
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

Fabio Kon
Alfredo Goldman
Paulo J. Silva e Silva

Editado e Revisado por:
Raphael Y. de Camargo

São Paulo, 12 de janeiro de 2005

Sugestões e Correções

Se você encontrar erros ou tiver sugestões para melhorias neste livro, favor enviar para o email *kon@ime.usp.br*.

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. A aula dirigida do Dr. Java foi preparada 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, têm nos dado inúmeras sugestões úteis.

Agradecemos ao Prof. Walter Savitch por ter autorizado o uso de sua classe para entrada de dados.

Finalmente, agradecemos aos alunos de MAC 110 e aos professores que não foram citados mas que deram sugestões e nos incentivaram a escrever este livro.

Sumário

Prefácio	xi
1 Teatro de Objetos	1
1.1 Introdução	1
1.2 Disputa de Pênaltis	1
2 História da Computação	7
2.1 História da Computação e Arquitetura do Computador	7
2.2 E a evolução do software?	12
3 Conversor de Temperaturas	15
3.1 Analogia entre dramatização da disputa de pênaltis e Programação Orientada a Objetos	15
3.2 Um exemplo real em Java: um conversor de Celsius para Fahrenheit	16
3.3 Exercícios:	18
4 Testes Automatizados	19
4.1 Testes	19
4.2 Exercícios	22
4.3 Resoluções	23
5 Métodos com vários parâmetros	25
5.1 Métodos com vários parâmetros	25
5.2 Exercícios	28
6 if else encaixados	31
6.1 if else encaixados	31
6.2 Exercícios	33
7 Programas com Vários Objetos	37
7.1 Programas com Vários Objetos	37
7.2 Exercício	39
8 Laços e Repetições	41
8.1 Laços em linguagens de programação	41
8.2 O Laço while	42

8.3	Números primos	44
8.4	Exercícios	45
9	Expressões e Variáveis Lógicas	49
9.1	Condições como expressões	49
9.2	Precedência	52
9.3	Exemplos	52
9.4	Exercício:	54
10	Mergulhando no “while”	57
10.1	Um pouco mais sobre primos.	57
10.2	Uma biblioteca de funções matemáticas.	59
10.3	do...while	60
10.4	Exercícios	60
11	Caracteres e cadeias de caracteres	63
11.1	Um tipo para representar caracteres	63
11.2	Cadeias de Caracteres (Strings)	65
11.3	Exercícios:	66
12	Manipulando números utilizando diferentes bases	67
12.1	Exercícios	68
13	Arrays (vetores)	71
13.1	Arrays (vetores)	71
13.1.1	O atributo <code>length</code>	73
13.1.2	Inicialização de <i>arrays</i>	73
13.2	Criação de Programas Java	73
13.3	Exercícios	75
14	for, leitura do teclado e conversão de Strings	77
14.1	O comando <code>for</code>	77
14.2	Leitura do Teclado	78
14.3	Conversão de String para números	79
14.4	Exercícios	79
15	Laços Encaixados e Matrizes	81
15.1	Laços encaixados	81
15.2	Matrizes (arrays multi-dimensionais)	82
15.3	Exemplo: LIFE, o jogo da vida	83
15.4	Exercícios	85

16 Busca e ordenação	89
16.1 Busca	89
16.2 Pondo ordem na casa	90
16.3 Exercícios	92
17 Busca binária, fusão e o que são as variáveis	93
17.1 Busca binária	93
17.2 Fusão	94
17.3 O que guardam as variáveis?	95
17.4 Exercícios	96
18 Construtores e especificadores de acesso	97
18.1 Construtores	97
18.2 Especificadores de acesso	100
18.3 Exercícios	101
19 Interfaces	103
19.1 O Conceito de Interfaces	103
19.2 Um primeiro exemplo	103
19.3 Implementando mais de uma interface por vez	107
19.4 Um exemplo mais sofisticado	108
19.5 A Importância de Interfaces	112
19.6 Exercícios:	113
20 Herança	117
20.1 O Conceito de Herança	117
20.2 Terminologia de herança	117
20.3 Implementação de herança na linguagem Java	118
20.4 Hierarquia de Classes	120
20.5 Relacionamento “é um”	120
20.6 Resumo	121
20.7 Exercícios	121
21 Javadoc	123
21.1 Javadoc	123
22 O C que há em Java	129
22.1 O C que há em Java	129
22.2 Detalhes de entrada e saída	131
22.3 Declaração de variáveis	131
22.4 Parâmetros de funções	132
22.5 Um último exemplo	132

A	Utilizando o Dr. Java	135
A.1	Introdução	135
A.1.1	Como obter, instalar e executar o DrJava	135
A.2	Conversor de Temperatura simples	136
A.3	Tratando erros	138

Lista de Figuras

2.1	Arquitetura do ENIAC	10
2.2	Arquitetura de Von Neumann	10
20.1	Diagrama de herança	118
20.2	Hierarquia de classes representando os seres vivos	120
20.3	Hierarquia errada	121
21.1	Documentação gerada pelo Javadoc	127

Prefácio

Caros alunos e alunas de MAC-110,

Sejam bem-vindos aos maravilhoso mundo da Ciência da Computação. Neste primeiro semestre do seu curso, dentre as inúmeras novidades que vocês encontrarão, estará a disciplina de Introdução à Ciência da Computação onde vocês irão aprender os elementos básicos da programação de computadores e terão contato com alguns dos conceitos fundamentais da Ciência da Computação. Pela terceira vez no IME/USP, em 2005 a disciplina será baseada numa das mais modernas e sofisticadas linguagens de programação: Java. Pela segunda vez, vamos utilizar esta apostila e esperamos que ela lhes ajude na fascinante empreitada de aprender a programar.

Esperamos melhorar a apostila sempre que possível. Portanto, se você achar algum erro ou tiver alguma sugestão para melhorá-la, não deixe de nos escrever.

Abraços e um bom 2005!

São Paulo, janeiro de 2005.

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

Capítulo 1

Teatro de Objetos

1.1 Introdução

O Teatro de Objetos é uma atividade realizada com os alunos de Introdução à Ciência da Computação durante a primeira aula da disciplina. O objetivo é fazer com que os alunos vivenciem um jogo interativo do qual participem vários "objetos" realizando diferentes formas de ações e comunicações. Os conceitos de orientação a objetos empregados no teatro não são explicitamente explicados já na primeira aula mas serão abordados ao longo da disciplina. A primeira metade da primeira aula é dedicada a uma conversa informal com os alunos explicando quais são os objetivos desta disciplina (e do curso inteiro, se for o caso). É também interessante fazer perguntas sobre contatos prévios que os alunos tiveram com informática e com programação. É bom deixar claro que a disciplina pode ser acompanhada por uma pessoa que nunca viu um computador na frente em sua vida, mas que não é uma disciplina fácil, é preciso se empenhar. Na segunda metade da aula, dizemos aos alunos que vamos exercitar as suas habilidades dramáticas: para ilustrar o funcionamento de um programa de computador complexo, nós vamos fazer de conta que somos partes de um programa de computador trabalhando em conjunto para atingir um certo objetivo. Podemos fazer um certo suspense sobre qual é o objetivo (simular uma disputa de pênaltis) e sobre como ele será alcançado.

1.2 Disputa de Pênaltis

A "peça" encenada em 2003 representará uma disputa de pênaltis entre dois times e contará com a participação de cerca de 26 atores desempenhando 8 papéis. 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 será responsável por escolher os alunos que desempenharão cada papel. Se houver limitação de espaço, é conveniente que os 8 torcedores fiquem em suas próprias carteiras para não tumultuar muito o ambiente. Obviamente, o professor 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 professor escolher alunos que pareçam bem barulhentos e faladores (por exemplo, a turma do fundão). Ao escolher os atores, o professor 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 o valor dos atributos do personagem devem ser escritos na hora à lápis pelo professor. Alguns papéis, como o 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 à caneta como valendo 1.

Além deste 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 professor não deve se 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 60 minutos.

Eis uma descrição detalhada dos dados que deverão aparecer nos cartões descritivos e no script (comportamento) de cada um dos 26 atores participantes da encenação.

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 desapeço 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 bater e para o goleiro defensor a cada nova batida. Quando os personagens estiverem apostos, 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 repetida 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-o 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 nesta aula?

- a história da computação
- evolução da arquitetura do computador
- 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 (calculadora) digital,
 - o capaz de somar
 - o entrada através de discos giratórios
 - o ajudou seu pai, coletor de impostos
- Leibniz (inventou 1671, construiu 1694)
 - computador capaz de somar e multiplicar
 - criou o mecanismo de engrenagens do "vai-um" usado até hoje
- Avanços nas calculadoras de mesa -> 1890 máquinas podiam:
 - acumular resultados parciais
 - armazenamento e reentrada automática de resultados passados (memória)

- imprimir resultados em papel
- Charles Babbage (Prof. 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 p/ construí-la
 - 1823: iniciou 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 (Jacquard)
 - 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 (820dC)
 - a máquina foi finalmente construída pelo governo inglês nos anos 1990 (e funciona!)
 - URL do vídeo do Roy?
- Herman Hollerith, 1890
 - criou cartões perfurados para uso no censo americano
 - tecnologia levou à criação da International Business Machines (IBM)
- MARK 1, criada em 1937 por Howard Aiken, prof. 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 Atanasoff, 1939
 - calculadora com válvulas a vácuo (240 válvulas)
 - resolvia equações lineares, diferenciais e de balística
 - manipulava números binários
- Rumo a 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.
 - 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>.
 - 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.
- ENIAC, 1945
 - considerado o primeiro computador eletrônico
 - números de 10 dígitos decimais
 - 300 multiplicações ou 5000 somas por segundo
 - 17486 válvulas, alguma queimava quase que diariamente
 - 6000 comutadores manuais e centenas de cabos usados na programação
 - programação era muito difícil
- Arquitetura do ENIAC

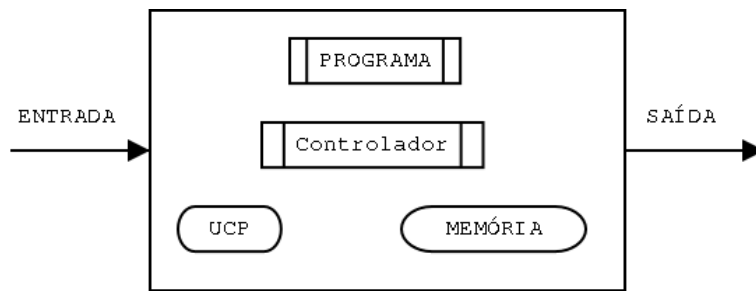


Figura 2.1: Arquitetura do ENIAC

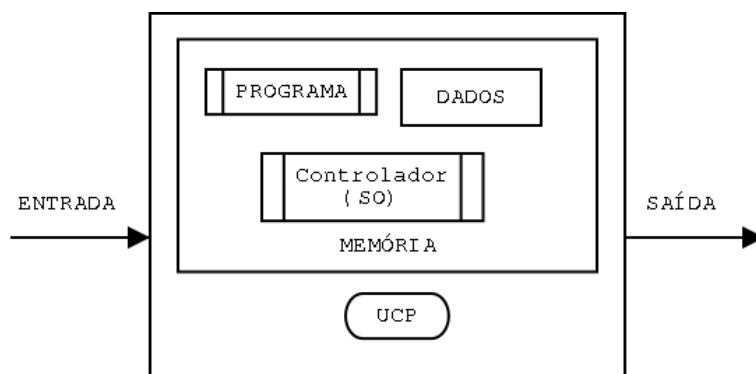


Figura 2.2: Arquitetura de Von Neumann

- programa especificado manualmente em "hardware" com conexões semelhantes àquelas que as velhas telefonistas utilizavam
 - memória de dados separada do controle e do programa
 - o controle é formado por circuitos eletro-eletrônicos
- 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 (indicar início de sistema operacional) (hoje em dia existe memória ROM, RAM, flash RAM...)
 - 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 até hoje (VLSI)
 - 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 Califórnia 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 Apple em garagem ~ 1975, 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)
 - Começa o movimento do software livre
- Anos 90
 - Microsoft pega carona na explosão de vendas de PCs, utiliza técnicas de marketing agressivas (consideradas por alguns estudiosos como moralmente questionáveis) para controlar mercado de software, estabelecendo quase um monopólio em certas áreas.
 - Intensifica-se o movimento por software livre
 - Nasce o Linux e uma nova forma de desenvolvimento de software baseada em comunidades distribuídas através da Internet.
 - No final da década, 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
 - Sistemas Embutidos
 - Computação Ubíqua

2.2 E a evolução do software?

Paralelamente á 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 ao 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 (algorithmic 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, não para entrada e saída: foi criada COBOL (Common Business Oriented Language)
- 1964, criação do Basic (Beginners All-purpose Symbolic Instruction Code);
- 1965, criação de uma linguagem específica para a simulação (SIMULA-1) por Ole-Johan Dahl and Kristen Nygaard. 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, linguagem Smalltalk (desenvolvida por Alan Kay, da Xerox), OO ganha força;
- 1983, criadas extensões de C incluindo suporte para OO: C++ e Objective-C

- 1987, linguagens baseadas em scripts, p.ex. Perl, desenvolvida por Larry Wall. Ferramentas de UNIX como sed e awk não eram suficientes;
- 1994, Java é divulgada como a linguagem para a Internet;

Capítulo 3

Conversor de Temperaturas

Quais novidades veremos nesta aula?

- 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 especificavam o comportamento dos atores no decorrer da peça. A linguagem Java permite que nós 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
    {
        // aqui vem a descrição do que o bandeirinha faz quando recebe esta mensagem
    }
    VenceuOTime (NomeDoTime)
    {
        // aqui vem a descrição do que o bandeirinha faz quando recebe esta mensagem
    }
}
```

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 isso 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 40C para Fahrenheit.

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

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

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

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

- 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 dentro do DrJava, podemos digitar o seguinte na janela do interpretador:

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

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

- 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.

- Note que devido à limitação acima, se tivéssemos escrito a fórmula como $\frac{9}{5} * C + 32$ o cálculo seria errado pois $\frac{9}{5} = 1$ se considerarmos apenas a parte inteira da divisão. Assim, o programa calcularia apenas $1 * C + 32$ o que seria errado.

Para corrigir isso, 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 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 15 casas decimais. Por comodidade, daqui para frente sempre que precisarmos de números fracionários, vamos utilizar o tipo `double`.

3. Terceira tentativa: conversor genérico usando `double`

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

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 dentro do DrJava, podemos digitar o seguinte na janela do interpretador:

```
Conversor4 anders = new Conversor4 ();
anders.celsiusParaFahrenheit (37.8);
anders.fahrenheitParaCelsius (-20.3);
```

Note que, para que o seu programa fique bem legível e elegante, é muito importante o alinhamento dos abres 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.

3.3 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:

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

Capítulo 4

Testes Automatizados

Quais novidades veremos nesta aula?

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

4.1 Testes

Desde o início, a computação sempre esteve sujeita a 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 frase célebre de Edsger Dijkstra 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. Logo, intuitivamente, quanto mais testes fizermos em um programa, mais confiantes podemos ficar com relação ao seu funcionamento. Por outro lado, nós podemos usar o próprio computador para nos ajudar, isto é, podemos criar testes automatizados. Ao invés de fazermos os testes “na mão”, faremos com que o computador seja capaz de verificar o funcionamento de uma sequência de testes. Veremos nesta aula 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, eles eram feitos de forma manual pelos próprios programadores, os grandes testadores eram os usuários finais. É interessante notar que isto ainda acontece com alguns produtos ainda hoje. Com o aparecimento da Engenharia de Software ficou clara a necessidade de se efetuar 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 em se 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 a 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 nós 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 não é atribuído, o sistema atribui valor 0 automaticamente.

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* vai ser impresso na tela a cada sucesso, isso 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 final dos testes. Ela é importante no caso em que todos os testes dão certo para que o usuário saiba que já terminamos a execução de todos os testes.

Uma forma de simplificar os comandos de impressão é usar a própria entrada como parâmetro, isto 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 tc = 10.0;
        double tf = 50.0;

        if (c4.celsiusParaFahrenheit(tc) != tf)
            System.out.println("C-> F não funciona para " + tc);
        if (c4.fahrParaCelsius(tf) != tc)
            System.out.println("F-> C não funciona para " + tf);
        tc = 20.0;
        tf = 68.0;
        if (c4.celsiusParaFahrenheit(tc) != tf)
            System.out.println("C-> F não funciona para " + tc);
        if (c4.fahrParaCelsius(tf) != tc)
            System.out.println("F-> C não funciona para " + tf);
```

```

tc = 101.0;
tf = 213.8;
if (c4.celsiusParaFahrenheit(tc) != tf)
    System.out.println("C-> F não funciona para " + tc);
if (c4.fahrParaCelsius(tf) != tc)
    System.out.println("F-> C não funciona para " + tf);
System.out.println("Final dos testes");
return 0;
}
}

```

Agora, vamos treinar o que aprendemos com alguns exercícios. Em particular, o exercício 3 de refatoração é muito importante.

4.2 Exercícios

1. 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.
2. Escreva uma classe `TestaConversor3` para testar a classe `Conversor3`.
3. Refatorar a classe `TestaConversos4` de modo a eliminar as partes repetidas do código. *Dica:* Crie um método que realiza o teste das duas funções.
4. 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);
}

```

5. *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 `ContasNaoConfiavel` dada abaixo:

```

class ContasNaoConfiavel
{
    double calculaQuadrado(double x);
    double calculaCubo(double x);
}

```

4.3 Resoluções

3

```
int testePontual(double tc, double tf)
{
    Conversor c4 = new Conversor4();

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

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

4

```
Contas a = new Contas();
double x = 4.0;

if (a.calculaQuadrado(x) != 16.0)
    System.out.println("Não funciona para calcular o quadrado de 4");
if (a.calculaCubo(x) != 64.0)
    System.out.println("Não funciona para calcular 4 ao cubo");
```


Capítulo 5

Métodos com vários parâmetros

Quais novidades veremos nesta aula?

- Métodos que devolvem nada (`void`);
- Métodos com vários parâmetros;
- Atributos.

5.1 Métodos com vários parâmetros

Na última aula nós já vimos um método que recebia vários parâmetros. Apesar de não estarmos apresentando nenhuma novidade conceitual vamos reforçar a possibilidade da passagem de mais de um parâmetro.

Assim como na passagem de apenas um parâmetro, é necessário que o valor passado seja do mesmo tipo do que o valor 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, na chamada de um método devemos respeitar a mesma 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 3.1415 * raio * raio;
    }
}
```

Nota Linguística: 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 3.1415 * 2.0 * raio;
}
```

O seguinte trecho de código calcula e imprime a área e 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));
```

Vejam 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));
```

Mas, 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);
}
}

```

Entretanto fica faltando algum método para carregar os valores dos lados nas variáveis lado1 e lado2.

```

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

```

Nota Linguística: void em inglês significa o vazio, nada, ou nulidade.

O funcionamento de um objeto da classe Retângulo 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 Retângulo, ou carregar diversos lados. Veja abaixo as duas opções:

```

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 3.1415 * raio * raio;
    }
    double calculoPerímetro()
    {
        return 3.1415 * 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 Exercícios

Métodos com vários parâmetros

1. Calcular a média aritmética de 4 notas e dizer 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”.

Atributos

1. 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 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.
```

2. 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 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 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 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 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.

Capítulo 6

if else encaixados

Quais novidades veremos nesta aula?

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

6.1 if else encaixados

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 TrianguloRetangulo
{
    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:

```
TrianguloRetangulo1 r = new TrianguloRetangulo1 ();
r.verificaLados (1, 1, 1);
r.verificaLados (3, 4, 5);
```

Limitações:

1. mesmo que um `if` seja verdadeiro, ele executa os outros `ifs`. Em particular, se tivéssemos um triângulo retângulo para o qual vários desses `ifs` fossem verdadeiros, ele imprimiria esta mensagem várias vezes (será que isso é possível neste caso específico???)
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 TrianguloRetangulo2
{
    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 TrianguloRetangulo3
{
    int a, b, c;
    void carregaLados (int l1, int l2, int l3)
    {
        a = l1;
        b = l2;
        c = l3;
    }
    int calculaPerimetro ()
    {
        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.");
}
}
```

6.2 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.
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) `raízesReais()` deve imprimir as raízes reais.
 - (d) `raízesImaginárias()` 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. **[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 a seguir.

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.

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##`.

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.)

Capítulo 7

Programas com Vários Objetos

Quais novidades veremos nesta aula?

- Programas com vários objetos

7.1 Programas com Vários Objetos

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

1. Vários objetos do mesmo tipo

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

rosa.cor("vermelha");
rosa.aroma("muito agradável");
margarida.aroma("sutil");
laranjeira.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:

```
class Prisma
{
    double altura;
    double areaDaBase;

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

    void carregaAreaDaBase(double a)
    {
        areaDaBase = a;
    }

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

class Quadrado
{
    double lado;

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

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

class TrianguloRetangulo
{
    double cateto1;
    double cateto2;

    // note a indentação super-compacta!
    void carregaCateto1(double c) { cateto1 = c; }
    void carregaCateto2(double c) { cateto2 = c; }

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

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

```

Quadrado q = new Quadrado();
TrianguloRetangulo tr = new TrianguloRetangulo();
Prisma prismaBaseQuadrada = new Prisma();
Prisma prismaBaseTriangular = new Prisma();

q.lado(10);
tr.cateto1(20);
tr.cateto2(30);

prismaBaseQuadrada.carregaAltura(3);
prismaBaseTriangular.carregaAltura(1);

prismaBaseQuadrada.carregaAreaDaBase(q.area());
prismaBaseTriangular.carregaAreaDaBase(tr.area());

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 tem o mesmo volume");

```

Nota sobre o interpretador do DrJava: para conseguir digitar todos os `ifs` encaixados no interpretador do DrJava sem que ele tente interpretar cada linha em separado, é preciso utilizar `Shift+Enter` ao invés de apenas `Enter` no final de cada linha dos `ifs` 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.

7.2 Exercício

1. 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:

- `rendimentosDePessoaFisica(double);`
- `rendimentosDePessoaJuridica(double);`
- `rendimentosDeAplicacoesFinanceiras(double);`
- `rendimentosNaoTributaveis(double);`
- `double totalDeRendimentosTributaveis();`

Em seguida, escreva uma classe `TabelaDeAliquotas` que possua:

- um método `aliquota()` 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.

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 define os vários rendimentos de uma pessoa e calcula o imposto a pagar.

Eis a tabela do IR 2004:

Rendimentos Tributáveis	Alíquota	Parcela a deduzir
Até R\$ 12.696		
De R\$ 12.696,01 a R\$ 25.380	0.15	R\$ 1.904,40
acima de R\$ 25.380	0.275	R\$ 5.076,90

2. 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 valorDaDivida = valorEmprestado + (a.a(meses, TAXA)*valorEmprestado);
        System.out.println("Dívida de " + valorDaDivida + " real(is), " +
            "calculada com taxa de " + TAXA + "% ao mês.");
    }
}
```

- O que fazem os métodos `a` (da classe `A`) e `b` (da classe `B`)? Não precisa entrar em detalhes. DICA: consulte a página <http://java.sun.com/j2se/1.4.1/docs/api/java/lang/Math.html>
- 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).
- Acrescente alguns comentários no código do método `b`.
- 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. Se você não sabe o que é uma constante, imagine que é um atributo cujo valor não pode ser alterado.

Capítulo 8

Laços e Repetições

Quais novidades veremos nesta aula?

- 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 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 das aulas de MAC-110: você já tem um conversor pronto. Basta então usá-lo para montar a tabela. Você chama então o DrJava e começa uma nova seção iterativa.

```
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)
21.111111111111111
> c.fahrenheitParaCelsius(80)
26.666666666666668
> c.fahrenheitParaCelsius(90)
32.222222222222222
> c.fahrenheitParaCelsius(100)
37.777777777777778
> 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` ao invés 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 isto 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 testar 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*, testa a condição (que ainda é verdadeira) e executa as instruções dentro das chaves. Esse processo continua até que *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 testada, 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 Fahrenheit => 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

Vejam 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 verificar se um número é primo podemos usar um laço que testa 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 na próxima aula.

```
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 numeroDeDivisores = 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)
                numeroDeDivisores = numeroDeDivisores + 1;
            candidatoADivisor = candidatoADivisor + 1;
        }

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

8.4 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 -1 se x for negativo.
2. Crie uma classe contendo um método que devolve a média dos valores $1, 2, 3, \dots, N$, onde N é o valor absoluto de um número fornecido ao método.
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 `imprimeFibonacciAte50`, que imprime os 50 primeiros números da seqüência de Fibonacci. Seqüência de Fibonacci:
- $F1 = 1$;
 - $F2 = 1$;
 - $F_n = F(n-1) + F(n-2)$, para todo $n > 2$, n inteiro.

O método deve então imprimir $F1, F2, F3, \dots, F49, F50$.

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 numeroDeDivisores = 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)
            numeroDeDivisores = numeroDeDivisores + 1;
    }

    // Imprime a resposta.
    if (numeroDeDivisores == 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 em $x/2$ ou mesmo ao chegar na 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 a_1 e a_2 , estritamente positivos, com $a_1 \geq a_2$, 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 a_1 e a_2 , com $a_1 \geq a_2$, quero o $m.d.c.(a_1, a_2)$.
- Calcule $a_3 = a_1 \% a_2$.
- Se $a_3 = 0$, fim. A solução é a_2 .
- Calcule $a_4 = a_2 \% a_3$.
- Se $a_4 = 0$, fim. A solução é a_3 .
- Calcule $a_5 = a_3 \% a_4$.
- ...

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 nesta aula?

- 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 de comparações, usando os operadores `==`, `>=`, `<=`, `>`, `<` e `!=`, entre variáveis e/ou constantes. Uma característica interessante em linguagens de programação é que as condições são na verdade 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
> boolean comp2 = a < a + 1
> comp2
true
>
```

Com isso acabamos de introduzir mais um tipo de variáveis, 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). Em português este tipo de variável é chamada de variável *booleana* em homenagem ao matemático inglês George Boole (1815-1864).

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.

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
```

¹Daí vem a explicação para o fato do sinal de diferente fazer referência ao ponto de exclamação.


```
> !(comp1 && (5 > 0))
true
> int a = 10
> (a > 5) && (!comp1)
true
> boolean comp2 = (a > 5) && comp1
> comp2
false
>
```

Também podemos “misturar” operadores aritméticos e lógicos, 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 lógica, misturando tipos. Isto não faria sentido em Java.

Outra coisa que pode ser feita é a criação de métodos que devolvem um resultado lógico. 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 na aula passada, para devolver a resposta (se o número é primo ou não), ao invés de imprimir na tela, teríamos o seguinte método `éPrimo`:

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean ÉPrimo(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;
    }

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

9.2 Precedência

Como acabamos de apresentar vários operadores novos, devemos estabelecer a precedência entre eles. Lembre 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 todos os operadores vistos em aula, 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 a 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 `TianguloRetangulo3` 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` da aula passada.

```
/**
 * Verifica se um número inteiro positivo é primo ou não.
 */
boolean ÉPrimo(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/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) && (x != 1) && (x != 0) && (x != -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: tente 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 l)
    {
        limiteInferior = l;
    }

    /**
     * Verifica se um número inteiro positivo é primo ou não.
     */
    boolean ÉPrimo(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/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) && (x != 1) && (x != 0) && (x != -1))
            return true;
        else
            return false;
    }
}

```

```

/**
 * A cada chamada, calcula um novo primo seguindo ordem crescente.
 */
int próximoPrimo ()
{
    // Busca o primeiro primo depois do limite.
    limiteInferior = limiteInferior + 1;
    while (!ÉPrimo(limiteInferior))
        limiteInferior = limiteInferior + 1;

    return limiteInferior;
}

```

Não deixe de brincar um pouco com objetos da classe `GeradorDePrimos` para entender melhor como ela funciona! Agora um desafio para você: usando o método `próximoPrimo`, escreva um novo método `void imprimePrimos (int quantidadeDePrimos)` que imprime uma dada quantidade de números primos a partir do `limiteInferior`. Experimente executar o método passando 50 como parâmetro.

9.4 Exercício:

- Escreva uma classe `TrianguloRetangulo` com um método denominado `defineLados(float x1, float x2, float x3)` que recebe três valores e verifica se eles correspondem aos lados de um triângulo retângulo. Em caso afirmativo, o método retorna `true`, caso contrário ele retorna `false`. Note que o programa deve verificar quais dos três valores corresponde à hipotenusa. Construa duas versões do método, uma contendo três `ifs` e outra contendo apenas um `if`!
- Escreva uma classe `Brincadeiras` que possua 3 atributos inteiros. Escreva um método para carregar valores nestes atributos e, em seguida, escreva os seguintes métodos:
 - `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>`.
 - `imprime()` que imprime o valor dos 3 atributos.
 - `imprimeEmOrdemCrescente()` que imprime o valor dos 3 atributos em ordem crescente.
- 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.

Capítulo 10

Mergulhando no “while”

Quais novidades veremos nesta aula?

- 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 torne-se 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 envolvem apenas polinômios que podem ser calculados facilmente usando laços. Pena que vocês só vão aprender isso na disciplina de Cálculo mais para o fim do semestre. De qualquer forma, vejamos a expansão de algumas dessas funções:

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

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 `int fat(int 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;
        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;
}

```

10.4 Exercícios

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

2. Implementar 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 item anterior.
3. Implementar 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 $\ln(1+x^2)$.
4. 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 cosseno e o seno do ângulo passado como parâmetro em radianos ($\text{graus} * \text{PI}/180 = \text{radianos}$). Compare os resultados com os obtidos com nossas implementações de `sen()` e `cos()`.

Capítulo 11

Caracteres e cadeias de caracteres

Quais novidades veremos nesta aula?

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

11.1 Um tipo para representar caracteres

Até hoje já vimos diferentes tipos de variáveis, vimos 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 Caracter1
{
    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 caracter usamos aspas simples (`'`). Também podemos ver que os caracteres minúsculos são diferentes do mesmo caracter 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:

```
*****
```

```

EEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEE
EEE
EEE
EEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEE
EEE
EEE
EEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEE
*****

```

Os diferentes métodos que imprimem caracteres pulando, ou sem pular linha serão usados

Para desenharmos letras onde é necessário intercalar espaços e letras em uma única linha, a implementação fica um pouco mais longa. 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 nós já havíamos feito algumas operações com strings. Como 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 caracter 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 `Strings`.
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
```

11.3 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 caracter `X` no final.
2. Escrever um método que devolveo número de ocorrências da vogal `a` em uma frase contida em uma `String`.
3. Implemente um método que determina a freqüência relativa de vogais em uma `String`. Considere as letras maiúsculas e minúsculas não acentuadas.

Capítulo 12

Manipulando números utilizando diferentes bases

Quais novidades veremos nesta aula?

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

Nesta aula, aprendemos como extrair e processar os dígitos de um número inteiro. Exemplos:

1. Imprimir um número natural dígito por dígito.
2. Dado um número natural n , na base binária, convertê-lo para a base decimal. *Exemplo:* Dado 10010 a saída deve ser 18 pois: $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 18$.

Veja agora com cuidado a implementação da classe Conversão que converte números binários para decimais e vice-versa. Não é muito fácil entender o funcionamento de seus métodos; sugerimos fortemente então, que você tente rastrear¹ 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;
        }
    }
}
```

¹Rastrear 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.

```

    pot2 = pot2 * 2;
}
return dec;
}

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

    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;
}
}

```

12.1 Exercícios

1. Dado um número natural n , contar quantos dígitos ele tem. Não esqueça de considerar 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, vamos 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, ou não palíndromo. Um número palíndromo é um número que lido de trás para frente é o mesmo número de quando lido normalmente, por exemplo:
 - 78087
 - 1221
 - 11111
 - 3456 não é palíndromo!!!

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 algarismo romanos em um inteiro e vice-versa.
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 13

Arrays (vetores)

Quais novidades veremos nesta aula?

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

13.1 *Arrays* (vetores)

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

Note que, no exemplo acima, a linha que define o *array* `temperaturas` faz duas coisas simultaneamente e que podem ser separadas em dois passos:

`double[] temperaturas;` define um novo *array* chamado `temperaturas` e que irá conter valores do tipo `double`.

Por enquanto, o *array* está vazio. Mas quando fazemos `temperaturas = new double[31];` estamos especificando 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 Linguística: 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.

Vejam 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++;
    }
}
```

13.1.1 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` e que contém o comprimento do *array*. Desta forma, o método `calculaQuadrados` acima poderia ser re-escrito 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.

13.1.2 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", "Jupiter", "Saturno",
                    "Urano", "Netuno", "Plutão"};
```

13.2 Criação de Programas Java

Até esta aula, 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 nós 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-se 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
```

java é o nome do interpretador Java BrincadeirasComArrays é o nome da classe que será carregada e cujo método main será executado. 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 *Strings*.

13.3 Exercícios

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[] frequenciaRelativa(int[] v, int n)` que recebe um vetor contendo números inteiros no intervalo $[0, n]$ e devolve um vetor contendo a frequência 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 subsequência do vetor de tamanho maior.
Ex: O vetor $[9, 5]$ é uma subsequência 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 $[7, 2]$

Capítulo 14

for, leitura do teclado e conversão de Strings

Quais novidades veremos nesta aula?

- for (mais um comando de repetição)
- Leitura de dados do teclado (classe SavitchIn)
- Conversão de Strings para números

14.1 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 na aula passada:

```
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 ao 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 ao laço ficam na mesma linha o que facilita a visualização e o entendimento de quem olha para o código.

14.2 Leitura do Teclado

Na aula passada, 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. Java oferece algumas formas de se fazer isso mas nenhuma delas é muito simples. Para facilitar a vida de iniciantes em programação Java, 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 Califórnia em San Diego.

Em seus EPs e exercícios, para usar a classe `SavitchIn`, você pode baixá-la do sítio do Prof. Savitch ¹.

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 Web². Mas, em MAC 110, nós vamos nos concentrar apenas nos seguintes métodos:

¹<http://www.cs.ucsd.edu/users/savitch/java/SavitchIn.txt>

²<http://www.sinc.sunysb.edu/Stu/fmquresh/SavitchIn.html>

```

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();
}

```

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.

14.3 Conversão de String para números

Você deve ter notado que o método `main` recebe sempre `Strings` 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);

```

14.4 Exercícios

1. Usando o comando `for`, escreva um método que recebe um *array* de `doubles` 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. O método preenche esse array com valores e estes valores estarão disponíveis, após o retorno do método, para quem o chamou. Isso funciona em Java apenas quando passamos objetos como parâmetro (e array é um objeto). Quando passamos tipos básicos, como `int`, isso não funciona, ou seja, o valor modificado pelo método não é visível a quem o chamou.*

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 produto vetorial (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 vetorial e imprime o resultado.

Capítulo 15

Laços Encaixados e Matrizes

Quais novidades veremos nesta aula?

- repetições encaixadas
- matrizes (arrays multi-dimensionais)

15.1 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 interações. No exemplo acima, o `println` é portanto executado $5 \times 3 = 15$ 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 numeroDeAlunos, numeroDeTurmas, turma, aluno;
        double nota, soma;
        System.out.print ("Você quer calcular as médias de quantas turmas? ");
        numeroDeTurmas = SavitchIn.readLineInt ();
        for (turma = 1; turma <= numeroDeTurmas; turma++)
        {
            soma = 0;
            nota = 0;
```


15.3 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 [][] iteracao()
{
    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 = iteracao(); // a matriz da iteração anterior é recolhida
                             // pelo coletor de lixo.
    }
}
}

```

```

.....
...*....*   .....   .....   .....   .....
...*....*   .....   .....   .....   .....
...*....*   .....   .....   .....   .....
...*....*   .....   .....   .....   .....
...*....*   .....   .....   .....   .....

```



```

> 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 $A_{n \times n}$ 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 frequencia(int v[])` que imprime a frequência absoluta dos elementos em `v`.
Dica: para calcular a frequência 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 $A_{m \times n}$ 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 16

Busca e ordenação

Quais novidades veremos nesta aula?

- Busca em array
- Ordenação em arrays

16.1 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. Lembrando do nosso EP, pense na busca de um CD pelo título. Sabendo onde o CD está podemos recuperar outras informações, como as faixas que o compõe, 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 retorna 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 vetor.

Por fim, devemos destacar que há uma certa sutileza no código acima. Note que no segundo caso não poderíamos acessar no vetor na posição i . Isso não ocorre devido a 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”.

16.2 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 vetor ordenados. Um bom exemplo é a busca. Caso o vetor esteja ordenado pela “chave” de busca é possível fazer algo bem mais eficiente do que a busca seqüencial descrita acima.

Mas como podemos ordenar os elementos de um vetor?

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. Agora, 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, 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.

Aposto que a solução imaginada poderia ser descrita em Java por um dos algoritmos abaixo. Neles desejava-se ordenar os elementos de 0 até `final` no vetor de inteiros `numeros`. Cada algoritmo é escrito como um método.

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

```
void selecaoDireta(int [] numeros)
{
    int i, j, indiceDoMinimo, temp;
    int fim = numeros.length;

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


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

```

void insercaoDireta(int []numeros, int fim)
{
    int i, j, numeroAInserir;

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

```

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

```

void bolha(int []numeros, int fim)
{
    int i, j, temp;

    for(i = fim - 1; i > 0; i--)
        // Varre o vetor desde 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 (numeros[j-1] > numeros[j])
            {
                // Troca os dois de lugar
                temp = numeros[j-1];
                numeros[j-1] = numeros[j];
                numeros[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 definidas 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 `imprimeVetor` para que o computador mostre todos os elementos válidos de seu vetor. Esse método é fácil, basta varrer o vetor e pedir a cada passo que o computador imprima o elemento atual (ou a parte dele que nos interessa ver).

16.3 Exercícios

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

Capítulo 17

Busca binária, fusão e o que são as variáveis

Quais novidades veremos nesta aula?

- Busca binária
- Fusão de duas seqüências ordenadas
- Diferenças entre variáveis que representam tipos primitivos e aquelas que representam tipos complexos

17.1 Busca binária

Nesta aula vamos ver 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, que tem atualização anual, 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, operação freqüente, muito mais rápida. A idéia é bem simples. Como a seqüência está ordenada, sabemos que o item do meio deixa metade dela de cada lado. Se, em uma primeira passagem, nós simplesmente perguntarmos qual a posição do valor buscado com relação a esse elemento intermediário, 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 parte do fim.
3. O objeto buscado fica depois do objeto intermediário: a busca deve continuar apenas na parte final.

Ou seja podemos, usando o elemento intermediário, eliminar metade do trabalho a cada passo. Isso faz com que a busca fique muito mais rápida! Vejamos com fica a implementação da busca binária para um array de inteiros. Este código pode ser facilmente adaptado para double e cadeias de caracteres.

```
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;
}
```

17.2 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;
}
```

17.3 O que guardam as variáveis?

Em primeiro lugar, vamos usar o nome variável para designar qualquer nome associado a dados em um programa. Isso inclui os atributos de objetos, parâmetros de métodos e variáveis locais. Em Java, uma variável associada a um tipo primitivo (`int`, `double`, `boolean` e `char`) tem um comportamento levemente diferente de variáveis associadas a objetos complexos (`String`, `arrays`, objetos pertencentes a classes definidas pelo programador, etc.) Vamos tentar explicar as diferenças usando a forma como o Java armazena os dados. Os dados de tipo primitivo são todos muito simples, ocupando pouco espaço de memória. Neste caso vale a pena guardar nessas 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 já está em `b`. Isso explica porque a mudança em um parâmetro de tipo primitivo dentro de métodos não se reflete fora. O parâmetro alterado não tem nenhuma ligação com a variável que estava em seu lugar na chamada além da cópia inicial do seu valor.

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 simplesmente a posição na memória na 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 que o objeto precisa ser usado, Java o localiza a partir de uma referência para ele armazenada em uma variável. 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 ocorreu porque `a` e `b` são referências para um mesmo objeto na memória. Como dois apelidos para mesma pessoa. Qualquer coisa que ocorre com um 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 é também um apelido.

17.4 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 lhe auxilie a determinar qual algoritmo é mais eficiente: a busca binária ou a busca ternária.

Capítulo 18

Construtores e especificadores de acesso

Quais novidades veremos nesta aula?

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

18.1 Construtores

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

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 tem 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 hoje não havíamos usado o `this` pois tomamos cuidado para escolher os nomes de parâmetros e variáveis locais diferentes dos nomes dos atributos.

18.2 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 curso. 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`. Se os atributos do `Conta` forem declarados da seguinte forma:

```
class Conta
{
    private String titular;
```

```
private double saldo;  
private String RG;  
...  
}
```

Não haverá mais 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.

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;`

18.3 Exercícios

1. (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 19

Interfaces

Quais novidades veremos nesta aula?

- Interfaces

19.1 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.

19.2 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 você 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 nasce ();
    void passeiePelaTela ();
    void durma ();
    double peso ();
}
```

O programador que for implementar o morcego terá que dizer explicitamente que vai usar a interface `Animal`, isto é feito através da palavra chave `implements`. Como o objetivo é de 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 nasce ()
    {
        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 nasce ()
    {
        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 respectivos nomes: `Ornitorrinco.java`, `Morcego.java` e `Zebra.java`.

Mas, o uso da interface é um pouco mais amplo, 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 ZoologicoVirtual
{

```

```

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

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

    cicloDeVida(z1);
    cicloDeVida(z2);
    cicloDeVida(m1);
    cicloDeVida(ol);
}
}

```

Veja o exemplo do painel de iterações abaixo:

```

> ZoologicoVirtual.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`, logo chamadas do tipo `z2.contaListras()`; não são válidas, mas chamadas a `z1.contaListras()` 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);
    bicharada[1] = new Zebra(101);
    bicharada[2] = new Morcego();
    bicharada[3] = new Ornitorrinco();
}

```



```

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

```

Ficou bem melhor, não?

19.3 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 frequentemente). Vejam as duas interfaces seguintes:

```

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

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

```

Agora vamos pensar em três classes: Ave, Onibus e Aviao. As classes Ave e Onibus 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 Onibus 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 Aviao que implementa as duas interfaces:

```

class Aviao 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 Onibus();
        Voador v = new Ave();
        Avião a = new Aviao();

        t.entramPessoas();
        t.saemPessoas();
        v.voa();           // bate asas
        v.aterrissa();
        // Com o Onibus 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 “Programar para as interfaces, não para as implementações”. Em outras palavras, toda vez em 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.

19.4 Um exemplo mais sofisticado

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

```

class Fruta
{
    double peso;
    double valor;
    String nome;

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

    void imprime()
    {
        System.out.println(nome + " pesa " + peso
            + "g e custa " + valor + " reais");
    }
}

```

Queremos criar um vetor de `Frutas` e ordena-lo primeiro por `valor` 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 16.

```

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 ordenaPorValor()
    {
        int i, j;
        Fruta aInserir;

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

```

```

        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 valor");
    xepa.ordenaPreco();
    xepa.imprime();
    System.out.println("Em ordem de peso");
    xepa.ordenaPeso();
    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 apenas 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 valor e o nome da fruta, respectivamente.

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

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

class ComparaNome implements ComparadorDeFrutas
{
    public boolean éMenor(Fruta a, Fruta b){ return (a.nome.compareTo(b.nome) == -1);}
}
```

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 (frutas [j-1], aInserir)))
            {
                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 valor:");
    ComparadorDeFrutas cmp = new ComparaValor();
    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 super-condensada.
    xepa.imprime();
}
}

```

19.5 A Importância de Interfaces

O conceito de interfaces na programação orientada a objetos é muito importante e o seu uso adequado trás 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 quando da utilização de 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 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 em 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 em 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`.

19.6 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 as seguintes interfaces:

```
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 Berçario
{
    Bebe [] ListaDeBebês;

    // aqui vão outros métodos para inserção e
    // remoção de bebês da lista

    void imprimeDadosSobreBebe (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 ();

        imprimeDadosSobreBebe (bebes[IDdoBebe], 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 `ZoologicoVirtual.fazFuncionar()` de forma a usar a classe `SavitchIn` 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 20

Herança

Quais novidades veremos nesta aula?

- Herança

20.1 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 re-utilização de código. Se conseguirmos escrever código que é re-utilizado 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 re-utilizadas diariamente por milhares de programadores ao redor do 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, acrescentando-se alguns métodos e atributos adicionais.

Esta re-utilização de código com extensões e modificações, pode ser obtida em linguagens orientadas a objetos através do conceito de Herança. A figura 20.1 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.

20.2 Terminologia de herança

- Dizemos que Pessoa é a *superclasse* de Aluno e de Professor.

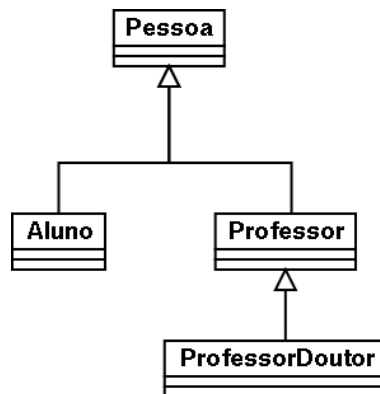


Figura 20.1: Diagrama de herança

- 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 class Pessoa.
- Dizemos que Pessoa é *pai* (ou *mãe*) de Professor; e que Professor é *filho* (a) de Pessoa.

20.3 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`.

20.4 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 20.2 uma hierarquia de classes para representar os diversos tipos de seres vivos.

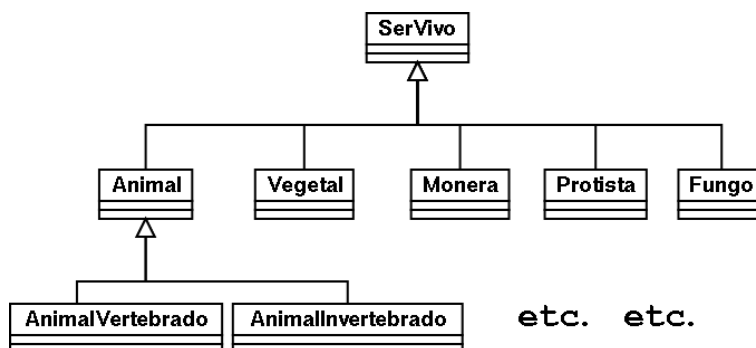


Figura 20.2: Hierarquia de classes representando os seres vivos

20.5 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:

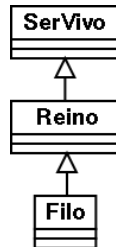


Figura 20.3: Hierarquia errada

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.

20.6 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* ("over-rides") ao método da superclasse.

20.7 Exercícios

1. Retornando à hierarquia de classes apresentada na figura 20.2, pense em quais métodos e atributos deveriam estar presentes nas classes superiores da hierarquia e quais deveriam estar nas partes inferiores da hierarquia.
2. 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 21

Javadoc

Quais novidades veremos nesta aula?

- Documentação com javadoc

21.1 Javadoc

Vimos nas últimas aulas 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 (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 FiguraGeometrica
{
    private double lado;
    private String cor;

    public Quadrado(double l, String c)
    {
        lado = l;
    }
}
```

```

    cor = c;
}

public double calculaArea()
{
    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 `calculaArea` e `calculaPerimetro`.

Cada um desses itens 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 FiguraGeometrica
{
    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 calculaArea()
    {
        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 FiguraGeometrica
 */
public class Quadrado implements FiguraGeometrica
{
    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 calculaArea()
{
    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 21.1 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.

Bom, agora que temos no 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 sub-diretório `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 SUMMARY: INNER FIELD CONSTR METHOD	FRAMES NO FRAMES 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:
nã - 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 SUMMARY: INNER FIELD CONSTR METHOD	FRAMES NO FRAMES DETAIL: FIELD CONSTR METHOD

Figura 21.1: Documentação gerada pelo Javadoc

Capítulo 22

O C que há em Java

Quais novidades veremos nesta aula?

- Veremos como escrever programas na linguagem C a partir de nossos conhecimentos de Java.

22.1 O C que há em Java

Veremos nessa aula como são os programas na linguagem C e como podemos escrevê-los usando o que aprendemos em nosso curso de introdução à computação. Em resumo podemos pensar que um programa em C é uma única classe Java sem a presença de atributos e composta apenas com métodos estáticos. Fora isso o resto é perfumaria. Vamos pensar um pouco quais são as consequências 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;

        System.out.print("De um temperatura em Fahrenheit: ");
        far = SavitchIn.readDouble();
        cel = fahrenheitParaCelsius(far);
        System.out.println("A temperatura em Celsius é: " + cel);
        System.out.print("De um temperatura em Celsius: ");
        cel = SavitchIn.readDouble();
        far = celsiusParaFahrenheit(cel);
        System.out.println("A temperatura em Fahrenheit é: " + far);
    }
}

```

Vejamos como ficaria esse programa na linguagem C:

```

/* Sempre coloque a proxima linha no inicio do programa. */
/* Ela permite que você imprima na tela e leia do teclado. */
#include <stdio.h>

/* Como nao ha classes simplemente apagamos a referencia a ela. */
/* 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 não 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 é: %f\n", cel);
printf("De um temperatura em Celsius: ");
scanf("%lf", &cel);
far = celsiusParaFahrenheit(cel);
printf("A temperatura em Fahrenheit é: %f\n", far);

/* Para nos esse retorno nao serve para nada, e um topico avancado. */
return 0;
}
```

22.2 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 `ScavitchIn` 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). Diferente do que ocorria com Java, não podemos usar a soma para concatenar cadeias. Como apresentar então variáveis no meio a string que será impressa pelo `printf`? Usamos nesse caso uns 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` pula 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);
```

22.3 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 seja um vetor de 100 inteiros basta:

```
int a[100];
```

A mesma coisa para matrizes:

```
int a[100][100];
```

22.4 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 ele 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 variáveis 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 um matriz, você deve informar na declaração da função qual o número de linhas que a matriz tem. Por exemplo

```
void umaFuncao(int umVetor[], int umaMatriz[100][])
```

22.5 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 o número e imprime a sua mediana.

```
#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 sub-vetor 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 fácil. */
    printf("A mediana e: %d\n", numeros[(i - 1)/2]);

    /* O tal return inutil. */
    return 0;
}
```


Apêndice A

Utilizando o Dr. Java

A.1 Introdução

DrJava é 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, por isso alguns recursos desejáveis ainda não estão disponíveis. Entretanto, os recursos mais simples que ele fornece já são apropriados para os nossos objetivos nesta disciplina. Planejamos então utilizar o DrJava para escrevermos nossos programas Java deste semestre.

Objetivos

Esperamos com esta aula introduzir o uso do DrJava. O conteúdo da aula 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.

Nesta aula você aprenderá a utilizar o DrJava para

- escrever, compilar, manipular e depurar classes/objetos simples;
- gravar e reutilizar os arquivos que descrevem seus objetos.

Questões mais gerais sobre uso do computador, como impressão do código fonte e manipulação de arquivos fora do DrJava, dependem muito do ambiente que você utiliza e por isso não são abordados nesta aula. No IME estão à disposição dos alunos dois laboratórios que possuem ambientes bem distintos: as salas Pró-Aluno <<http://www.linux.ime.usp.br>> e o CEC. Sugerimos que você consulte os documentos de ajuda disponibilizados pela própria administração de cada laboratório para resolver esse tipo de dúvida.

A.1.1 Como obter, instalar e executar o DrJava

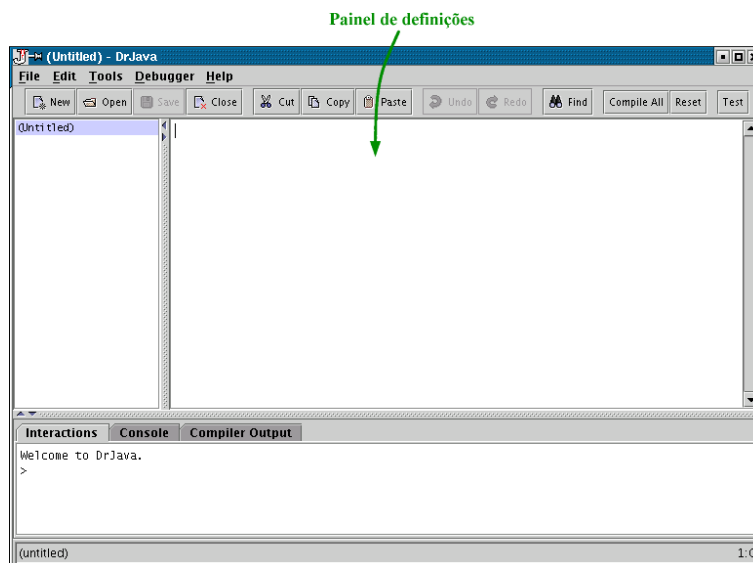
Essas informações estarão disponíveis no sítio da disciplina.

A.2 Conversor de Temperatura simples

Como exemplo inicial, vamos construir a classe `Conversor` vista em aula. Lembrando, cada objeto dessa classe converte apenas a temperatura de 40 graus celsius para a correspondente em graus fahrenheit. Para isso, ao receber a mensagem `celsiusParaFahrenheit`, ele 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 destaque do texto.

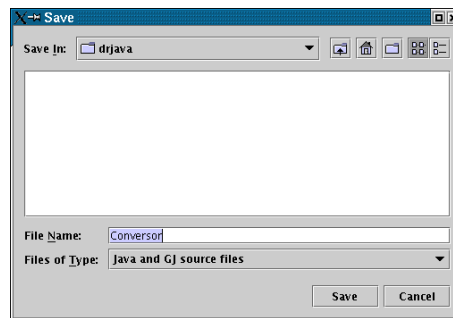
Precisamos criar um arquivo novo para conter o código da nossa classe. Para isso, bastaria escolher a opção **New** do menu **File**. Porém, quando abrimos o DrJava, um novo arquivo sem nome já foi criado, então podemos usá-lo nesse momento (em vez de criar um novo). Sendo assim, digite o seguinte código 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 item **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, isso será feito implicitamente quando finalizarmos (mas não há problema em digitar o sufixo mesmo assim).

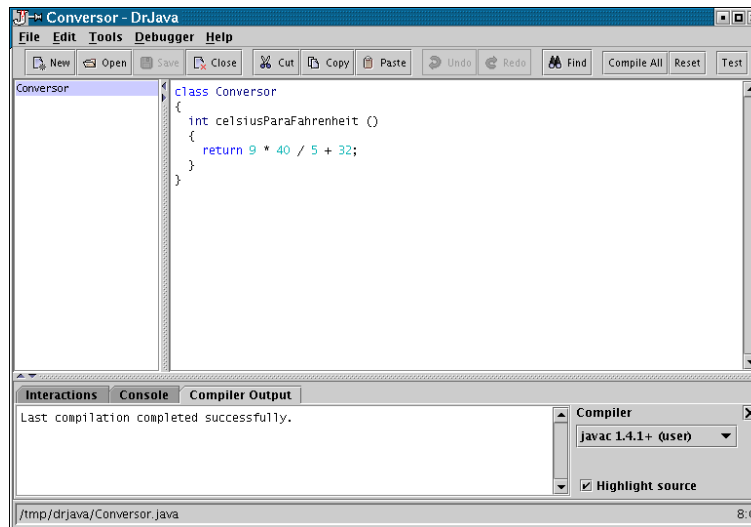
Para confirmar a gravação, basta clicar no botão **Save**. As modificações futuras podem ser gravadas com o comando **Save**, sem precisar escolher o nome do arquivo novamente.

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 `Converter`. 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.

```
Converter conv = new Converter();
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.3 Tratando erros

Utilizaremos agora a classe `Converter4` vista em aula 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.

Para esta aula dirigida, porém, usaremos um código um pouco diferente em relação ao dado em aula normal.

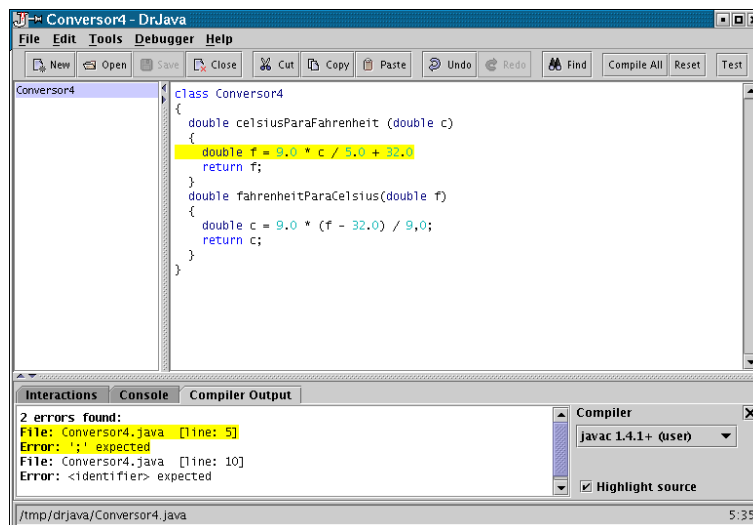
Erros no código

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 descobirmos 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`.

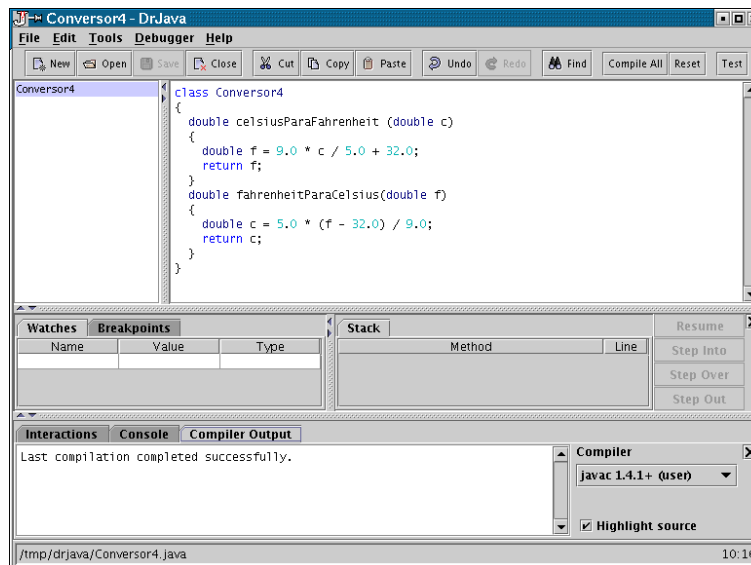
Vimos em aula uma maneira de detectarmos tais erros através de testes. Mas depois de detectarmos, precisamos descobrir a causa deles. Veremos a seguir uma ferramenta útil para essa tarefa.

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)
```

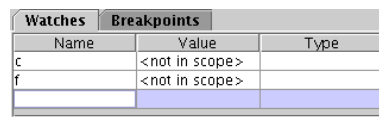
A execução do método será interrompida antes da linha 10 ser executada. O destaque azul da linha indica isso. Para continuar a execução passo a passo (linha a linha), execute o comando *step over* algumas vezes (pressionando a tecla <F11>).

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>.

Então faça isso para as variáveis *c* e *f*. Teremos algo como a próxima figura.



Name	Value	Type
c	<not in scope>	
f	<not in scope>	

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**.

Referências Bibliográficas

- [1] Philippe Breton, *História da Informática*, Editora Unesp, 1987
- [2] Walter Savitch, *Java, An Introduction to Computer Science & Programming*, second edition, Prentice Hall, 2001
- [3] Samuel N. Kamin, M. Dennis Mickunas and Edward M. Reingold, *An Introduction to Computer Science Sing Java*, McGraw-Hill, 1998