fusão

Quais novidades veremos neste capítulo?

- Busca binária;
- Complexidade Computacional;
- Fusão de duas seqüências ordenadas.

18.1 Busca binária

Neste capítulo, veremos alguns dos usos da ordenação. Um exemplo especialmente interessante é a busca binária. Suponha que estamos escrevendo um programa que vai guardar um grande número de informações. Estes dados mudam pouco no tempo mas são sempre consultados. Um exemplo típico é a lista telefônica, cujo conteúdo muda muito pouco, mas que é consultada várias vezes por dia no sítio da operadora. Neste caso, vale a pena "pagar o preço" de guardar os dados ordenados pela chave de busca (no caso da lista, o nome do assinante). Isso torna a busca, que é a operação mais freqüente, muito mais rápida. Vejamos como isso pode ser feito.

A idéia da busca binária é bem simples. Grosso modo, como a seqüência está ordenada, sabemos que o ítem que está localizado bem no meio da seqüência, a divide em duas partes: os ítens menores que o ítem do meio ficarão à esquerda dele enquanto que os ítens maiores que ele ficarão à sua direita. Se, em cada iteração, simplesmente compararmos o valor buscado ao valor contido no ítem do meio da seqüência, podemos jogar metade do trabalho fora. Há três possibilidades:

1. O objeto buscado tem a mesma chave do objeto do meio: então já encontramos o que queríamos e nosso trabalho acabou;

2. O objeto buscado vem antes do objeto do meio: podemos então concentrar a busca na porção inicial da seqüência e esquecer a segunda metade;

3. O objeto buscado fica depois do objeto intermediário: a busca deve continuar apenas na segunda metade.

Ou seja, podemos, usando o elemento intermediário, eliminar metade do trabalho a cada passo do nosso algoritmo. Isso faz com que a busca fique muito mais rápida! Vejamos como fica a implementação da busca binária para um array de inteiros. O método abaixo retorna a posição do valor buscado caso este esteja presente no array e -1 caso contrário. Este código pode ser facilmente adaptado para `doubles`, cadeias de caracteres ou qualquer outro tipo de objeto.

```
int buscaBinária(int valor, int[] vetor)
{
  int esq = 0,
      dir = vetor.length - 1,
      meio;

  while (esq <= dir)
  {
    meio = (esq + dir) / 2;
    if (valor > vetor[meio])
      esq = meio + 1;
    else if (valor < vetor[meio])
      dir = meio - 1;
    else
      return meio;
  }
  return -1;
}
```

18.2 Complexidade Computacional

Vamos agora pensar um pouco sobre quanto tempo o algoritmo de busca binária necessita para ser executado, ou em outras palavras, queremos descobrir qual o seu custo computacional ou qual a sua complexidade computacional.

Note que o algoritmo seqüencial de busca que vimos no capítulo anterior precisa percorrer todos os elementos de um vetor, de um em um, até localizar o elemento que estamos procurando. Em particular, se o elemento buscado não se encontra no vetor, seria necessário compará-lo a todos os elementos do vetor para descobrir que ele não se encontra lá. Podemos então dizer que o algoritmo de busca seqüencial utiliza da ordem de n comparações para localizar o ítem procurado, onde n é o número de elementos do vetor.

Suponha que queiramos realizar a busca numa lista telefônica de uma megalópole como São Paulo que possua, por exemplo, 18 milhões de telefones e suponha que nosso software demore, em média, 10μs (10 milionésimos de segundo) para comparar dois nomes. Neste caso, seriam necessários 180s ou 3 minutos para descobrir que um determinado nome não consta na lista. Nos dias de hoje, uma espera dessas é, obviamente, considerada inaceitável.

Se levarmos em conta que o vetor já se encontra previamente ordenado, é possível melhorar um pouco o algoritmo da busca seqüencial fazendo com que a busca termine assim que encontrarmos um elemento maior do que o elemento procurado. Mas, note que se estamos procurando por um valor próximo aos valores dos últimos elementos do vetor, o tempo total da busca continua tão alto quanto antes. Dizemos que a complexidade de pior caso do algoritmo ainda é da ordem de n, onde n é o número de elementos do vetor.

Por outro lado, a busca binária utiliza a informação de que o vetor se encontra pré-ordenado de uma maneira muito perspicaz. Qual a complexidade de pior caso da busca binária? Em outras palavras, qual o maior número

possível de iterações do `while` do método `buscaBinária`? A resposta é simples: note que a cada iteração do `while`, a parte que resta do vetor é dividida ao meio. Então a pergunta que devemos fazer é: quantas vezes eu consigo dividir ao meio um vetor de n elementos até que o vetor resultante contenha apenas um elemento? Pense um pouco... Aqueles com os conhecimento matemáticos em dia vão se lembrar... A resposta é $\log_2 n$, ou seja, o logaritmo de n na base 2. Portanto, a complexidade de pior caso do algoritmo de busca binária é da ordem de $\log_2 n$.

Voltemos ao exemplo da lista telefônica de nossa megalópole. Se nosso software demora, em média, 10μs para comparar dois nomes, então utilizando o algoritmo da busca binária, demoraríamos $\log_2 18.000.000 * \frac{10}{1.000.000} = 24,1 * \frac{10}{1.000.000} = 0.000241s$, ou 0,24 milissegundos, o que é bem melhor do que 3 minutos, não?

Portanto, quando nos deparamos com grandes volumes de dados ou problemas computacionalmente difíceis, a escolha do algoritmo que utilizamos pode ter um grande impacto na eficiência do programa que estamos criando. A área de Análise de Complexidade de Algoritmos estuda exatamente estas questões.

18.3 Fusão

Um outro algoritmo interessante ligado a seqüências ordenadas é a fusão de duas delas. Ou seja, a união (preservando repetições) de duas seqüências ordenadas em uma única seqüência maior ainda em ordem. Esta operação é conhecida como fusão. Vejamos como podemos implementá-la:

```
// Combinação de dois vetores ordenados em um novo vetor ordenado.
// Note que este método devolve um vetor.
int[] fusão(int []a, int b[])
{
  int posa = 0,
      posb = 0,
      posc = 0;
  int[] c = new int[a.length + b.length];

  // Enquanto nenhuma das duas seqüências está vazia...
  while (posa < a.length && posb < b.length)
  {
    // Pega o menor elemento atual entre a e b.
    if (b[posb] <= a[posa])
    {
      c[posc] = b[posb];
      posb++;
    }
    else
    {
      c[posc] = a[posa];
      posa++;
    }
    posc++;
  }
  // Completa com a seqüência que ainda não acabou.
  while (posa < a.length)
  {
    c[posc] = a[posa];
    posc++;
    posa++;
```

```
    }
    while (posb < b.length)
    {
      c[posc] = b[posb];
      posb++;
      posc++;
    }

    return c;
  }
```

Exercícios

1. Implemente o algoritmo de busca ternária, que funciona de forma similar ao de busca binária mas que divide o vetor em três partes iguais a cada iteração.

2. Construa um programa que o auxilie a determinar qual algoritmo é mais eficiente: a busca binária ou a busca ternária.

3. Escreva novamente o algoritmo de fusão de seqüências de forma a não preservar repetições de elementos. Ou seja, se um elemento aparece em ambas seqüências de entrada, a seqüência resultante conterá apenas uma cópia dele.

4. Analise a complexidade do algoritmo de fusão para o caso em que ambos os vetores de entrada têm tamanho n. Em outras palavras, qual a ordem do número de comparações e atribuições utilizadas pelo algoritmo no pior caso.

5. Analise a complexidade do algoritmo de ordenação por seleção direta que vimos no capítulo passado. Quantas comparações e quantas trocas de elementos o algoritmo realiza? Dica: a fórmula da PA (progressão aritmética) pode ajudar.

Capítulo 19

Construtores e Especificadores de Acesso

Quais novidades veremos neste capítulo?

- Construtores;
- Especificadores de acesso;
- `final` e `static`.

19.1 Construtores

Desde o início deste livro, trabalhamos com diferentes tipos de objetos. Mas, antes de começar a usar um objeto de uma classe `ClasseX`, devemos criá-lo. Isto foi feito até agora através de chamadas do tipo: `ClasseX x = new ClasseX();` na qual a variável x passa a se referir a um novo objeto da classe `ClasseX`.

Na linguagem Java, quando não especificamos como um objeto deve ser criado, a própria linguagem nos fornece um construtor padrão. Vejamos com mais detalhes um exemplo abaixo:

```
class Ex1
{
    int a;
    double d;
    String s;
    boolean b;

    void imprime()
    {
      System.out.println("o inteiro vale " + a);
      System.out.println("o real vale " + d);
      System.out.println("a String vale " + s);
      System.out.println("o boolean vale " + b);
    }

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

```
        Ex1 e = new Ex1();
        e.imprime();
    }
}
```

No exemplo acima podemos ver que os números foram inicializados automaticamente com zero, a `String` com `null` e o `boolean` com `false`. A inicialização com a referência nula, `null`, é o padrão para as referências a objetos em geral.

Mas, e se por alguma razão queremos que as variáveis da classe tenham algum valor pré-definido ou, melhor ainda, que seja definido durante a "construção"do objeto? Neste caso podemos definir explicitamente um construtor. Veja o exemplo abaixo:

```
class Ex2
{
    int a;
    double d;
    String s;
    boolean b;

    Ex2(int i1, double d1, String s1, boolean b1)
    {
      a = i1;
      d = d1;
      s = s1;
      b = b1;
    }

    void imprime()
    {
      System.out.println("o inteiro vale " + a);
      System.out.println("o real vale " + d);
      System.out.println("a String vale " + s);
      System.out.println("o boolean vale " + b);
    }
    public static void main(String []args)
    {
      // Ex2 e = new Ex2();   ERRO não podemos mais usar o construtor padrão
      Ex2 obj1 = new Ex2(2, 3.14, "Oi", true);
      Ex2 obj2 = new Ex2(1, 1.0, "Tudo 1", true);

      obj1.imprime();
      System.out.println();
      obj2.imprime();
    }
}
```

A primeira observação importante é que não podemos mais usar o construtor padrão. Se a própria classe nos fornece um construtor, é de se esperar que devamos respeitar algumas regras na hora de construir o objeto, logo o construtor padrão não está mais disponível.

Podemos também pensar em um exemplo um pouco mais sofisticado, criando uma classe que representa contas em um banco. Como atributos óbvios podemos pensar em ter o nome do titular e o saldo de cada conta. É natural que não possamos criar contas sem titular, logo no construtor sempre será necessário fornecer um

nome (`String`).

```
class Conta
{
    String titular;
    double saldo;

    Conta(String s, double val)
    {
      titular = s;
      saldo = val;
    }

    void imprime()
    {
      System.out.println("O cliente: " + titular + " tem saldo " + saldo);
    }

    public static void main(String []args)
    {
      Conta c1 = new Conta("José", 100);
      Conta c2 = new Conta("Maria", 1000);

      c1.imprime();
      System.out.println();
      c2.imprime();
    }
}
```

Mesmo no exemplo bem simples acima podemos notar que uma operação usual é a criação de contas com saldo zero. Logo, parece natural que tenhamos um construtor que receba apenas o nome, e não o saldo (ficando subentendido neste caso que não houve depósito inicial).

```
    Conta(String s)
    {
      titular = s;
      saldo = 0;
    }
```

Pode-se observar também que podemos reescrever o construtor acima usando funcionalidades do primeiro construtor.

```
    Conta(String s)
    {
      this(s, 0.0);
    }
```

Para isso usamos uma palavra reservada da linguagem Java, a palavra `this`, que nada mais é do que uma referência ao próprio objeto. Isto é, quando o construtor `Conta("João")` é chamado, ele repassa o trabalho ao outro construtor usando saldo 0.

Também é interessante notar a diferença entre os dois construtores: como eles têm o mesmo nome (nome da classe), a única diferença entre eles está nos parâmetros recebidos. O compilador Java escolhe o construtor

correto conforme a assinatura (parâmetros e sua ordem). Esta mesma técnica também pode ser usada se precisarmos de dois métodos com o mesmo nome, mas que recebam parâmetros diferentes. Isto é chamado de polimorfismo de nome.

Também é interessante ressaltar um outro uso da palavra chave `this`. Vamos supor que adicionamos um novo parâmetro para as contas, o RG. Logo, teremos um atributo a mais: `String RG`. Se por acaso quisermos criar um novo construtor que receba um parâmetro também chamado de `RG`, isto se torna possível.

```
Conta(String s, double val, String RG)
{
  titular = s;
  saldo = val;
  this.RG = RG;
}
```

No caso, o `this` reforça que estamos falando de um dos atributos do objeto. Até o momento não havíamos usado o `this` pois tomamos o cuidado de escolher os nomes de parâmetros e variáveis locais diferentes dos nomes dos atributos.

19.2 Especificadores de acesso

Continuando com o exemplo anterior, poderíamos adicionar métodos seguros para efetuar depósitos e saques.

```
void saque(double val)
{
  if (val <= saldo){
    System.out.println("Saque efetuado com sucesso");
    saldo = saldo - val;
  } else
    System.out.println("Saldo INSUFICIENTE");
    imprime();
}

void deposito(double val)
{
  saldo = saldo + val;
  imprime();
}
```

No código acima, no caso de saque, veríamos mensagens avisando sobre o sucesso ou falha na operação. Existem outras formas mais bonitas de mostrar as falhas através de exceções (`Exceptions`), mas estas fogem ao escopo deste livro. Entretanto, um usuário poderia autorizar saques quaisquer, da seguinte forma:

```
Conta c2 = new Conta("Maria", 1000);

c2.imprime();
c2.saldo = c2.saldo - 100000;
c2.imprime();
```

Pois, neste caso, o usuário do objeto do tipo `Conta` estaria interagindo diretamente com atributos do objeto, mexendo nas suas partes internas. Para evitar isto podemos usar proteções explícitas, indicando que só métodos

do próprio objeto possam alterar os seus atributos. Isto é feito com a palavra chave `private`, como neste exemplo:

```
class Conta
{
    private String titular;
    private double saldo;
    private String RG;
  ...
}
```

Neste caso, não existe mais o acesso direto aos atributos de objetos da classe `Conta`. O oposto de `private` é `public` que dá acesso irrestrito. Os especificadores de acesso também podem ser usados com os métodos. Quando nenhum especificador é utilizado, a linguagem Java usa o acesso amigável (*friendly*) que só permite a visibilidade dentro do mesmo pacote (por enquanto pode se pensar em pacote como sendo equivalente a um diretório).

Um outro qualificador interessante é o `static` que indica quais atributos devem ser considerados como da classe e não específicos a cada objeto. Veja o seguinte exemplo:

```
class TesteStatic {
  static int quantidade = 0;

  TesteStatic()
  {
    System.out.println("Criando um objeto do tipo TesteStatic");
    quantidade++;
    System.out.println("Até agora foram criados: " + quantidade + "
                  objetos TesteStatic");
  }
}
```

A cada objeto `TesteStatic` criado veremos quantos objetos deste tipo já foram criados anteriormente. Observem que não é uma prática usual colocar impressões no construtor, mas de forma a apresentar um exemplo simples optamos por tomar esta liberdade. Além disto, métodos `static` também são métodos de classe, isto é, podem ser chamados, mesmo que não existam objetos da classe criados. Um exemplo já visto são as funções da classe `Math` como, por exemplo, `Math.sin(double x)`. O método `main` também é estático pois ao iniciarmos a execução de um programa não existe objeto criado.

Finalmente, um último qualificador interessante é o `final` que serve para definir variáveis que não podem mais ter o seu valor modificado, como, por exemplo, `final double PI = 3.1415926538;`

Exercício

1. Reescreva a classe `CalculadoraDeImposto` descrita no exercício 2 do Capítulo 7, contendo um método que não recebe nenhum parâmetro e devolve o imposto a ser pago. A classe `CalculadoraDeImposto` deve possuir em seus atributos as classes `Rendimentos` e `TabeladeAlíquotas` e receber como parâmetros em seu construtor todos os valores necessários para inicializar objetos destas classes. Compare a abordagem do exercício do Capítulo 7 com a deste exercício.

2. Suponha que você possua um programa de edição de imagens que manipula objetos geométricos. Você deseja definir uma classe `Quadrado` contendo como atributos um inteiro `cor` e um `String tamanho`, definidos pelo usuário no momento de sua criação, mas que não podem ser posteriormente modificados. Como você escreveria esta classe?

3. Modifique a classe `Quadrado` do Exercício 2 de modo que a cor e o tamanho do objeto possam ser alterados após sua atribuição inicial. Além disso, a classe deve garantir que sejam atribuídos, para o campo `tamanho`, apenas valores positivos e, para o campo `cor`, apenas as cores "amarelo", "azul", "verde" ou "vermelho".

4. (difícil): Como fazer para criar uma classe na qual só seja possível construir um único objeto. Este tipo de objeto é chamado de *Singleton*.
   Dica: Limitar o acesso ao construtor.

Capítulo 20

Interfaces

Quais novidades veremos neste capítulo?

- Interfaces.

Interface: local onde dois sistemas independentes se encontram, atuam ou se comunicam. *Webster*.

20.1 O conceito de interfaces

Um dos conceitos principais de orientação a objetos é o *encapsulamento*, através do qual, tanto os atributos quanto a implementação dos métodos de uma certa classe não são visíveis ao usuário da classe. Conhecendo-se apenas a *interface* de uma classe, isto é, os métodos disponíveis e suas respectivas assinaturas, podemos utilizar objetos desta classe sem conhecer detalhes de como ela é implementada internamente.

Além disto, existem casos, onde existe a necessidade de se ter uma classe mas não queremos implementá-la. Neste caso, pode-se terceirizar a implementação, fornecendo como especificação a interface desejada.

20.2 Um primeiro exemplo

Vejamos um exemplo prático: você tem a missão de criar um zoológico virtual com vários tipos de animais. Você gostaria de enviar as seguintes mensagens a cada animal:

- `nasça();`
- `passeiePelaTela();`
- `durma();`

Mas, apesar de ser especialista em computação, você conhece muito pouco a respeito de animais, logo você terá que pedir a outros programadores, que conhecem bem os animais, as seguintes classes: `Ornitorrinco`, `Morcego` e `Zebra`. Neste caso, você passará a seguinte especificação:

```
interface Animal
{
   void nasça();
   void passeiePelaTela();
   void durma();
   double peso();
}
```

O programador que for implementar o morcego terá que dizer explicitamente que vai usar a interface `Animal`, o que é feito através da palavra chave `implements`. Como o objetivo é apresentar como funcionam as interfaces, o código dos animais será apenas composto de comandos de impressão de mensagens.

```
public class Morcego implements Animal
{
   public void nasça()
   {
     System.out.println("Nasce um lindo morcego");
   }
   public void passeiePelaTela()
   {
     System.out.println("Voa de um lado para o outro");
   }
   public void durma()
   {
     System.out.println("Dorme de ponta cabeça");
   }
   public double peso()
   {
     return 4.5; // morcegão :-)
   }
}
```

A palavra chave `implements` obriga o programador a escrever o código correspondente a todos os métodos com suas respectivas assinaturas. Além disto, todos os métodos da interface devem ser obrigatoriamente públicos. Vejamos as implementações das outras classes:

```
public class Ornitorrinco implements Animal
{
   double peso;

   Ornitorrinco (double p)
   {
     peso = p;
   }

   public double peso()
   {
     return peso;
   }
   public void nasça()
   {
     System.out.println("Quebra o ovo para sair");
   }
```

```
    public void passeiePelaTela()
    {
      System.out.println("Anda e nada de um lado para o outro");
    }
    public void durma()
    {
      System.out.println("Dorme dentro de túneis, durante o dia");
    }
}

public class Zebra implements Animal
{
   int listras;
   double peso;

   public Zebra(int l, double p)
   {
      listras = l; // cria uma zebra com l listras
      peso = p     // e peso p
   }
   public void nasça()
   {
     System.out.println("Nasce mais uma zebra");
   }
   public void passeiePelaTela()
   {
     System.out.println("Galopa pelo campo");
   }
   public void durma()
   {
     System.out.println("Dorme em pé");
   }
   public double peso()
   {
     return peso;
   }
   // nada impede que sejam implementados métodos adicionais
   public void contaListras()
   {
     System.out.println("Esta zebra tem " + l + " listras  " );
   }
}
```

Em tempo: Existe uma regra em Java com relação ao número de classes públicas que podem existir em um arquivo .java. Em cada arquivo deve existir no máximo uma classe pública, sendo que, caso exista uma, o nome do arquivo deve ser igual ao nome da classe pública. Logo, no exemplo acima, as classes `Ornitorrinco`, `Morcego` e `Zebra` devem estar em arquivos separados, com os respectivos nomes: Ornitorrinco.java, Morcego.java e Zebra.java.

Mas, o uso da interface é um pouco mais amplo, pois podemos considerar que cada um dos animais além de ser um objeto da própria classe também é um objeto do tipo `Animal`. É interessante ressaltar que não podemos criar novos objetos a partir da interface `Animal`. Vejamos mais um exemplo:

```
class ZoológicoVirtual
{
  static public void cicloDeVida(Animal animal)
  {
    animal.nasça();
    animal.passeiePelaTela();
    animal.durma();
  }

  static public void fazFuncionar()
  {
    Zebra         z1 = new Zebra(102, 99);  // cria duas zebras
    Animal        z2 = new Zebra(101, 107);  // sendo uma do tipo Animal
    Morcego       m1 = new Morcego();
    Ornitorrinco o1 = new Ornitorrinco(25);

    cicloDeVida(z1);
    cicloDeVida(z2);
    cicloDeVida(m1);
    cicloDeVida(o1);
  }
}
```

Veja o exemplo do painel de iterações abaixo:

```
> ZoológicoVirtual.fazFuncionar()
Nasce mais uma zebra
Galopa pelo campo
Dorme de pé
Nasce mais uma zebra
Galopa pelo campo
Dorme de pé
Nasce um lindo morcego
Voa de um lado para o outro
Dorme de ponta cabeça
Quebra o ovo para sair
Anda e nada de um lado para o outro
Dentro de túneis, durante o dia
>
```

Observe que apesar de `z2` ter sido definido como uma nova `Zebra`, a referência é para um objeto do tipo `Animal`, de modo que chamadas do tipo `z2.contaListras()` não são válidas, mas chamadas `z1.contaListras()` o são.

Na verdade, seria interessante refatorarmos o método `fazFuncionar()` da seguinte forma.

```
  static public void fazFuncionar()
```

```
  {
    Animal [] bicharada = new Animal [4];
    bicharada[0] = new Zebra(102, 99);
    bicharada[1] = new Zebra(101, 107);
    bicharada[2] = new Morcego();
    bicharada[3] = new Ornitorrinco(25);

    for (int i = 0; i < bicharada.length; i++)
      cicloDeVida(bicharada[i]);
  }
```

Ficou bem melhor, não? Dá para melhorar um pouco mais ainda:

```
  static public void fazFuncionar()
  {
    Animal [] bicharada =
     { new Zebra(102, 99), new Zebra(101, 107),
       new Morcego(),      new Ornitorrinco(25) }

    for (int i = 0; i < bicharada.length; i++)
      cicloDeVida(bicharada[i]);
  }
```

20.3 Implementando mais de uma interface por vez

Vimos acima que podemos ver objetos como sendo do mesmo tipo, desde que eles implementem a mesma interface. Isto também é válido no caso de objetos implementarem várias interfaces (pensando no mundo real isto acontece muito mais freqüentemente). Vejam as duas interfaces seguintes:

```
interface Voador
{
  void voa();
  void aterrissa();
}

interface TransportadorDePessoas
{
  void entramPessoas();
  void saemPessoas();
}
```

Agora vamos pensar em três classes: `Ave`, `Ônibus` e `Avião`. As classes `Ave` e `Ônibus` podem implementar a primeira e segunda interface, respectivamente.

```
class Ave implements Voador
{
  public void voa()
  {
    System.out.println("Bate as asas bem forte");
  }
  public void aterrissa()
  {
```

```
    System.out.println("Bate as asas mais fraco e põe os pés no chão");
  }
}

class Ônibus implements TransportadorDePessoas
{
  public void entramPessoas()
  {
    System.out.println("Abre as portas e entram as pessoas");
  }
  public void saemPessoas()
  {
    System.out.println("Abre as portas e saem as pessoas");
  }
}
```

Finalmente, podemos ver o `Avião` que implementa as duas interfaces:

```
class Avião implements Voador, TransportadorDePessoas
{
  public void voa()
  {
    System.out.println("Liga as turbinas; recolhe o trem de pouso");
  }
  public void aterrissa()
  {
    System.out.println("Abaixa o trem de pouso e desce");
  }
  public void entramPessoas()
  {
    System.out.println("Procedimento de embarque");
  }
  public void saemPessoas()
  {
    System.out.println("Procedimento de desembarque");
  }
}
```

Observe o trecho abaixo:

```
public class TesteDeInterface
{
  static public void faz()
  {
    TransportadorDePessoas t = new Ônibus();
    Voador v = new Ave();
    Avião  a = new Avião();

    t.entramPessoas();
    t.saemPessoas();
    v.voa();         // bate asas
    v.aterrissa();
    // Com o Ônibus e Ave não podemos chamar a outra interface.
    a.entramPessoas();
    a.voa();
```

```
      a.aterrissa();
      a.saemPessoas();
      v = a;
      v.voa();          // liga turbinas
    }
  }
```

Uma boa prática seguida por bons programadores OO é "Programe para as interfaces, não para as implementações". Em outras palavras, toda vez que você escrever código que utiliza outras classes, não pense em como essas outras classes são implementadas internamente, pense apenas na sua interface. Nunca baseie o seu código em alguma idiossincrasia interna da classe, use apenas conceitos que são claros a partir das interfaces das classes que você usa.

20.4 Um exemplo mais sofisticado

Vamos supor que temos uma classe `Fruta` com os seguintes atributos: `peso`, `preço` e `nome`. Os três atributos já devem ser carregados no construtor.

```
class Fruta
{
  double peso;
  double preço;
  String nome;

  Fruta(String n, double v, double p)
  {
    nome  = n;
    preço = v;
    peso  = p;
  }

  void imprime()
  {
    System.out.println(nome + " pesa " + peso
                       + "gramas e custa " + preço + "reais");
  }
}
```

Queremos criar um vetor de `Frutas` e ordená-lo primeiro por `preço` e posteriormente por `peso`. Como fazer isto? Observando os procedimentos de ordenação já vistos é fácil ver que a única mudança é o critério de comparação. Veja uma implementação usando o algoritmo de inserção direta visto no Capítulo 17.

```
public class Quitanda
{
   Fruta []frutas = new Fruta[5];

   public Quitanda()
   {
     frutas[0] = new Fruta("Laranja", 0.5, 100);
     frutas[1] = new Fruta("Maçã", 0.8, 120);
     frutas[2] = new Fruta("Mamão", 1.2, 110);
     frutas[3] = new Fruta("Cereja", 5.0, 20);
```

```
    frutas[4] = new Fruta("Jaca", 0.4, 500);
  }

  public void imprime()
  {
    for(int i = 0; i < frutas.length; i++)
      frutas[i].imprime();
  }

  public void ordenaPorPreço()
  {
    int i, j;
    Fruta aInserir;

    for (i = 1; i < frutas.length; i++)
    {
      aInserir = frutas[i];
      j = i;
      while((j > 0) && (frutas[j-1].preço > aInserir.preço))
      {
         frutas[j] = frutas[j-1];
         j--;
      }
      frutas[j] = aInserir;
    }
  }

  public void ordenaPorPeso()
  {
    int i, j;
    Fruta aInserir;

    for (i = 1; i < frutas.length; i++)
    {
      aInserir = frutas[i];
      j = i;
      while((j > 0) && (frutas[j-1].peso > aInserir.peso))
      {
         frutas[j] = frutas[j-1];
         j--;
      }
      frutas[j] = aInserir;
    }
  }

  public static void main(String [] args)
  {
    Quitanda xepa = new Quitanda();

    System.out.println("Desordenado");
    xepa.imprime();
    System.out.println("Em ordem de preço");
    xepa.ordenaPorPreço();
    xepa.imprime();
```

```
        System.out.println("Em ordem de peso");
        xepa.ordenaPorPeso();
        xepa.imprime();
    }
}
```

No programa acima, fica claro que tivemos que duplicar o código de ordenação, o que é bem desagradável. Mas, e se existissem outros critérios para ordenação, teríamos que criar um novo método repetindo o código de ordenação para cada um dos critérios? Isso seria, com certeza, muito ruim.

Como regra geral, devemos sempre evitar código repetido. Quando identificamos um trecho de código que aparece repetido em vários lugares, como no caso acima, devemos tentar refatorar (reorganizar) o código para evitar a repetição. Neste caso, iremos manter o código de ordenação em um método e colocar em outros métodos apenas aquele pequeno trecho que difere de um para outro (o critério de comparação). Para implementar esta nova solução, utilizaremos uma `interface` definindo a interface de comparação:

```
interface ComparadorDeFrutas
{
  boolean éMenor(Fruta a, Fruta b);
}
```

O significado do método é óbvio: se `a` for menor que `b`, o método devolve `true`, caso contrário, devolve `false`. O que a interface `ComparadorDeFrutas` não define é o significado da palavra "menor". Isso é deixado para cada classe concreta que irá implementar esta interface, como vemos abaixo, onde são definidas três implementações diferentes para `ComparadorDeFrutas` que utilizam, como critério de comparação, o peso, o preço e o nome da fruta, respectivamente.

```
class ComparaPeso implements ComparadorDeFrutas
{
  public boolean éMenor(Fruta a, Fruta b){ return (a.peso < b.peso);}
}

class ComparaPreço implements ComparadorDeFrutas
{
  public boolean éMenor(Fruta a, Fruta b){ return (a.preço < b.preço);}
}

class ComparaNome implements ComparadorDeFrutas
{
  public boolean éMenor(Fruta a, Fruta b){ return (a.nome.compareTo(b.nome) < 0);}
}
```

Agora, basta colocar como parâmetro adicional do método de ordenação o comparador desejado:

```
public class Quitanda
{
    Fruta []frutas = new Fruta[5];

    public Quitanda()
    {
      frutas[0] = new Fruta("Laranja", 0.5, 100);
      frutas[1] = new Fruta("Maça", 0.8, 120);
      frutas[2] = new Fruta("Mamão", 1.2, 110);
```

```
        frutas[3] = new Fruta("Cereja", 5.0, 20);
        frutas[4] = new Fruta("Jaca", 0.4, 500);
    }

    public void imprime()
    {
      for(int i = 0; i < frutas.length; i++)
        frutas[i].imprime();
    }

    public void ordena(ComparadorDeFrutas c)
    {
      int i, j;
      Fruta aInserir;

      for (i = 1; i < frutas.length; i++)
      {
        aInserir = frutas[i];
        j = i;
        while((j > 0) && (c.éMenor(aInserir, frutas[j-1])))
          {
            frutas[j] = frutas[j-1];
            j--;
        }
        frutas[j] = aInserir;
      }
    }

    public static void main(String [] args)
    {
      Quitanda xepa = new Quitanda();

      System.out.println("Frutas desordenadas");
      xepa.imprime();

      System.out.println("Em ordem de preço:");
      ComparadorDeFrutas cmp = new ComparaPreço();
      xepa.ordena(cmp);
      xepa.imprime();
      System.out.println("Em ordem de peso:");
      cmp = new ComparaPeso();
      xepa.ordena(cmp);
      xepa.imprime();
      System.out.println("Em ordem alfabética:");
      xepa.ordena(new ComparaNome()); // note esta forma supercondensada.
      xepa.imprime();
    }
}
```

20.5 A importância de interfaces

O conceito de interfaces na programação orientada a objetos é muito importante e o seu uso adequado traz inúmeras vantagens no desenvolvimento de sistemas grandes e complexos. Eis algumas dessas vantagens:

1. Se os objetos interagem entre si através de referências a interfaces e não a classes específicas, fica fácil mudar as classes utilizadas em um sistema sem interferir com aquelas que as utilizam. Por exemplo, se uma classe implementa uma certa interface, mudanças na implementação da classe que não alterem a assinatura dos métodos desta interface não são notadas ao se utilizar objetos através desta interface.

2. Fica fácil implementar algo chamado *polimorfismo de comportamento*, ou seja, podemos utilizar em diferentes momentos classes com diferentes comportamentos, chaveando de um comportamento para outro quando for necessário. O uso de interfaces permite que este chaveamento seja feito tanto quando o programa é compilado quanto durante a sua execução. O exercício do berçário (veja a seção de exercícios a seguir) é um exemplo deste polimorfismo onde interfaces são utilizadas para tornar um programa multilingüe, ou seja, com muito pouco esforço pode-se escolher qual língua (português, inglês, francês, etc.) um programa ou sistema utilizará para interagir com seus usuários.

3. Se se está desenvolvendo um programa ou sistema muito grande e complexo, pode ser necessário utilizar vários programadores ou, até mesmo, várias equipes de programadores. Neste caso, o que se costuma fazer é dividir o sistema em sub-sistemas e definir muito bem as interfaces de cada sub-sistema. A partir daí, cada equipe pode desenvolver independentemente o seu sub-sistema sem necessitar de conhecimentos sobre o funcionamento interno dos outros sub-sistemas; basta conhecer as suas interfaces. Após esta fase, passa-se à fase de integração, durante a qual os vários sub-sistemas são compilados, executados e testados em conjunto.

4. O uso de interfaces pode ajudar a eliminar código repetido (como no exemplo anterior do `ComparadorDeFrutas`), o que ajuda a melhorar muito a qualidade do código.

5. Usando-se interfaces, é possível encomendar a terceiros a escrita de partes de um programa sem que esta pessoa tenha que conhecer o resto do programa ou mesmo ter acesso ao seu código-fonte. Veja um exemplo desta prática no exercício das figuras geométricas abaixo, que se utiliza de uma interface `Figura2D`.

Exercícios

1. Seu chefe está fazendo um programa para manipulação de figuras geométricas. Como o programa é muito complexo, ele pediu a sua ajuda encomendando a você a implementação de três classes (`Quadrado`, `Retângulo` e `Círculo`). Você é livre para definir como será a implementação mas seu chefe definiu que as suas classes devem implementar a seguinte interface:

```
interface Figura2D
{
    double calculaÁrea();
    double calculaPerímetro();
    void   mudaCor(String cor);
    String pegaCor();
```

```
}
```

2. Escreva duas implementações para a interface `VeículoDeCorrida`, a seguir:

```
interface VeículoDeCorrida
{
    String marca();
    String modelo();
    String cor();
    int potênciaEmCavalos();
}
```

Agora, escreva um método `veículoPreferido()` que recebe um *array* de veículos de corrida como parâmetro e, dentre os veículos vermelhos, imprime a marca e o modelo do que possuir a maior potência.

3. Você foi contratado para trabalhar em um berçário (!!!) e no seu primeiro dia de trabalho, deve escrever uma classe para informar aos pais dados sobre os seus bebês. A dificuldade é que muitos estrangeiros freqüentam esse berçário e o seu programa deve ser capaz de dar informações em português e em inglês (e, futuramente, em outros idiomas também). Para permitir isso, você deverá prover duas implementações da interface a seguir, que indica as mensagens de texto que deverão ser mostradas aos pais:

```
interface MensagensSobreNeoNatal
{
   String nomeDoBebê(String nome);
   String dataDeNascimento(Bebê b);
   String peso(double pesoEmQuilos);
   String temperatura(double temperaturaEmCelsius);
}
```

Ao se chamar o método `nomeDoBebê`, por exemplo, sua implementação deve devolver uma mensagem como "O nome do bebê é Godofredo Manoelino de Moraes". Ao se chamar o método `peso`, deve-se devolver algo como "O peso do bebê é 3140 gramas ". A primeira implementação deve se chamar MensagensBrasileiras e a segunda MensagensEstadosunidenses. As mensagens para os americanos devem apresentar o peso em libras (*pounds*) e a temperatura em graus fahrenheit. A interface Bebê, que vocês não precisam implementar, pois podem supor que o berçário já possui as implementações, é a seguinte:

```
interface Bebê
{
   String nome();
   int diaNascimento();
   int mesNascimento();
   int anoNascimento();
   double peso(); // SI, ou seja, em quilos
   double temperatura(); // SI, ou seja, em celsius
}
```

A classe que o berçário vai usar para imprimir as informações para os pais terá um método similar ao seguinte:

```
class Berçário
{
```

```
  Bebê [] listaDeBebês;

  // aqui vão outros métodos para inserção e
  // remoção de bebês da lista

  void imprimeDadosSobreBebê(Bebê b, MensagensSobreNeoNatal m)
  {
     System.out.println(m.nomeDoBebê (b.nome ());
     System.out.println(m.dataDeNascimento (b);
     System.out.println(m.peso (b.peso ());
     System.out.println(m.temperatura (b.temperatura ());
  }

  void imprime(int IDdoBebê, String idioma)
  {
     MensagensSobreNeoNatal m;
     if (idioma.equals ("Português"))
        m = new MensagensBrasileiras ();
     else
        m = new MensagensEstadosunidenses ();

     imprimeDadosSobreBebê(listaDeBebês[IDdoBebê], m);
  }
}
```

4. Dada a classe

```
class Pessoa
{
  String nome;
  int rg;
  int cpf;
}
```

escreva duas implementações da interface

```
interface LocalizadorDePessoas
{
  Pessoa localizaPorRG(int rg, Pessoa []vp);
}
```

utilizando busca seqüencial e busca binária.

5. Refatore o método `ZoológicoVirtual.fazFuncionar()` de forma a usar a classe `Scanner` para permitir que um usuário crie interativamente uma coleção de animais virtuais de diferentes tipos.

6. (Longo) Você foi contratado para ajudar na implementação de uma loja virtual para venda, através da Web, de livros, CDs, LPs, DVDs, filmes VHS, etc. No seu primeiro dia de trabalho, seu chefe lhe mostrou a seguinte interface que é utilizada para representar todos os produtos a serem vendidos pela loja:

```
interface Produto
```

```
{
  String título();
  int    númeroDeIdentificação(); // número único identificando o produto
  int    ano();
  String autor();
  String mídia(); // devolve "livro", "CD", "DVD", etc.
}
```

No seu primeiro dia de trabalho, você deve implementar uma classe contendo métodos para:

(a) dada uma lista de compras (um *array* de `Produtos`), devolver uma nova lista de compras contendo o mesmo conteúdo mas com os produtos ordenados de acordo com os seus `númerosDeIdentificação`;

(b) dadas duas listas de compras, cada uma delas já ordenadas com o método acima, gerar e devolver uma nova lista ordenada resultante da fusão das duas listas iniciais; elementos que aparecerem nas duas listas iniciais deverão aparecer duplicados na lista final.

(c) Agora, escreva pelos menos duas classes que implementem a interface `Produto` e, em seguida, dê um exemplo de código que cria duas listas de compras, as ordena, as funde, e finalmente imprime de forma clara e organizada as informações sobre os produtos contidos na lista resultante da fusão.

Capítulo 21

Herança

Quais novidades veremos neste capítulo?

- Herança.

21.1 O Conceito de herança

A demanda pela construção de software em nossa sociedade é muito grande. O número de bons programadores disponíveis é relativamente pequeno. Por isso, os projetistas de linguagens e sistemas de computação têm buscado formas de acelerar e facilitar o desenvolvimento de software. Uma das principais abordagens utilizadas é a reutilização de código. Se conseguirmos escrever código que é reutilizado em diversos programas diferentes, teremos economizado tempo pois não será necessário reescrever este código para cada novo programa.

A biblioteca de classes de Java é um exemplo de classes que foram escritas pelos programadores da Sun e que são reutilizadas diariamente por milhares de programadores em todo o mundo. Porém, algumas vezes, o programador necessita fazer alguma modificação em uma classe existente pois ela não atende exatamente às necessidades de um dado programa. Em alguns casos deseja-se modificar a implementação de um de seus métodos, em outros casos deseja-se acrescentar alguma funcionalidade extra à classe, com a adição de novos métodos e atributos.

Esta reutilização de código com extensões e modificações, pode ser obtida em linguagens orientadas a objetos através do conceito de Herança. A figura 21.1 mostra um diagrama contendo 4 classes que se relacionam entre si através de herança.

Este diagrama de classes segue um padrão chamado UML (*Unified Modeling Language*). Este padrão é muito utilizado para modelar programas orientados a objetos, mas não veremos mais detalhes sobre ele neste livro. Apresentamos o diagrama aqui apenas a título de ilustração.

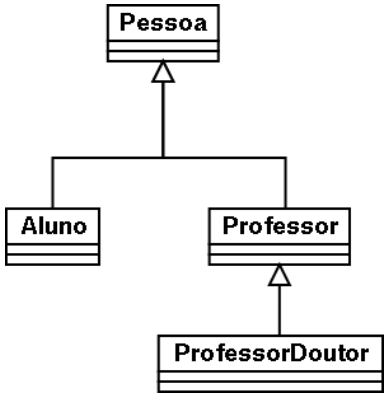


Figura 21.1: Diagrama de herança

21.2 Terminologia de herança

- Dizemos que `Pessoa` é a ***superclasse*** de `Aluno` e de `Professor`.
- `Professor` é ***subclasse*** de `Pessoa` e ***superclasse*** de `ProfessorDoutor`.
- Dizemos ainda que `Aluno` ***herda*** de `Pessoa` os seus atributos e métodos; ou que `Aluno` ***estende*** a classe `Pessoa`.
- Dizemos que `Pessoa` é ***pai*** (ou ***mãe***) de `Professor` e que `Professor` é ***filho*** (a) de `Pessoa`.

21.3 Implementação de herança na linguagem Java

Para especificar, em Java, que uma classe B é subclasse de A, utilizamos a palavra `extends`:

```
class A
{
  int a1;
  void pata(){ a1 = 1; }
}

class B extends A
{
  int b1;
  void vina()
  {
    pata();
    b1 = a1;
  }
}
```

Note que `B` contém os métodos e atributos de `A` e ainda acrescenta um novo atributo e um novo método.

Vejamos agora um exemplo mais complexo que implementa, em Java, o diagrama de classes apresentado no diagrama UML acima.

```
class Pessoa
{
   private String nome;
   private char sexo;
   private String CPF;
   private String RG;
   private int anoDeNascimento;
   void imprimeDados()
   {
     if (sexo == 'F')
       System.out.println("A Sra. " + nome + "nasceu no ano" + anoDeNascimento +
                          ". CPF: " + CPF + ", RG " + RG);
     else
       System.out.println("O Sr. " + nome + "nasceu no ano" + anoDeNascimento +
                          ". CPF: " + CPF + ", RG " + RG);
   }
}

class Aluno extends Pessoa
{
   private String curso;
   private int anoDeIngresso;
   void imprimeDados()
   {
    super.imprimeDados();
    System.out.println("Ingressou no curso " + curso + " em " +
                       anoDeIngresso);
   }
}

class Professor extends Pessoa
{
   private String departamento;
   private int anoDeAdmissão;
   void imprimeDados()
   {
    super.imprimeDados();
    System.out.println("Ingressou no dept. " + departamento + " em " +
                       anoDeAdmissão);
   }
}

class ProfessorDoutor extends Professor
{
   private int anoDeObtençãoDoutorado;
   private String instituiçãoDoDoutorado;
   void imprimeDados()
   {
    super.imprimeDados();
    System.out.println("Doutorado obtido em " +  instituiçãoDoDoutorado
                       + " em " + anoDeObtençãoDoutorado);
   }
}
```

Um banco de dados da universidade pode armazenar objetos do tipo `Pessoa` da seguinte forma:

```
class ListaDePessoasDaUSP
{
   Pessoa [] membrosDaUSP;

   ListaDePessoasDaUSP(int tamanho)
   {
      membrosDaUSP = new Pessoa [tamanho];
   }

   // métodos para acrescentar e remover pessoas da lista de pessoas

   void listaTodos()
   {
     int i;
     for (i = 0; i < membrosDaUSP.length; i++)
        membrosDaUSP[i].imprimeDados();
   }

    // demais métodos...
}
```

Note que, para o método `listaTodos`, não interessa qual o tipo específico de cada pessoa (aluno, professor, professor doutor) uma vez que ele manipula apenas os atributos da superclasse `Pessoa`.

21.4 Hierarquia de classes

Quando um programa possui uma série de classes relacionadas através de herança, dizemos que temos uma *hierarquia de classes*.

Apresentamos na figura 21.2 uma hierarquia de classes para representar os diversos tipos de seres vivos.

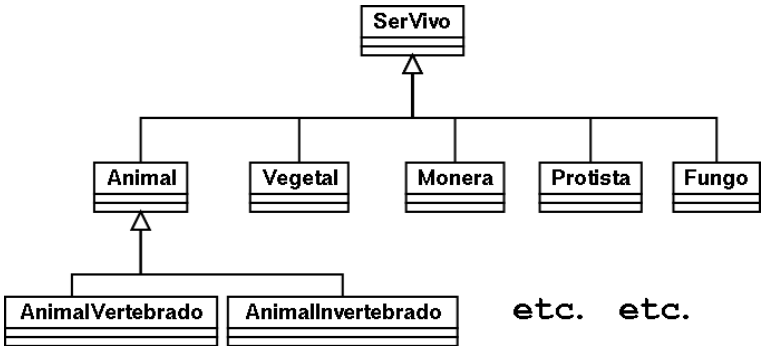


Figura 21.2: Hierarquia de classes representando os seres vivos

21.5 Relacionamento “é um”

Nem sempre é fácil determinarmos qual hierarquia de classes devemos utilizar. A relação superclasse-subclasse deve necessariamente ser um relacionamento do tipo "é um", ou seja, se `B` é subclasse de `A` então todo `B` é um `A`.

Em termos mais concretos, no exemplo dos seres vivos, `Animal` é um `SerVivo`, e mais, todo animal é um ser vivo. Não existe nenhum animal que não seja um ser vivo. Então o relacionamento superclasse-subclasse pode ser apropriado.

Outra possibilidade seria ter uma hierarquia como a seguinte:

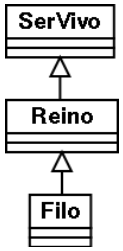


Figura 21.3: Hierarquia errada

A princípio, esta pode parecer uma hierarquia aceitável. Mas na verdade ela está ERRADA. Não podemos dizer que "um reino é um ser vivo", não faz sentido, não podemos dizer que "um filo é um reino"porque não é. Então a hierarquia não está boa.

21.6 Resumo

- uma classe pode herdar de outra seus atributos e métodos;
- uma subclasse pode estender a funcionalidade de sua superclasse acrescentando novos atributos e métodos.

Mas, e se uma subclasse implementar um método com assinatura idêntica a um método da superclasse? Neste caso, quem prevalece é o método da subclasse e dizemos que o método da subclasse ***se sobrepõe ("overrides")*** ao método da superclasse.

Exercícios

1. Retornando à hierarquia de classes apresentada na figura 21.2, pense em quais métodos e atributos deveriam estar presentes nas classes superiores da hierarquia e quais deveriam estar nas partes inferiores da hierarquia.
2. Figuras geométricas são um bom exemplo para se construir um hierarquia de classes. Considere as figuras quadrado, retângulo, triângulo (retângulo, acutângulo e obtusângulo) e losango. Construa duas diferentes hierarquias para estas classes, uma com dois níveis e outra com três níveis, colocando os atributos que

devem ser considerados comuns em cada nível da hierarquia. Compare estas duas hierarquias, discutindo suas vantagens e desvantagens.

3. Desenvolva um conjunto de classes para controlar o saldo, depósitos e retiradas de contas bancárias bem como os dados do titular. Escreva inicialmente um diagrama modelando tanto contas corrente quanto contas poupança e aplicações em fundo. Em seguida, implemente estas classes em Java.

Capítulo 22

Javadoc

Quais novidades veremos neste capítulo?

- Documentação com javadoc.

Vimos nos últimos capítulos que um conceito fundamental em programação orientada a objeto é a separação clara entre implementação e interface. Ou seja, um programador que vá usar objetos de uma classe deve se ater à porção pública da mesma, desprezando detalhes de implementação e as porções privadas (que não estariam acessíveis de qualquer forma). Deste modo, passa a ser natural a exigência de documentação de boa qualidade que permita ao usuário de uma classe saber tudo o que precisa sem que necessite ler o código que a implementa. Aí entra o `javadoc`.

O `javadoc` é um programa que permite extrair, de um arquivo com código Java, a sua documentação (texto explicativo). Essa documentação é então formatada em HTML (HyperText Markup Language), que é a mesma linguagem que descreve as páginas da Web, de modo a facilitar a sua consulta. Assim é possível consultar a documentação de uma classe sem as distrações presentes sempre que esta está misturada ao código. Um exemplo disso é que, em sua configuração padrão, o `javadoc` processa apenas a documentação das partes públicas de seu código, omitindo tudo o que é privado.

Para facilitar a vida do programa *javadoc* existem algumas convenções que devem ser seguidas. A primeira é que apenas comentários iniciados por /** (e terminados por */) são processados. Também o comentário de cada parte pública deve precedê-la imediatamente. Vejamos um exemplo simples.

```
public class Quadrado implements FiguraGeométrica
{
  private double lado;
  private String cor;

  public Quadrado(double l, String c)
  {
    lado = l;
    cor = c;
  }

  public double calculaÁrea()
```

```
  {
    return lado*lado;
  }

  public void mudaCor(String c)
  {
    cor = c;
  }
}
```

As partes públicas no código acima são:

1. A própria classe, ou seja, todo mundo pode criar objetos baseados nela;
2. O construtor;
3. Os métodos `calculaÁrea` e `mudaCor`.

Cada um desses ítens deveria ser antecedido por um comentário explicando qual sua função e como ele deve ser usado. Lembrando as convenções de como demarcar os comentários descritas acima, teríamos:

```
/**
 * Uma classe para representar quadrados.
 */
public class Quadrado implements FiguraGeométrica
{
  private double lado;
  private String cor;

  /**
   * Construtor
   * Exige o comprimento do lado e a cor do quadrado.
   */
  public Quadrado(double l, String c)
  {
    lado = l;
    cor = c;
  }

  /**
   * Calcula a área do quadrado baseada no seu lado.
   */
  public double calculaÁrea()
  {
    return lado*lado;
  }

  /**
   * Altera a cor do quadrado para a cor representada no String c.
   */
  public void mudaCor(String c)
  {
    cor = c;
  }
```

```
}
```

Uma outra característica interessante do javadoc é que ele possui alguns marcadores especiais para que possa extrair informações relevantes dos comentários e formatá-las de modo especial. Os marcadores mais importantes são:

- Marcadores para comentários de classes:
  - `@author`: descreve o autor do código;
  - `@version`: usado para guardar informação sobre a versão, como seu número e data de última alteração.
- Marcadores para comentários de métodos:
  - `@param`: usado para descrever os parâmetros, geralmente na forma:
    `@param nome-do-parâmetro descrição`
  - `@return`: serve pra descrever o valor devolvido pelo método.
- Marcador geral: `@see`. Pode ser usado em qualquer lugar para referenciar uma outra classe ou um método de outra classe; assume geralmente uma das duas forma a seguir.
  `@see nome-da-classe`
  `@see nome-da-classe#nome-do-método`

Se usarmos os marcadores descritos acima na nossa classe Quadrado, teremos:

```
/**
 * Uma classe para representar quadrados.
 * @author Paulo Silva
 * @version 1.0, alterada em 10/06/2003
 * @see FiguraGeométrica
 */
public class Quadrado implements FiguraGeométrica
{
  private double lado;
  private String cor;

  /**
   * Construtor
   * @param l double representando o comprimento dos lados
   * @param c String com o nome da cor da figura
   */
  public Quadrado(double l, String c)
  {
    lado = l;
    cor = c;
  }

  /**
   * Calcula a área do quadrado baseada no seu lado
   * @param Não há parâmetros
   * @return área computada
```

```
     */
    public double calculaÁrea()
    {
      return lado*lado;
    }

    /**
     * Altera a cor do quadrado.
     * @param c String com o nome da nova cor
     * @return Não há retorno
     */
    public void mudaCor(String c)
    {
      cor = c;
    }
  }
```

Na figura 22.1 é apresentada a documentação gerada pelo `Javadoc` para a classe `Quadrado`.

É claro que para o marcador `@see` funcionar, você também deve documentar a interface usando os padrões de javadoc. Ao documentar a interface você pode usar os mesmos marcadores usados para classe.

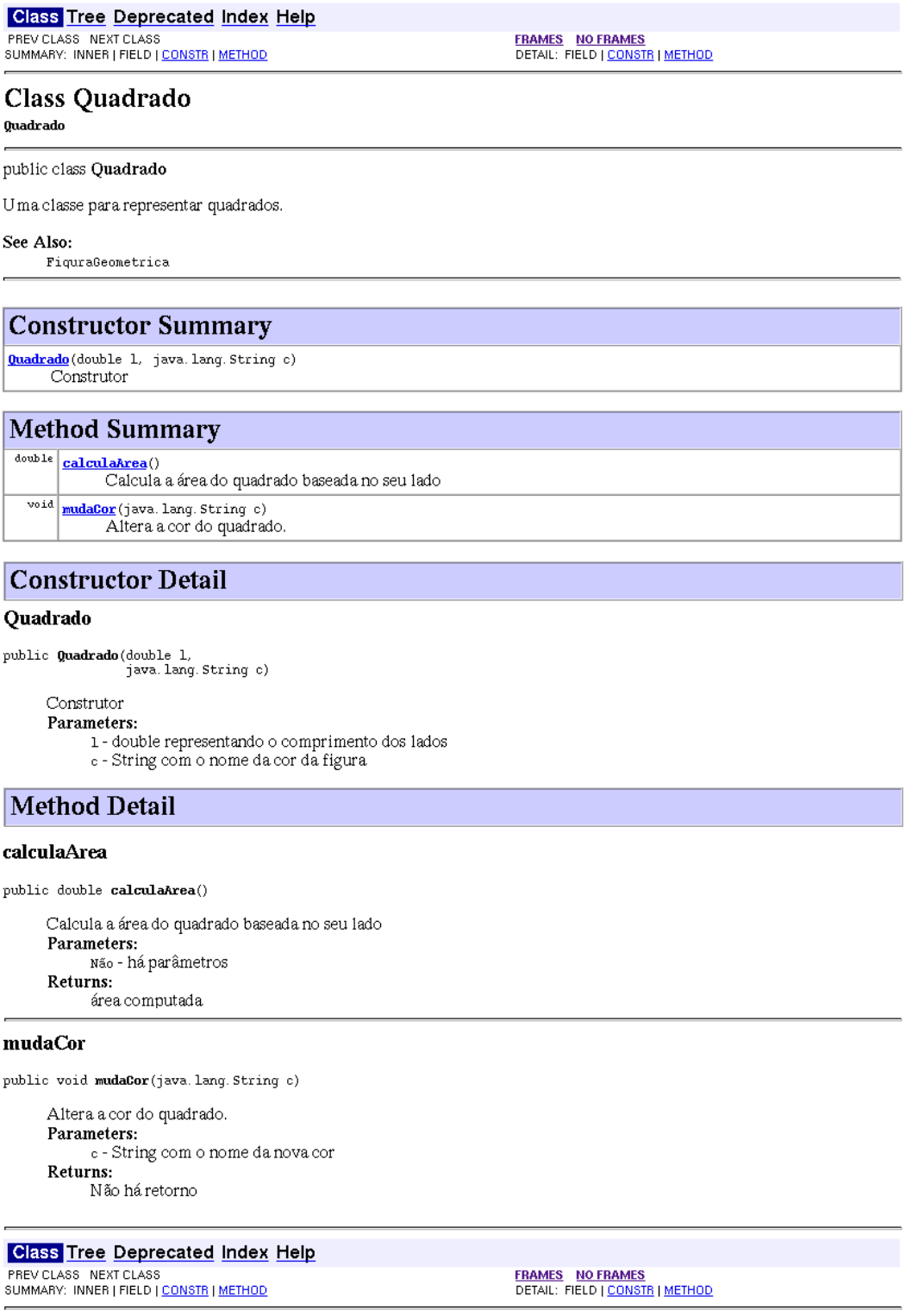
Agora que temos nosso código documentado de forma adequada, como podemos gerar a documentação em html? Para isso, basta ir ao diretório com os arquivos `.java` e digitar

```
javadoc -version -author -d doc *.java
```

A documentação será então gerada e os arquivos resultantes serão colocados dentro do subdiretório `doc` do diretório atual. É claro que o nome desse diretório pode ser alterado mudando-se a palavra que segue o `-doc` presente acima.

Por fim, podemos também documentar as porções privadas de nossas classes. Para forçar o javadoc a gerar documentação também para a parte privada, basta acrescentar o parâmetro `-private` na linha de comando acima.

Obs.: se desejado, os comentários de javadoc podem conter comandos em HTML.



Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Class Quadrado

Quadrado

public class **Quadrado**

Uma classe para representar quadrados.

See Also:

`FiguraGeometrica`

Constructor Summary

Quadrado(double l, java.lang.String c)	Construtor

Method Summary

double	**calculaArea**() — Calcula a área do quadrado baseada no seu lado
void	**mudaCor**(java.lang.String c) — Altera a cor do quadrado.

Constructor Detail

Quadrado

```
public Quadrado(double l,
                java.lang.String c)
```

Construtor

Parameters:

`l` - double representando o comprimento dos lados

`c` - String com o nome da cor da figura

Method Detail

calculaArea

```
public double calculaArea()
```

Calcula a área do quadrado baseada no seu lado

Parameters:

`Não` - há parâmetros

Returns:

área computada

mudaCor

```
public void mudaCor(java.lang.String c)
```

Altera a cor do quadrado.

Parameters:

`c` - String com o nome da nova cor

Returns:

Não há retorno

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Figura 22.1: Documentação gerada pelo `Javadoc`

Capítulo 23

O C Que Há em Java

Quais novidades veremos neste capítulo?

- Veremos como escrever programas na linguagem C a partir de nossos conhecimentos de Java.

23.1 O C que há em Java

Veremos neste capítulo como são os programas na linguagem C e como podemos escrevê-los usando o que aprendemos neste livro. Em resumo, podemos pensar que um programa em C é uma única classe Java sem a presença de atributos e composta apenas de métodos estáticos. Fora isso, o resto é perfumaria. Vamos pensar um pouco em quais são as conseqüências da frase acima:

- como há apenas uma classe e não há atributos não é possível organizar o programa como diversos objetos, eventualmente pertencentes a classes diferentes, interagindo;
- como não há atributos só existem dois tipos de variáveis: os parâmetros e as variáveis locais às funções;
- como não há classes e objetos de verdade, todas as variáveis são de tipos primitivos: `int`, `double` ou `char`. Pelo menos há também a idéia de array, ou vetor, e matrizes em C;
- em C não há o tipo `boolean`. No seu lugar usamos inteiros com o 0 representando falso e qualquer número não nulo representando verdadeiro. Note que as expressões lógicas passam então a gerar valores inteiros como resultados.

Esses pontos já são interessantes o suficiente para vermos o que acontece. Consideremos o primeiro exemplo visto neste livro: o conversor de temperaturas. Você lembra que ele era um objeto sem atributos? Vejamos o seu código (adicionamos um `main` e os identificadores de acesso):

```
public class Conversor
{
  static double celsiusParaFahrenheit (double c)
  {
    return 9.0 * c / 5.0 + 32.0;
  }
  static double fahrenheitParaCelsius(double f)
  {
    return 5.0 * (f - 32.0) / 9.0;
  }
  public static void main(String[] args)
  {
    double far, cel;
    Scanner sc = new Scanner(System.in);

    System.out.print("De um temperatura em Fahrenheit: ");
    far = sc.nextDouble();
    cel = fahrenheitParaCelsius(far);
    System.out.println("A temperatura em Celsius e: " + cel);
    System.out.print("De um temperatura em Celsius: ");
    cel = sc.nextDouble();
    far = celsiusParaFahrenheit(cel);
    System.out.println("A temperatura em Fahrenheit e: " + far);
  }
}
```

Vejamos como ficaria esse programa na linguagem C:

```
/* Sempre coloque a proxima linha no inicio do programa. */
/* Ela permite que voce imprima na tela e leia do teclado. */
#include <stdio.h>

/* Como nao ha classes, simplesmente apagamos a referencia a elas. */
/* Do mesmo modo, como todos os metodos sao estaticos nao          */
/* precisamos escrever isso.                                       */
/* Por fim nao existem especificadores de acesso em C, logo        */
/* eles tambem sumiram.                                            */

double celsiusParaFahrenheit (double c)
{
  return 9.0 * c / 5.0 + 32.0;
}
double fahrenheitParaCelsius(double f)
{
  return 5.0 * (f - 32.0) / 9.0;
}

/* Veja que a cara da main mudou. Agora ela nao recebe nada */
/* e devolve um inteiro.                                    */
int main()
{
  double far, cel;

  /* Oba, imprimir ficou mais facil: o comando e mais curto. */
```

```
    printf("De um temperatura em Fahrenheit: ");
    /* A leitura tambem mudou, veja detalhes abaixo. */
    scanf("%lf", &far);
    cel = fahrenheitParaCelsius(far);
    printf("A temperatura em Celsius e: %f\n", cel);
    printf("De um temperatura em Celsius: ");
    scanf("%lf", &cel);
    far = celsiusParaFahrenheit(cel);
    printf("A temperatura em Fahrenheit e: %f\n", far);

    /* Para nos esse retorno nao serve para nada, e um topico avancado. */
    return 0;
}
```

23.2 Detalhes de entrada e saída

Além do sumiço das classes, dos indicadores de acesso e dos termos `static` (pois todas as funções são assim) dos programas em C, uma outra mudança bastante visível é que os comandos para escrever na tela e para leitura do teclado mudam bastante. No lugar do `System.out.println` aparece o `printf` e no lugar dos métodos da classe `SavitchIn` usamos o `scanf`. A forma de usá-los também muda um pouco:

1. `printf`: imprime na tela. O primeiro parâmetro deve ser sempre uma string (texto entre aspas). Diferentemente de Java, não podemos usar a soma para concatenar cadeias. Como apresentar então variáveis no meio de uma string que será impressa pelo `printf`? Usamos nesse caso marcadores especiais para "deixar espaço"para imprimir o valor da variável e em seguida passamos estas variáveis como parâmetro. Vejamos um exemplo.

   ```
   printf("Esse e inteiro %d, esse double %f e o ultimo um char %c",
          umInteiro, umDouble, umChar);
   ```

   Caso desejemos que o `printf` pule de linha ao final, devemos adicionar um `\n` no final da string do `printf`:

   ```
   printf("Esse e inteiro %d, esse double %f e o ultimo um char %c\n",
          umInteiro, umDouble, umChar);
   ```

   Vale a pena consultar um manual de C para ver o `printf` em ação. Ele é um comando mais poderoso do que parece.

2. `scanf`: para ler valores do teclado usamos o `scanf`. Ele também recebe uma string com marcadores semelhantes do `printf` (a principal diferença é que para ler um double usamos `%lf` e não `%f`). Depois aparecem as variáveis que devem ser lidas antecedidas de um `&`. Por exemplo, se queremos ler um inteiro e um double fazemos:

   ```
   scanf("%d%lf", &umInteiro, &umDouble);
   ```

23.3 Declaração de variáveis

Em C, é comum que a declaração de todas as variáveis seja no início da função. Isso, porém, não é obrigatório (já foi, não é mais). Para declarar vetores em C, a sintaxe é mais simples do que em Java. Por exemplo, se queremos que a variável a seja um vetor de 100 inteiros basta:

```
int a[100];
```

A mesma coisa para matrizes:

```
int a[100][100];
```

23.4 Parâmetros de funções

Assim como em Java, os parâmetros que são tipos primitivos modificados dentro da função não se refletem fora dela. Já se alterarmos o conteúdo de um vetor, esta se reflete fora. Uma coisa interessante é que é possível pedir ao C que ele permita que a alteração de parâmetros que são de tipo primitivo reflita-se fora da função. Para isso deve-se anteceder o parâmetro de um asterisco em toda a função (inclusive na declaração do parâmetro). Ao chamar a função, a variável que queremos alterar deve estar precedida de um `&`.

```
#include <stdio.h>

/* Troca duas variaveis de lugar*/
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

int main()
{
  int c, d;
  c = 1;
  d = 2;
  swap(&c, &d);
  printf("c = %d, d = %d", c, d);
}
```

Outra mudança é que os colchetes de parâmetros que são vetores e matrizes devem vir após os seus nomes (e não antes como em Java). Além disso, se o parâmetro é uma matriz, você deve informar na declaração da função qual o número de linhas da matriz. Por exemplo

```
void umaFuncao(int umVetor[], int umaMatriz[100][])
```

23.5 Um último exemplo

Vejamos um último exemplo de programa em C. Queremos escrever um programa que lê uma seqüência de inteiros estritamente positivos terminada por zero e imprime a sua mediana.

```c
#include <stdio.h>

void selecaoDireta(int numeros[], int fim)
{
  int i, j, minimo, temp;
  for (i = 0; i < fim - 1; i = i + 1)
  {
    /* Inicialmente o menor elemento ja visto e o primeiro elemento. */
    minimo = i;
    for (j = i + 1; j < fim; j = j + 1)
    {
      if (numeros[j] < numeros[minimo])
    minimo = j;
    }
    /* Coloca o menor elemento no inicio do subvetor atual. */
    temp = numeros[i];
    numeros[i] = numeros[minimo];
    numeros[minimo] = temp;
  }
}

int main()
{
  /* Aceita no maximo 100 numeros no vetor. */
  int numeros[100];
  int i = -1;

  /* Le o vetor. */
  do
  {
    i++;
    scanf("%d", &numeros[i]);
  } while (numeros[i] > 0);

  /* Ordena para encontrar a mediana. */
  selecaoDireta(numeros, i);

  /* Agora ficou facil. */
  printf("A mediana e: %d\n", numeros[(i-1)/2]);

  /* O tal return inutil. */
  return 0;
}
```

Exercícios

1. Escreva um programa em C que lê uma sequência de números do teclado. Para cada valor x lido, o programa deve imprimir o valor de x^2 e $x!$. Se x for igual a 0 (zero), o programa termina. Além da função `main`, seu programa deverá conter as funções `calculaQuadrado` e `calculaFatorial`.

2. Escreva um programa em C que lê um inteiro do teclado e verifica se este número é palíndromo. Um

número palíndromo é um número que lido de trás para frente é o mesmo quando lido normalmente. Por exemplo, os números 11111, 64546 e 1001 são palíndromos, mas 1232 não é.

3. Escreva um programa em C que lê um número inteiro decimal do teclado e imprime a representação do número na base binária.

4. Escreva um programa em C que lê um número real x do teclado e calcula os valores do seno e coseno de x. Para tal, utilize as séries de Taylor descritas no Capítulo 10.

5. Escreva um programa em C que lê uma seqüencia de números inteiros terminada por 0 (zero) do teclado e imprime estes valores ordenados. Para este exercício, você pode utilizar a função de ordenação fornecida no exemplo da Seção 23.5.

6. Escreva uma função `metodoDaBolha` que recebe um vetor e o ordena utilizando o método da bolha explicado no Capítulo 17. Em seguida, modifique o programa do Exercício 5 para utilizar esta nova função de ordenação. Para finalizar, escreva uma função que testa automaticamente sua função `metodoDaBolha`.

Resoluções

- Exercício 1

```c
#include <stdio.h>

int calculaQuadrado(int numero) {
  int quadrado;
  quadrado = numero*numero;
  return quadrado;
}

int calculaFatorial(int numero) {
  int fatorial = 1;
  while (numero > 1) {
    fatorial = fatorial * numero;
    numero = numero - 1;
  }
  return fatorial;
}

int main() {
  int numero;     /* valor lido do teclado          */
  int quadrado;   /* armazena o quadrado de numero  */
  int fatorial;   /* armazena o fatorial de numero  */

  printf("Digite um inteiro nao nulo, ou 0 (zero) para sair: ");
  scanf("%d", &numero);

  while (numero != 0) {
    quadrado = calculaQuadrado(numero);
    fatorial = calculaFatorial(numero);
    printf("O quadrado do numero %d e' %d\n", numero, quadrado);
    printf("O fatorial do numero %d e' %d\n", numero, fatorial);
```

```
    printf("Digite um inteiro nao nulo, ou 0 (zero) para sair: ");
    scanf("%d", &numero);
  }

  return 0;
}
```

- Exercício 2

```
#include <stdio.h>

int calculaReverso (int numero) {

  int reverso = 0;
  while (numero != 0) {
    /* acrescenta mais um digito a direita de 'reverso' */
    reverso = reverso * 10 + numero % 10;
    /* remove esse digito de 'numero' */
    numero = numero / 10;
  }
  return reverso;
}

int main() {
  int numero;   /* numero lido do teclado      */
  int reverso;  /* armazena o numero invertido */

  printf("Digite um número inteiro positivo: ");
  scanf("%d", &numero);

  reverso = calculaReverso(numero);

  /* Compara o reverso com o numero original */
  if (reverso == numero)
    printf("O numero %d e palindrome.\n", numero);
  else
    printf("O numero %d nao e palindrome.\n", numero);

  return 0;
}
```

- Exercício 3

```
#include <stdio.h>

int decimalParaBinario(int numero)
{
  int digito;
  int binario = 0;
  int potencia = 1;

  while (numero > 0) {
```

```
    /* extrai proximo digito binario menos significativo (mais a direita) */
    digito = numero % 2;
    /* remove esse digito de numero */
    numero = numero / 2;
    /* adiciona o digito como o mais significativo ate o momento */
    binario = binario + digito * potencia;
    potencia = potencia * 10;
  }
  return binario;
}

int main() {
  int numero;  /* valor na base decimal lido do teclado */
  int binario; /* valor convertido para a base binaria  */

  printf("Digite um numero decimal inteiro positivo: ");
  scanf("%d", &numero);

  binario = decimalParaBinario(numero);

  printf("A represtacao binaria do numero decimal %d e %d\n", numero, binario);

  return 0;
}
```

Apêndice A

Utilizando o Dr. Java

Neste apêndice introduzimos a ferramenta `DrJava`, um ambiente de desenvolvimento para a linguagem `Java`. Por ser ele mesmo escrito em `Java`, pode ser usado em diversos ambientes, como, por exemplo, `Linux`, `Windows` e `Mac OS`. Um ambiente de desenvolvimento (também conhecido por IDE, de *Integrated Development Environment*) é um conjunto de ferramentas integradas que auxiliam a construção de programas.

O `DrJava` se encontra em desenvolvimento e, por isso, alguns recursos desejáveis ainda não estão disponíveis. Entretanto, os recursos mais simples que ele fornece já são apropriados para os objetivos deste livro. Planejamos então utilizar o `DrJava` para escrevermos nossos programas `Java`.

Objetivos

Neste apêndice forneceremos uma breve introdução ao uso do `DrJava`. O conteúdo é limitado e específico, suficiente para que você posteriormente seja capaz de conhecer melhor esta ferramenta por conta própria. Recomendamos para isso a consulta de manuais e outros documentos, não necessariamente sobre `DrJava` apenas. As descobertas pelo próprio uso e através de dicas de colegas também são incentivadas.

Neste apêndice você aprenderá a utilizar o `DrJava` para:

- escrever, compilar, manipular e depurar classes/objetos simples;
- gravar e reutilizar os arquivos que descrevem seus objetos.

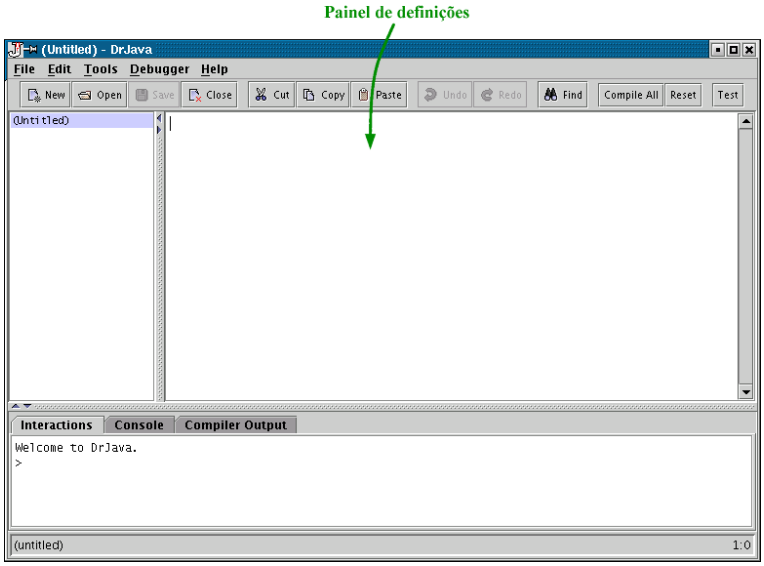
O `DrJava` pode ser obtido no endereço `http://drjava.org/`. Neste sítio é possível ainda encontrar documentação detalhada sobre como instalar e utilizar o `DrJava`.

A.1 Conversor de Temperatura simples

Como exemplo inicial, vamos construir a classe `Conversor` descrita no Capítulo 3. Lembrando, cada objeto dessa classe converte apenas a temperatura de 40 graus Celsius para a correspondente em graus Fahrenheit. Ao receber a mensagem `celsiusParaFahrenheit`, a classe `Conversor` devolve a temperatura em Fahrenheit equivalente a 40 graus Celsius.

Editando o código-fonte num arquivo

Vejamos o que o `DrJava` nos oferece para criarmos a classe. Ao iniciar o ambiente `DrJava`, abre-se uma janela parecida com a seguinte.



O painel de definições, indicado na figura acima, é um editor de textos. É nele que digitaremos o código `Java` que define a classe `Conversor`. Ele se parece muito com um editor de textos comum, exceto que possui alguns recursos para facilitar a digitação de código. Em particular, o comportamento das teclas <Enter> e <Tab> favorece a indentação do código. Outro recurso útil é a coloração e o destaque do texto.

Precisamos criar um arquivo novo para conter o código da nossa classe. Para isso, bastaria escolher a opção **New** do menu **File**. Porém, quando abrimos o `DrJava`, um novo arquivo sem nome já foi criado, então podemos usá-lo nesse momento (em vez de criar um novo). Sendo assim, digite o seguinte código no painel de definições.

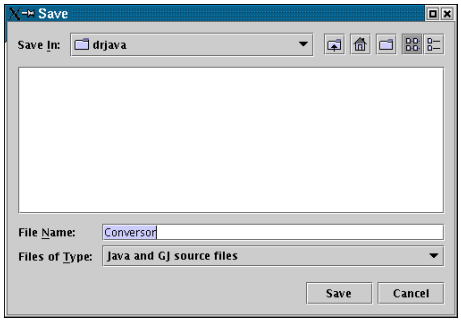
```
class Conversor
{
  int celsiusParaFahrenheit ()
  {
    return 9 * 40 / 5 + 32;
  }
}
```

Ao digitar, note os recursos mencionados que ajudam a programação (principalmente o comportamento das teclas <Enter> e <Tab>).

Gravando e reutilizando o arquivo

Vamos agora gravar o arquivo no disco. Escolha o menu **File** (clicando com o *mouse* ou usando as teclas de atalho, que nesse caso é a combinação <Alt>+<F>) e escolha o ítem **Save**. Como o arquivo ainda não tem

nome, estaremos na verdade acionando a opção **Save as...**. Por isso, surgirá um *diálogo* para definirmos a localização e o nome do arquivo, como mostra a seguinte figura.



Podemos determinar a localização, manipulando a *combo box* com rótulo **Save In:** (no topo do diálogo) e escolhendo um diretório na caixa abaixo dela, ou podemos deixar a localização como está. Desse modo, provavelmente o arquivo será gravado no diretório de onde o `DrJava` foi chamado.

Também precisamos escolher o nome do arquivo. Em algumas ocasiões, este deve ser *obrigatoriamente* idêntico ao nome da classe, mais o sufixo .java. Não é o nosso caso, mas mesmo assim vamos chamar o arquivo de Conversor.java. Como já digitamos o código da classe, o `DrJava` preencheu o nome do arquivo no *input field* de rótulo **File Name:** com o nome da classe. Note também que ele não acrescentou o sufixo .java no nome, o que será feito implicitamente quando finalizarmos (mas não há problema em digitar o sufixo mesmo assim).

Para confirmar a gravação, basta clicar no botão **Save**. As modificações futuras podem ser gravadas com o comando **Save**, sem precisar escolher o nome do arquivo novamente.

Com os arquivos das classes gravados em disco, podemos querer reutilizá-los no futuro. No `DrJava`, basta escolhermos a opção **Open** do menu **File** e um diálogo permitirá que você escolha o arquivo que deseja abrir novamente (é semelhante ao processo do **Save as...**).

Compilando a classe

Acabamos de definir a nossa classe, precisamos agora compilar para que possamos usá-la. No menu **Tools**, temos duas opções para fazer isso, **Compile All Documents** e **Compile Current Document**. Como só temos um documento aberto (Conversor.java), qualquer uma das opções serve. Escolha então **Compile Current Document**.

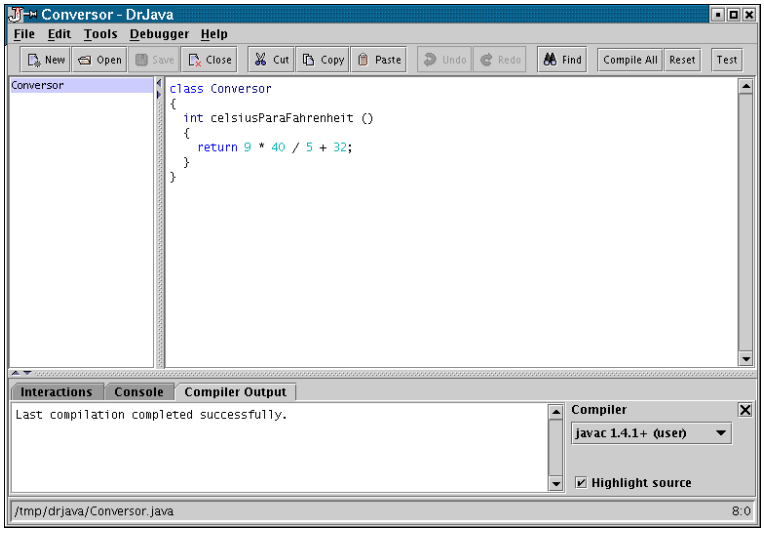
Com isso compilaremos a nossa classe. Note que, no painel inferior da janela, a *guia* **Compiler Output** se abre mostrando algumas mensagens. Se tudo der certo, a nossa janela se parecerá com a seguinte.

Usando a classe

Podemos finalmente usar a guia **Interactions**, que chamaremos de *janela do interpretador*, para criar e usar objetos da classe `Conversor`. Essa janela recebe comandos num *prompt* e a sintaxe desses comandos é muito parecida com a da linguagem `Java`.

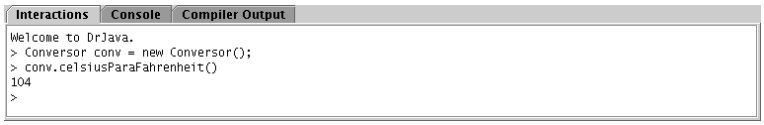
Clique então em **Interactions** e digite o seguinte.

```
Conversor conv = new Conversor();
conv.celsiusParaFahrenheit()
```



A ausência de `;` no final da linha do comando faz com que o interpretador imprima o valor devolvido pelo comando.

A janela do interpretador deverá se parecer com a figura abaixo.



A.2 Tratando erros

Utilizaremos agora a classe `Conversor4` descrita no Capítulo 3 para mostrarmos alguns outros recursos do `DrJava`. Para quem não lembra, objetos dessa classe convertem temperaturas entre graus Celsius e Fahrenheit fornecendo os seguintes métodos.

* `double celsiusParaFahrenheit (double c)`: recebe uma temperatura `c` em graus celsius e devolve a temperatura correspondente em graus fahrenheit. * `double fahrenheitParaCelsius (double f)`: recebe uma temperatura `f` em graus fahrenheit e devolve a temperatura correspondente em graus celsius.

Neste apêndice, porém, usaremos um código um pouco diferente em relação ao do Capítulo 3.

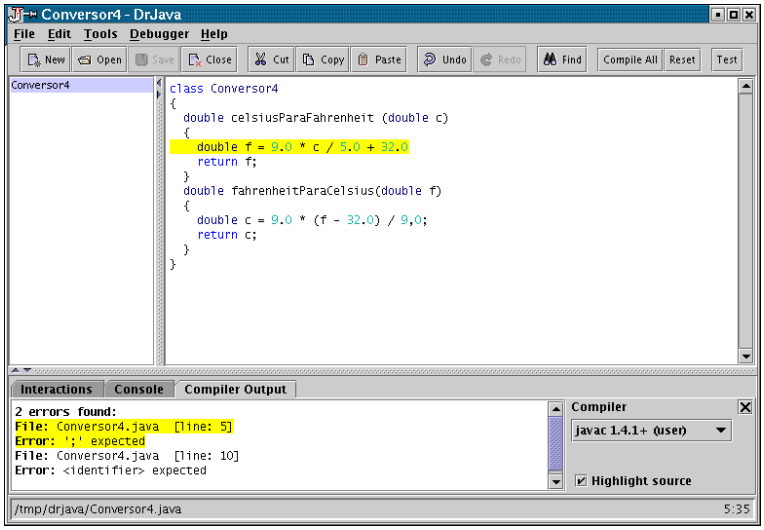
Erros no código

Feche o arquivo Conversor.java usando a opção **Close** do menu **File** e crie um novo arquivo (na verdade, como havia apenas um arquivo aberto, um novo é criado automaticamente).

Digite nesse arquivo o código abaixo (há erros intencionais nele). Você pode também usar o recurso de *copiar* e *colar* (*copy and paste*).

```
class Conversor4
{
  double celsiusParaFahrenheit (double c)
  {
    double f = 9.0 * c / 5.0 + 32.0
    return f;
  }
  double fahrenheitParaCelsius(double f)
  {
    double c = 9.0 * (f - 32.0) / 9,0;
    return c;
  }
}
```

Grave o arquivo e compile. Como o código contém erros, o compilador não terá sucesso e imprimirá algumas mensagens. Algo como mostra a figura abaixo.



O primeiro erro pode ser eliminado acrescentando-se um `;` no final da linha 5. Veja que o próprio compilador sugere isso.

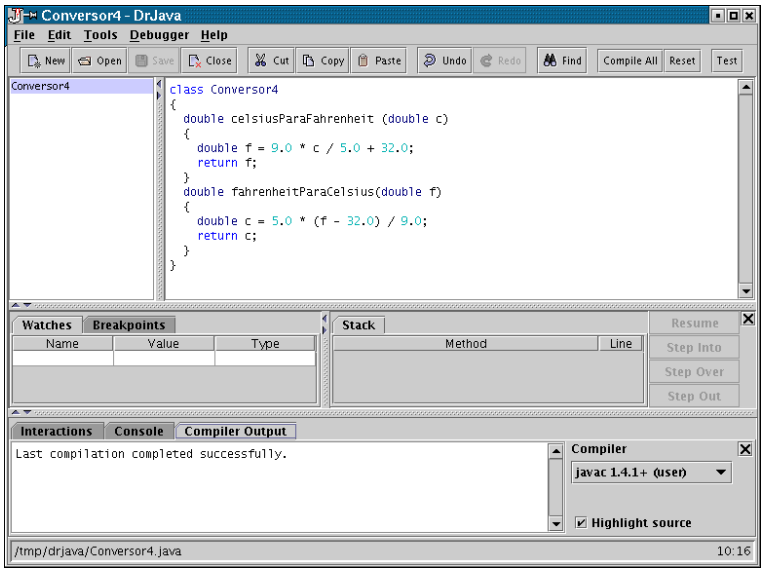
Já a descrição do segundo erro pode ser um pouco confusa. É importante saber que o compilador é capaz de encontrar erros no código, mas nem sempre pode determinar a causa exata. Nesses casos, as mensagens de erro apenas dão pistas para descobrirmos o que está errado (algumas vezes pistas falsas). O segundo erro é uma vírgula no lugar de um ponto, no final da linha 10.

Há outros tipos de erro os quais o compilador não tem condições de detectar. Um exemplo é o erro na fórmula de conversão da linha 10. Há um `9.0` onde deveria estar um `5.0`. No Capítulo 4 apresentamos uma maneira de detectarmos tais erros através de testes. Mas depois de detectarmos, precisamos descobrir a causa deles. Veremos a seguir uma ferramenta útil para essa tarefa. Mas antes corrija os erros e compile.

Depurador

Depurador (*debugger*) é uma ferramenta que nos ajuda a corrigir erros (*bugs*) de programas. O depurador do `DrJava` oferece apenas recursos básicos: pontos de parada (*breakpoints*), execução passo a passo (*step*) e inspeção simples de variáveis (*watch*). Mesmo assim, a classe `Conversor4` não é complexa o suficiente para justificar a aplicação do depurador, vamos utilizá-la apenas para demonstrar rapidamente cada um desses recursos.

Para ativar o depurador, escolha a opção **Debug Mode** do menu **Debugger**. Ao fazer isso, o painel de depuração é exibido na janela principal.



Começaremos selecionando um ponto de parada no código do `Conversor4`. Isso é feito no painel de definições, posicionando o cursor na linha desejada e escolhendo a opção **Toggle Breakpoint on Current Line** do menu **Debugger**.

O ponto de parada faz com que a execução do programa seja temporariamente interrompida exatamente antes da linha ser executada. A partir daí, para continuar a execução, deve-se usar os comandos de "passo a passo"(*step into*, *step over*, *step out*, *resume*). O que cada comando faz pode ser encontrado na seção *Documentation* da página do `DrJava` <http://drjava.sourceforge.net>

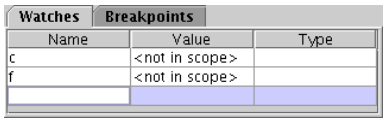
Posicione então o cursor do painel de definições na linha 10 do `Conversor4` e insira um ponto de parada (a linha ficará com um destaque vermelho). Para que o ponto de parada seja usado, temos que fazer com que o método seja executado. Para isso digite o seguinte na janela do interpretador.

```
Conversor4 c4 = new Conversor4();
c4.fahrenheitParaCelsius(42)
```

A execução do método será interrompida antes da linha 10 ser executada. O destaque azul da linha indica isso. Para continuar a execução passo a passo (linha a linha), execute o comando *step over* algumas vezes (pressionando a tecla <F11>). Dessa forma, é possível constatar quais trechos de código são usados numa execução em particular.

Um outro recurso bastante útil, que deve ser usado em conjunto com esses que acabamos ver, é a inspeção de valores de variáveis (*watch*). Esse recurso nos informa o valor de certas variáveis durante a execução passo a passo. Para isso, é necessário preencher a coluna *Name* da tabela da guia *Watches* com os nomes das variáveis que se deseja inspecionar (uma variável em cada linha da tabela). Basta clicar numa célula da coluna *Name*, digitar o nome da variável e pressionar <Enter>.

Faça este procedimento para as variáveis `c` e `f`. Teremos algo como a próxima figura.



Repita a execução passo a passo descrita anteriormente e observe o que ocorre com a tabela.

Dica: o interpretador armazena os últimos comandos digitados na janela. Para acessá-los, pressione as setas para cima e para baixo do teclado. Esse recurso se chama *History* e possui algumas opções úteis no menu **Tools**.

Apêndice B

Desenvolvimento Dirigido por Testes

Neste apêndice, veremos um exemplo de como podemos desenvolver um programa ao mesmo tempo em que desenvolvemos seus testes. Ou melhor, a idéia é até um pouco mais radical, criar os testes e ir adaptando o programa até que o mesmo satisfaça a todos os testes criados. Esta abordagem é chamada de Desenvolvimento Dirigido por Testes (*Test-Driven Development*) e a prática de escrever os testes antes do código a ser testado é chamada de Programação com Testes a Priori (*Test-First Programming*). Para isto, devemos nos guiar pelo seguinte ciclo, de forma incremental:

1. Escreva teste(s);
2. Escreva código para que os testes funcionem;
3. Observe o código buscando formas de simplificá-lo [1]
4. Enquanto o programa não acabou, volte ao primeiro passo.

O exemplo escolhido [2] é o de um conversor de números inteiros para algarismos romanos.

B.1 O Exemplo

Vamos começar do básico. Inicialmente, queremos criar um teste que, nada mais natural, para o número 0 tenhamos como resultado um `String` vazio. Para isto usaremos uma palavra chave que existe a partir de Java 1.4: `assert`. Em linhas gerais, `assert` recebe um único parâmetro booleano e, se ele for verdadeiro, nada acontece; no entanto, se ele for falso, o `assert` gera uma mensagem de erro e interrompe a execução do programa. É interessante notar que é necessário "ativar" as verificações do `assert` usando o parâmetro `-ea` da máquina virtual Java, ou seja o seguinte programa deve ser executado com: `java -ea TesteConversorNúmerosRomanos`.

Usando o `assert`, o teste fica:

```
class TesteConversorNúmerosRomanos
{
```

[1] Na verdade existe toda uma teoria para isto, que se chama refatoração.

[2] A idéia original para este exemplo foi encontrada em `http://www.differentpla.net/node/58`, mas o exemplo a seguir foi criado de forma totalmente independente.

```
  public void testes()
  {
    Conversor c = new Conversor();

    System.out.println("Início dos Testes");
    assert (c.converte(0).compareTo("")==0);
    System.out.println("Fim dos Testes");
  }
  public static void main(String []args)
  {
    TesteConversorNúmerosRomanos t = new TesteConversorNúmerosRomanos();
    t.testes();
  }
}
```

Como seria de se esperar, o trecho isolado acima nem mesmo compila, pois a classe `Conversor` ainda não existe. Mas, podemos resolver isto criando a seguinte classe:

```
class Conversor
{
  public String converte(int x)
  {
    return "";
  }
}
```

Agora, não só o programa compila, como é executado sem erros. Logo, podemos aprimorar o nosso teste com a verificação da conversão do inteiro 1.

```
class TesteConversorNúmerosRomanos
{
  public void testes()
  {
    Conversor c = new Conversor();

    System.out.println("Início dos Testes");
    assert (c.converte(0).compareTo("")==0);
    assert (c.converte(1).compareTo("I")==0);
    System.out.println("Fim dos Testes");
  }
  public static void main(String []args)
  {
    TesteConversorNúmerosRomanos t = new TesteConversorNúmerosRomanos();
    t.testes();
  }
}
```

Agora, ao executar este teste, recebemos a seguinte mensagem de erro:

```
AssertionError:
 at TesteConversorNúmerosRomanos.testes(TesteConversorNúmerosRomanos.java:9)
 at TesteConversorNúmerosRomanos.main(TesteConversorNúmerosRomanos.java:15)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

```
    at java.lang.reflect.Method.invoke(Method.java:585)
```

Esta mensagem indica que a segunda verificação falhou. O nosso próximo passo é generalizar o programa para que ele funcione para o teste atual:

```
class Conversor
{
 public String converte(int x)
 {
   if (x == 0)
      return "";
   return "I";
 }
}
```

Desta forma, o `Conversor` "voltou" a funcionar. Está na hora de aumentarmos a nossa classe de testes com mais uma verificação:

```
    assert (c.converte(2).compareTo("II")==0);
```

Como novamente o `Conversor` falhou, vamos corrigí-lo:

```
class Conversor
{
  public String converte(int x)
  {
    if (x == 0)
      return "";
    if (x == 1)
      return "I";
    return "II";
  }
}
```

Mas, quando acrescentarmos o teste para o número 3, o nosso conversor vai falhar novamente. Continuando o desenvolvimento teremos:

```
class Conversor
{
  public String converte(int x)
  {
    if (x == 0)
      return "";
    if (x == 1)
      return "I";
    if (x == 2)
      return "II";
    return "III";
  }
}
```

Olhando o código acima, vemos um padrão de repetição no uso de `if`s e sempre que aparecem estas repetições é interessante encontrar formas de reduzí-las. Podemos simplificar o exemplo acima usando um laço simples:

```
class Conversor
{
  public String converte(int x)
  {
    String s = "";

    while (x > 0)
    {
      x--;
      s = s + "I";
    }
    return s;
  }
}
```

O novo código continua passando pelo teste, mas ainda temos que ver o funcionamento para mais números, no caso o 4 e o 5.

```
class TesteConversorNúmerosRomanos
{
  public void testes()
  {
    Conversor c = new Conversor();

    System.out.println("Início dos Testes");
    assert (c.converte(0).compareTo("")==0);
    assert (c.converte(1).compareTo("I")==0);
    assert (c.converte(2).compareTo("II")==0);
    assert (c.converte(3).compareTo("III")==0);
    assert (c.converte(4).compareTo("IV")==0);
    assert (c.converte(5).compareTo("V")==0);
    System.out.println("Fim dos Testes");
  }
  public static void main(String []args)
  {
    TesteConversorNúmerosRomanos t = new TesteConversorNúmerosRomanos();
    t.testes();
  }
}

class Conversor
{
  public String converte(int x)
  {
    String s = "";

    if (x == 4)
      return "IV";
    if (x == 5)
      return "V";
    while (x > 0)
    {
      x--;
      s = s + "I";
```

```
    }
    return s;
  }
}
```

O nosso conversor já funciona para os inteiros de 0 a 5. Pensando um pouco, podemos ver que os números 6, 7 e 8 tem construção análoga ao 1, 2 e 3, e o `while` no final pode nos ajudar. Logo, para a seguinte série de `asserts`:

```
    assert (c.converte(0).compareTo("")==0);
    assert (c.converte(1).compareTo("I")==0);
    assert (c.converte(2).compareTo("II")==0);
    assert (c.converte(3).compareTo("III")==0);
    assert (c.converte(4).compareTo("IV")==0);
    assert (c.converte(5).compareTo("V")==0);
    assert (c.converte(6).compareTo("VI")==0);
    assert (c.converte(7).compareTo("VII")==0);
    assert (c.converte(8).compareTo("VIII")==0);
```

O conversor seguinte funciona:

```
class Conversor
{
  public String converte(int x)
  {
    String s = "";

    if (x == 4)
      return "IV";
    if (x >= 5)
    {
      s = s + "V";
      x = x - 5;
    }
    while (x > 0)
    {
      x--;
      s = s + "I";
    }
    return s;
  }
}
```

Já que o programa funciona vamos aumentar os testes para os números 9 e 10. Para sanar a nova falha, vamos adicionar dois novos `ifs`:

```
class Conversor
{
  public String converte(int x)
  {
    String s = "";

    if (x == 4)
      return "IV";
```

```
    if (x == 9)
      return "IX";
    if (x == 10)
      return "X";
    if (x >= 5)
    {
      s = s + "V";
      x = x - 5;
    }
    while (x > 0)
    {
      x--;
      s = s + "I";
    }
    return s;
  }
}
```

Novamente, aparece uma padrão de repetição, ou duplicação, para simplificar o código, apenas observando o padrão `IV` e `IX`:

```
class Conversor
{
  public String converte(int x)
  {
    String s = "";

    if ((x == 4) || (x == 9))
    {
      s = s + "I";
      x++;
    }
    if (x == 10)
      return s + "X";
    if (x >= 5)
    {
      s = s + "V";
      x = x - 5;
    }
    while (x > 0)
    {
      x--;
      s = s + "I";
    }
    return s;
  }
}
```

Além do que é bem fácil corrigir o código para que o mesmo funcione para os números 11, 12 e 13, basta trocar o `if (x == 10)` por:

```
    if (x >= 10)
    {
      s = s + "X";
```

```
      x = x - 10;
    }
```

Vamos agora, eliminar o código duplicado criando um novo método `aux`. Como este método modifica tanto a `String` como o valor de `x`, foi necessário usar um objeto do tipo `StringBuffer`

```
class Conversor
{
  private int aux(int x, int val, StringBuffer s, char ch)
  {
    if (x >= val)
    {
      s.append(ch);
      x = x - val;
    }
    return x;
  }

  public String converte(int x)
  {
    StringBuffer s = new StringBuffer("");

    if ((x == 4) || (x == 9))
    {
      s.append("I");
      x++;
    }
    x = aux(x, 10, s, 'X');
    x = aux(x,  5, s, 'V');
    while (x > 0)
    {
      x--;
      s.append("I");
    }
    return s.toString();
  }
}
```

Continuando com os testes, para os números 14 e 15, podemos verificar que basta repetir um trecho de código para que o programa funcione, logo vamos criar mais um método:

```
class Conversor
{
  private int aux(int x, int val, StringBuffer s, char ch)
  {
    if (x >= val)
    {
      s.append(ch);
      x = x - val;
    }
    return x;
  }

  private int aux2(int x, StringBuffer s)
```

```
    {
      s.append("I");
      return x + 1;
    }

    public String converte(int x)
    {
      StringBuffer s = new StringBuffer("");

      if ((x == 4) || (x == 9))
      {
        x = aux2(x, s);
      }
      x = aux(x, 10, s, 'X');
      if ((x == 4) || (x == 9))
      {
        x = aux2(x, s);
      }
      x = aux(x,  5, s, 'V');
      while (x > 0)
      {
        x--;
        s.append("I");
      }
      return s.toString();
    }
  }
```

É interessante notar que agora podemos adicionar os números até 18 que o teste funciona. Mas, para o 19 o teste falha novamente. Com um pouco de observação podemos ver que com pequenas alterações tudo volta a funcionar, basta colocar uma chamada adicional ao método `aux`. Mas, lembrando que teremos números como o 30, o melhor seria adicionar um outro laço ao programa:

```
  class Conversor
  {
    private int aux(int x, int val, StringBuffer s, char ch)
    {
      if (x >= val)
      {
        s.append(ch);
        x = x - val;
      }
      return x;
    }

    private int aux2(int x, StringBuffer s)
    {
      s.append("I");
      return x + 1;
    }

    public String converte(int x)
    {
      StringBuffer s = new StringBuffer("");
```

```
    while (x >= 9)
    {
      if (x == 9)
      {
        x = aux2(x, s);
      }
      x = aux(x, 10, s, 'X');
    }
     if (x == 4)
      {
        x = aux2(x, s);
      }
    x = aux(x,  5, s, 'V');
    while (x > 0)
    {
      x--;
      s.append("I");
    }
    return s.toString();
  }
}
```

Este novo conversor funciona até o número 39. Para que ele funcione para o número 40 temos que entrar com uma nova letra, o L que corresponde ao 50. Agora que já conhecemos melhor o mecanismo de conversão de números, podemos escolher alguns testes para adicionar, ao invés de colocar testes para todos os números. Para a nova série de testes:

```
...
   assert (c.converte(17).compareTo("XVII")==0);
   assert (c.converte(18).compareTo("XVIII")==0);
   assert (c.converte(19).compareTo("XIX")==0);
   assert (c.converte(20).compareTo("XX")==0);
   assert (c.converte(23).compareTo("XXIII")==0);
   assert (c.converte(34).compareTo("XXXIV")==0);
   assert (c.converte(39).compareTo("XXXIX")==0);
   assert (c.converte(46).compareTo("XLVI")==0);
   assert (c.converte(59).compareTo("LIX")==0);
   assert (c.converte(63).compareTo("LXIII")==0);
```

O seguinte conversor apenas adiciona a letra L:

```
class Conversor
{
  private int aux(int x, int val, StringBuffer s, char ch)
  {
    if (x >= val)
    {
      s.append(ch);
      x = x - val;
    }
    return x;
  }
```

```
  private int aux2(int x, StringBuffer s)
  {
    s.append("I");
    return x + 1;
  }

  public String converte(int x)
  {
    StringBuffer s = new StringBuffer("");


    if (x >= 40)
    {
      if (x < 50)
      {
        s.append("X");
        x = x + 10;
      }
      s.append("L");
      x = x - 50;
    }
    while (x >= 9)
    {
      if (x == 9)
      {
        x = aux2(x, s);
      }
      x = aux(x, 10, s, 'X');
    }
    if (x == 4)
    {
      x = aux2(x, s);
    }
    x = aux(x,  5, s, 'V');
    while (x > 0)
    {
      x--;
      s.append("I");
    }
    return s.toString();
  }
}
```

Observe que colocar o X antes do L é similar a colocar o I antes do X, ou do V, logo podemos mudar o nome do método `aux2` para `precede` e usá-lo também neste caso. Assim como o método `aux` pode ser chamado de `colocaLetra` e usado em outras partes:

```
class Conversor
{
 private int colocaLetra(int x, int val, StringBuffer s, char ch)
 {
   if (x >= val)
   {
     s.append(ch);
```

```
      x = x - val;
    }
    return x;
  }

  private int precede(int x, int val, StringBuffer s, char ch)
  {
    s.append(ch);
    return x + val;
  }
  public String converte(int x)
  {
    StringBuffer s = new StringBuffer("");


    if (x >= 40)
    {
      if (x < 50)
      {
        x = precede(x, 10, s, 'X');
      }
      x = colocaLetra(x, 50, s, 'L');
    }
    while (x >= 9)
    {
      if (x == 9)
      {
        x = precede(x, 1, s, 'I');
      }
      x = colocaLetra(x, 10, s, 'X');
    }
     if (x == 4)
      {
        x = precede(x, 1, s, 'I');
      }
    x = colocaLetra(x,  5, s, 'V');
    while (x > 0)
    {
      x = colocaLetra(x, 1, s, 'I');
    }
    return s.toString();
  }
}
```

Observando o código acima vemos que existe uma correspondência entre os valores e os números romanos I, V, X e L nas chamadas dos métodos, logo, podemos criar um método adicional que faz esta correspondência, simplificando as chamadas.

```
class Conversor
{
 private char corresponde(int i)
 {
   if (i == 1)
     return 'I';
```

```
    else if (i == 5)
      return 'V';
    else if (i == 10)
      return 'X';
    else
      return 'L';
  }

  private int colocaLetra(int x, int val, StringBuffer s)
  {
    if (x >= val)
    {
      s.append(corresponde(val));
      x = x - val;
    }
    return x;
  }

  private int precede(int x, int val, StringBuffer s)
  {
    s.append(corresponde(val));
    return x + val;
  }
  public String converte(int x)
  {
    StringBuffer s = new StringBuffer("");

    if (x >= 40)
    {
      if (x < 50)
      {
        x = precede(x, 10, s);
      }
      x = colocaLetra(x, 50, s);
    }
    while (x >= 9)
    {
      if (x == 9)
      {
        x = precede(x, 1, s);
      }
      x = colocaLetra(x, 10, s);
    }
    if (x == 4)
    {
      x = precede(x, 1, s);
    }
    x = colocaLetra(x,  5, s);
    while (x > 0)
    {
      x = colocaLetra(x, 1, s);
    }
    return s.toString();
  }
```

```
}
```

Olhando o código acima parece existir um padrão entre as chamadas aos métodos `precede` e `colocaLetra`, mas como ainda não está claro o que pode ser feito, vamos criar mais testes para que o conversor funcione com números maiores do que 89.

```
...
    assert (c.converte(90).compareTo("XC")==0);
    assert (c.converte(94).compareTo("XCIV")==0);
    assert (c.converte(99).compareTo("XCIX")==0);
    assert (c.converte(103).compareTo("CIII")==0);
    assert (c.converte(149).compareTo("CXLIX")==0);
    assert (c.converte(349).compareTo("CCCXLIX")==0);
```

Para isto temos que alterar os métodos `corresponde`, adicionando a nova letra (C), e o método `converte`, transcrito abaixo:

```
public String converte(int x)
 {
    StringBuffer s = new StringBuffer("");

    while (x >= 90)
    {
      if (x < 100)
      {
        x = precede(x, 10, s);
      }
      x = colocaLetra(x, 100, s);
    }
    if (x >= 40)
    {
      if (x < 50)
      {
        x = precede(x, 10, s);
      }
      x = colocaLetra(x, 50, s);
    }
    while (x >= 9)
    {
      if (x == 9)
      {
        x = precede(x, 1, s);
      }
      x = colocaLetra(x, 10, s);
    }
    if (x == 4)
    {
      x = precede(x, 1, s);
    }
    x = colocaLetra(x,  5, s);
    while (x > 0)
    {
      x = colocaLetra(x, 1, s);
    }
```

```
    return s.toString();
  }
}
```

Neste momento, o padrão de duplicação do código fica mais visível, e podemos buscar uma forma de eliminar as repetições. Logo, foram feitas diversas modificações no código até chegarmos a versão seguinte (como tínhamos os testes, a cada modificação, o correto funcionamento da nova versão pode ser verificado).

```
class Conversor
{
 private char corresponde(int i)
 {
   if (i == 1)
     return 'I';
   else if (i == 5)
     return 'V';
   else if (i == 10)
     return 'X';
   else if (i == 50)
     return 'L';
   else
     return 'C';
 }

 private int colocaLetra(int x, int val, StringBuffer s)
 {
   s.append(corresponde(val));
   x = x - val;
   return x;
 }

 private int precede(int x, int val, StringBuffer s)
 {
   s.append(corresponde(val));
   return x + val;
 }
 public String converte(int x)
 {
   StringBuffer s = new StringBuffer("");

   while (x >= 90)
   {
     if (x < 100)
     {
       x = precede(x, 10, s);
     }
     x = colocaLetra(x, 100, s);
   }
   while (x >= 40)
   {
     if (x < 50)
     {
       x = precede(x, 10, s);
     }
```

```
      x = colocaLetra(x, 50, s);
    }
    while (x >= 9)
    {
      if (x < 10)
      {
        x = precede(x, 1, s);
      }
      x = colocaLetra(x, 10, s);
    }
    while (x >= 4)
    {
      if (x < 5)
      {
        x = precede(x, 1, s);
      }
      x = colocaLetra(x, 5, s);
    }
    while (x > 0)
    {
      x = colocaLetra(x, 1, s);
    }
    return s.toString();
  }
}
```

Neste ponto, o padrão ficou bem claro (é interessante notar que mesmo que alguns `if`s tenham sido trocados por `while`s devido as características do programa o laço só é executado no máximo uma vez). Vale ressaltar mais uma vez que a garantia que o programa acima está correto vem da nossa classe de testes.

Com um pouco de paciência, dá para trocar os `while`s acima por apenas 3, temos abaixo o programa mais limpo:

```
class Conversor
{
 private int v[] = {100, 50, 10, 5, 1};
 private char corresp[] = {'C', 'L', 'X', 'V', 'I'};

 private int colocaLetra(int x, int i, StringBuffer s)
 {
   s.append(corresp[i]);
   x = x - v[i];
   return x;
 }

 private int precede(int x, int i, StringBuffer s)
 {
   s.append(corresp[i]);
   return x + v[i];
 }
 public String converte(int x)
 {
   StringBuffer s = new StringBuffer("");
   int i = 0;
```

```
    while (i < v.length - 1)
    {
      while (x >= v[i] - v[i+2])
      {
        if (x < v[i])
        {
          x = precede(x, i+2, s);
        }
        x = colocaLetra(x, i, s);
      }
      while (x >= v[i+1] - v[i+2])
      {
        if (x < v[i+1])
        {
          x = precede(x, i+2, s);
        }
        x = colocaLetra(x, i+1, s);
      }
      i = i + 2;
    }
    while (x > 0)
    {
      x = colocaLetra(x, v.length - 1, s);
    }
    return s.toString();
  }
}
```

É interessante notar que o programa acima ainda pode ser simplificado, mas já chegamos a um conversor que funciona até o número 399.

Os passos apresentados, foram na medida do possível, muito próximos a um desenvolvimento real. É claro que durante o desenvolvimento do conversor ocorreram diversos erros, inclusive de compilação, mas os mesmos foram omitidos pois não apresentam interesse didático.

Exercícios

1. Verifique, através da metodologia testes primeiro, que o programa acima é flexível, adicionando os números romanos D e M.

2. Usando a mesma metodologia de testes primeiro, crie um programa que converte números romanos em números decimais.

Bibliografia

- Philippe Breton, *História da Informática*, Editora Unesp, 1987

  Para aqueles que desejam se aprofundar na história da computação, este livro fornece uma visão técnica, econômica e social sobre a evolução do uso dos computadores em nossa sociedade. Por ter sido escrito no final da década de 1980, não cobre os últimos avanços da informática, como a linguagem Java e a Web.

- Samuel N. Kamin, M. Dennis Mickunas e Edward M. Reingold, *An Introduction to Computer Science Using Java*, McGraw-Hill, 1998

  Talvez o primeiro livro a propor o uso de Java para um curso introdutório à Ciência da Computação. Escrito por alguns dos principais especialistas em linguagens de programação da Universidade de Illinois em Urbana-Champaign.

- Walter Savitch, *Java, An Introduction to Computer Science & Programming*, second edition, Prentice Hall, 2001

  Um bom livro de introdução à computação escrito pelo prolífico Prof. Savitch da Universidade da California em San Diego. Daqui foi tirada a classe para entrada de dados sugerida neste livro para versões de Java anteriores à 1.5.

- Bruce Ecklel, *Thinking in Java*, fourth edition, Prentice Hall, 2006.

  Excelente livro para aqueles que desejam se aprofundar na linguagem Java. Sua principal vantagem é que ele explica com bastante detalhe o funcionamento da linguagem. Além disso, o livro fornece diversas dicas sobre estilos de programação. O texto completo da terceira edição do livro encontra-se disponível para ser baixado livremente em `http://mindview.net/Books`.

- Harvey Deitel e Paul Deitel, *Java: Como Programar*, sexta edição, Prentice Hall, 2005.

  Livro em português para aqueles que desejam obter mais detalhes sobre como programar em Java. Possui uma abordagem bastante acessível e baseada em exemplos, além de cobrir uma grande quantidade de tópicos.

- Jaime Niño e Frederick A. Hosch *A Introduction to Programming and Object Oriented Design*

  Um bom livro de introdução à Ciência da Computação. Possui algumas semelhanças com nosso livro como o enfoque em orientação a objetos e ênfase em testes e mesmo o uso do DrJava e seu painel interativo. O livro possui vários exemplos interessantes e é bastante completo, usando suas 900 páginas

para cobrir com profundidade vários tópicos que apenas arranhamos ou nem citamos, como por exemplo herança e polimorfismo, exceções, interfaces gráficas e iteradores.

- *The Java Tutorial*, disponível online em `http://java.sun.com/docs/books/tutorial/`

  Referência online que introduz os principais conceitos de Java. Apesar de ter um conteúdo bastante básico, este livro possui a vantagem de poder ser facilmente acessado através da internet.